Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Efficient Metadata Update Techniques for Storage Systems

A Dissertation Presented

by

Maohua Lu

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2010

Stony Brook University

The Graduate School

Maohua Lu

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

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

Jie Gao – Chairperson of Defense
Professor, Department of Computer Science

Robert Johnson
Professor, Department of Computer Science

Dr. David D. Chambliss
Researcher Staff
IBM, San Jose, CA

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Efficient Metadata Update Techniques for Storage Systems

by

Maohua Lu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2010

The simple read/write interface exposed by traditional disk I/O systems is inadequate for low-locality update-intensive workloads because it limits the flexibility of the disk I/O systems in scheduling disk access requests and results in inefficient use of buffer memory and disk bandwidth. We proposed a novel disk I/O subsystem architecture called Batching mOdifications with Sequential Commit (BOSC), which is optimized for workloads characterized by intensive random updates. BOSC improves the sustained disk update throughput by effectively aggregating disk update operations and sequentially committing them to disk.

We demonstrated the benefits of BOSC by adapting it to 3 different storage systems. The first one is a continuous data protection system called Mariner. Mariner is an iSCSI-based storage system that is designed to provide comprehensive data protection on commodity hardware while offering the same performance as those without any such protection. With the help of BOSC in metadata updating, the throughput of Mariner has less than 10% degradation compared to that without metadata updating.

Flash-based storage is the second storage system we leveraged BOSC. Because of the physics underlying the flash memory technology and the coarse address mapping granularity used in the on-board flash translation

layer (FTL), commodity flash disks exhibit poor random write performance. We designed LFSM, a Log-structured Flash Storage Manager, to eliminate the random write performance problem of commodity flash disks by employing data logging and BOSC in metadata updating. LFSM is able to reduce the average write latency of a commodity flash disk by a factor of more than 6 under standard benchmarks.

As a third example, we applied BOSC to a scalable data de-duplication system based on the incremental backups. Each input block is de-duplicated by comparing its fingerprint, a collision-free hash value, with existing fingerprints. A range-based block group, called segment, is the basic unit to preserve data locality for incremental backups. We propose four novel techniques to improve the de-duplication throughput with minimal impact on data de-duplication ratio (DDR). BOSC is employed to eliminate the performance bottleneck due to committing segment updates to the disk.

This dissertation is dedicated to my dear wife, Dr. Yang Yang.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Motivation 1
1.2 Continuous Data Protection (CDP) 3
1.2.1 Incremental File-System ChecKer (iFSCK) 6
1.2.2 User-level Versioning File System (UVFS) 7
1.3 Write Optimization for Solid State Disk (SSD) 8
1.4 Data Deduplication atop Block Change Tracking 9
1.5 Contributions 12
1.6 Outline 12

2 Related Work 14

2.1 Related Work of Efficient Metadata Update 14
2.1.1 Write Buffering 14
2.1.2 Reorganization of Index Structures 16
2.1.3 Smart Data Placement and Management 19
2.1.4 Logging and Batching of Individual Records 21
2.1.5 Insert/Update Optimizations of State-of-Art DBMSs 21
2.1.6 Insert and Update Optimizations Exploring Domain Specific Knowledge 24
2.2 Related Work of Asynchrony 26
2.3 Related Work of CDP 26
2.4 Related Work of File Versioning 29
2.5 Related Work of Consistency Check and Ensurance 29
2.6 Related Work of Write Optimization of SSD 30
2.6.1 Flash Translation Layer (FTL) 30
2.7 Related Work of Data De-duplication 32

3 Batching mOdification and Sequential Commit (BOSC) 36
3.1 Update-Aware Disk Access Interface 36
3.2 Batching Modifications with Sequential Commit 37
3.2.1 Low-Latency Synchronous Logging 37
3.2.2 Sequential Commit of Aggregated Updates 40
3.2.3 Recovery Processing 41
3.2.4 Extensions 42
3.3 BOSC-Based B^+ Tree 42
3.4 Performance Evaluation 45
3.4.1 Evaluation Methodology 45
3.4.2 Overall Performance Improvement 46
3.4.3 Sensitivity Study 49
3.4.4 Hash Table 53
3.4.5 R Tree and K-D-B Tree 54
3.4.6 Read Query Latency 56
3.4.7 Logging and Recovery Performance 56

4 Continuous Data Protection (CDP) 59
4.1 System Architecture 59
4.2 Low-Latency Disk Array Logging and Logging-based Replication . . . 60
4.3 Transparent Reliable Multicast (TRM) 64
4.3.1 Multicast Transmission of Common Payloads 65
4.3.2 Common Payload Detection 66
4.3.3 Tree-Based Link-Layer Multicast 66
4.4 Incremental File System ChecKing (iFSCK) 67
4.4.1 Snapshot Access 67
4.4.2 Ensuring File System Consistency 68
4.5 User-level Versioning File System (UVFS) 72
4.5.1 Overview 72
4.5.2 Version Query Processing Algorithms 73
4.5.3 Optimizations 74
4.6 Performance Results and Analysis 75
4.6.1 Evaluation Methodology 75
4.6.2 Low-Latency Disk Logging 77
4.6.3 Array of Logging Disks 79
4.6.4 Sensitivity Study 80
4.6.5 Impact of iSCSI Processing 83
4.6.6 Impact of Modified Two-Phase Commit Protocol 85
4.6.7 TRM Evaluation 87
4.6.8 Effectiveness of BOSC 89
4.6.9 Performance Evaluation of UVFS 91
4.6.10 Performance Evaluation of iFSCK 99

5 Random Write Optimization for SSD **105**
5.1 Design 105
5.1.1 Overview 105
5.1.2 Disk Write Logging 106
5.1.3 Ensuring Metadata Consistency 108
5.1.4 Garbage Collection 109
5.2 Prototype Implementation 111
5.3 Performance Evaluation 113
5.3.1 Methodology 113
5.3.2 Performance Results 115
5.3.3 Sensitivity Study 118
5.3.4 End-to-End Performance Evaluation 121
5.3.5 Effectiveness of BOSC 123

6 Scalable and Parallel Segment-based De-duplication for Incremental Backups **126**
6.1 Background and Motivation 126
6.2 Design 129
6.2.1 Data Flow for De-duplication 130
6.2.2 Segment Identification 131
6.2.3 Variable-Frequency Sampled Fingerprint Index 132
6.2.4 Segment-based Container 134
6.2.5 Segment-based Summary 136
6.2.6 Put Them Together for Stand-alone De-duplication 138
6.2.7 Garbage Collection (GC) 140
6.2.8 Parallel De-duplication 146
6.2.9 Optimizations 154
6.3 BOSC in De-duplication 156
6.4 Performance Evaluation 156
6.4.1 Evaluation Methodology 157
6.4.2 Segment-based De-duplication 162
6.4.3 Variable-Frequency Sampled Fingerprint Index 163
6.4.4 Segment to Container Association 167
6.4.5 Segment-based Summary 168
6.4.6 BOSC to Update Containers 169
6.4.7 Conclusion of Design Decisions 170

7 Conclusion and Future Work **172**
7.1 Summary of BOSC 172
7.2 Summary of Applications of the BOSC Scheme 173
7.3 Conclusion of De-duplication for Incremental Backups 175
7.4 Future Research Directions 176

Bibliography **177**

List of Tables

3.1 Read Query Performance for 4 Index Structures 56
3.2 Breakdown of Recovery Processing Time 57

4.1 Metadata Updates of File Operations 69
4.2 Impact of T_{wait} on Modified Trail 79
4.3 Elapsed Time for NFS Operations under Mariner 87
4.4 The cost of NFS operations 93
4.5 Comparison with Ext3cow 97

5.1 Breakdown of Write Latency for Random Writes 114
5.2 Read Latency of LFSM with On-Disk and In-Memory BMT 117
5.3 Compression Regions of Three Workloads 117

6.1 Comparison of Different GC Techniques 143
6.2 Statistics of Workgroup Trace 157

List of Figures

3.1 Architecture of BOSC . . . 38
3.2 Insert/Update Throughput of BOSC-based B^+ Tree and Vanilla B^+ Tree . . . 45
3.3 Comparison among 4 Configurations . . . 48
3.4 Throughput with Varied Leaf Node Size . . . 49
3.5 Throughput with Varied Record Size . . . 50
3.6 Throughput with Fixed Record/Leaf Ratio . . . 51
3.7 Throughput with Varied Index Size . . . 52
3.8 Throughput with Varied Memory Size . . . 53
3.9 Insert/Update Throughput of BOSC-based Hash Table and Vanilla Hash Table . . . 54
3.10 Insert Throughput of BOSC-based R Tree/K-D-B Tree and Vanilla R Tree/K-D-B Tree . . . 55
3.11 Recovery Time . . . 57

4.1 Architecture of Mariner . . . 60
4.2 Architecture of Augmented Trail . . . 62
4.3 Flow Graph of Modified Two-Phase Commit Protocol . . . 62
4.4 Data Flow of an iSCSI-based TRM System . . . 65
4.5 Demonstration of Skew in TRM . . . 66
4.6 Example of *iFSCK* to Extract the Metadata Redo List . . . 71
4.7 Write Latency for Different Inter-Request Intervals . . . 75
4.8 Write Latency for Different Request Sizes . . . 78
4.9 Impact of T_{switch} on Modified Trail . . . 80
4.10 Impact of Log Disk Number on Modified Trail . . . 80
4.11 Impact of T_{wait} on Modified Trail . . . 81
4.12 Impact of the Recalibration Frequency on Modified Trail . . . 81
4.13 Impact of Input Rates on Modified Two-Phase Commit . . . 82
4.14 Latency for Modified Two-Phase Commit for Synthetic Workload . . . 83
4.15 Latency for Modified Two-Phase Commit for Two Real Traces . . . 84
4.16 Postmark Throughput of Local Disks and Mariner's Storage . . . 85
4.17 Postmark Transaction Rate of Local Disks and Mariner's Storage . . . 86
4.18 Effect of Packet Aggregation . . . 88
4.19 Effect of Transferred File Size . . . 89

4.20 Throughput Comparison among Four CDP Systems 90
4.21 Performance of Synthetic Workload for Searching Utilities 94
4.22 The Search Time for the SPECsfs Workload 96
4.23 Elapsed Time of *iFSCK* Under 3 Different Consistency Levels 98
4.24 Elapsed Time Comparison of *iFSCK* and Vanilla FSCK for Ext2 . . . 99
4.25 Modified File System Metadata Blocks by *iFSCK* 100
4.26 Elapsed Time Comparison of *iFSCK* and Vanilla FSCK for Ext3 . . . 101
4.27 Elapsed Time of *iFSCK* for Real Workloads 102
4.28 Modified Blocks of *iFSCK* for Real Workloads 103

5.1 Architecture of LFSM 106
5.2 Effectiveness of LFSM's Write Optimization 115
5.3 I/O Rate Comparison of On-Disk and In-Memory BMT 117
5.4 LFSM's Write Latency under Three Traces with Varied T_{free} 118
5.5 LFSM's Write Latency under Three Traces with Varied T_{update} 119
5.6 LFSM's write latency under three traces with varied L_{queue} 120
5.7 Latency and Throughput of SPECsfs Workload 120
5.8 Ratio of Physical Writes over Logical Ones with Varied HList Length Limit 121
5.9 Latency and Throughput of TPC-E Workload 122
5.10 Performance Overhead due to GC for SPECsfs Workload 123
5.11 Performance Overhead due to GC for TPC-E Workload 124
5.12 Effectiveness of BOSC in LFSM 125

6.1 Data Flow of the Standalone Data De-duplication 131
6.2 An Example of Segment Identification 133
6.3 Container Structure 134
6.4 Offset Interval Distribution of a Store Segment 135
6.5 Algorithm of Standalone De-duplication 137
6.6 Metadata Updating for Garbage Collection 142
6.7 Algorithm of Garbage Collection 144
6.8 An Example of Garbage Collection 145
6.9 Parallel Garbage Collection 147
6.10 Architecture of Parallel De-duplication 148
6.11 An Example of Distributed Segment Identification 149
6.12 Example of Inconsistency between Garbage Collector and Mapping Updater 150
6.13 Consistency between Garbage Collection and L2P Mapping Updater . 151
6.14 Algorithm of Distributed De-duplication 153
6.15 Distribution of Store Segments and Duplicate Segments 158
6.16 The Cumulative Distribution Probability of Blocks in Segments with Varied Segment Length 159
6.17 Ratio of Duplicate Segments with Varied K 160

6.18 The Identification of Basis Sharing Unit with Varied Reference Count Threshold . . . 161
6.19 The Data De-duplication Ratio with Varied T_{BSU} . . . 162
6.20 Temporal Locality of Duplicate Blocks . . . 163
6.21 The comparison of LRU policies . . . 164
6.22 The Data De-duplication Ratio Comparison . . . 165
6.23 Comparison of Segment-clustered Container and Stream-based Container 166
6.24 Percentage of Container Splits . . . 167
6.25 Savings of Container Reading . . . 168
6.26 Memory Overhead of Summary Segment Cache . . . 169
6.27 Average Queue Length of Per-Container Queue . . . 170

Chapter 1

Introduction

1.1 Motivation

Random update-intensive disk I/O workloads, such as those resulting from index updates in data deduplication, user-generated content management and OLTP applications, have been the most challenging workload for storage stack designers for the last several decades. Mainstream physical storage devices do not perform well under these workloads and thus require the upper-layer disk I/O subsystem to carefully schedule incoming disk access requests to mitigate the performance penalty associated with such workloads. In fact, even emerging flash memory-based solid-state disks perform poorly in the face of these workloads, sometimes faring ever worse when compared with magnetic disks. A large body of previous research efforts on disk buffering [1, 2], caching [3–5], and scheduling [6, 7] attest the need of such optimizations. However, these optimizations are generally ineffective for random update-intensive workloads, because the working set is too large to fit in available system memory, and the locality in the input disk access stream is inherently low. This paper describes an *update-aware* disk access interface and a novel disk I/O system exploiting this new interface that together effectively rise up to this performance challenge.

Traditional disk I/O systems expose a read/write interface for higher-layer system software, such as a file system or a DBMS, to access data stored on disks. The granularity of reads and writes ranges from disk blocks [8, 9] to more sophisticated constructs such as objects [10, 11]. Regardless of access granularity, these simple read/write interfaces are not adequate for random update-intensive workloads for the following two reasons. First, disk I/O systems tend to minimize the response time for disk read requests in the hope of reducing their critical path latency. However, for a disk update request, which typically involves a disk read followed by a disk write, the leading read request should be serviced like a write, and it is acceptable to delay such reads in order to optimize the disk access performance. Unfortunately, a simple read/write interface does not allow the disk I/O system to distinguish between plain

reads and reads associated with updates. Second and more importantly, an ideal way to optimize the performance of a set of updates to a disk block is to aggregate and apply them to the disk block when it is brought into memory. When updates to a disk block originate from multiple processes, the simple read/write interface prevents such aggregation, because the disk I/O system does not know how to apply individual updates and has no choice but to wake up each issuing process to perform the associated update.

To overcome the performance problem associated with random update-intensive workloads, we propose a new disk access interface that allows applications of a disk I/O system (a) to explicitly declare a disk access request as an `update` request to a disk block, in addition to a standard `read` or `write` request, and (b) to associate with a disk update request a call-back function that performs the actual update against its target disk block. With disk update requests explicitly labeled, the disk I/O system can batch multiple of them, including the implicit reads contained within, in the same way as it does with write requests. With access to application-specific update functions, the disk I/O system can directly apply updates to an in-memory disk block on behalf of the processes issuing the update requests, greatly increasing the flexibility in disk request scheduling.

This new disk access interface enables a novel disk I/O system architecture called *BOSC* (Batching mOdifications with Sequential Commit), which sits between storage applications, e.g., DBMS process or file system, and hardware storage devices, and is specifically optimized for 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 disks to bring in disk blocks whose queue is not empty. When a disk block is fetched into memory, BOSC applies all of its pending updates to it in one batch. Compared with traditional disk I/O systems, BOSC boasts three important performance advantages. First, BOSC dedicates its buffer memory to queuing incoming requests rather than buffering target disk blocks and is therefore capable of buffering more requests per unit of memory, as the size of a disk block's pending update request queue is typically much smaller than the size of the disk block itself. For example, assume each record in a B^+ tree is 24 bytes, then a queue of 20 insert/update requests is about 500 bytes, which is much smaller than a B^+ tree page, which is typically 4 KB or 8 KB. Second, BOSC brings in needed blocks by *sequentially* traversing the disks, and thus significantly reduces the performance cost of fetching each accessed block. This is made possible by BOSC's ability to defer processing of reads in update requests and accumulate update requests that share the same target disk block. Finally, when a disk block is fetched into memory, BOSC applies all pending updates to it in an FIFO fashion without involving processes originally issuing these update requests. This design greatly simplifies the implementation complexity of BOSC and decreases the memory requirements of disk blocks with pending updates and thus the overall memory pressure.

To test the effectiveness and generality of BOSC, we have ported four different

database indexes on top of BOSC, including B^+ tree, R tree, K-D-B tree and Hash Table, and compared their performance when running on a vanilla disk I/O system that supports the conventional read/write interface, and on BOSC, respectively. Across all indexes and test workload mixes, the *sustained* performance improvement of BOSC over the vanilla disk I/O system is quite impressive, between one to two orders of magnitude. In addition, through a space-efficient low-latency logging technique, BOSC is able to achieve this performance improvement while delivering the same durability guarantee as if each update request is written to disk synchronously.

In the following subsections, we further explore 3 areas where efficient metadata update is critical, sketch the use of BOSC in all these areas, present our contributions in these areas, and finally outline the organization of this report. In concrete, in subsection 1.2, the *Continuous Data Protection* (CDP) and two related features built atop CDP are introduced. In subsection 1.3, I introduce the research challenges for *Solid State Disk* (SSD) and how the metadata update performance is enhanced by employing BOSC. Subsection 1.4 proposes a new technical solution for data deduplication in backup storage systems with BOSC in improving the performance of metadata update. In subsection 1.5, I summarize the contributions of our work in the area and the outline of this report is presented in the subsection 1.6. .

1.2 Continuous Data Protection (CDP)

As modern enterprises increasingly rely on digital data for continuous and effective operation, data integrity and availability become the critical requirements for enterprise storage systems. Replication is a standard technique to improve data integrity and availability, but typically incurs a performance overhead that is often unacceptable in practice. We implemented an iSCSI-based storage system called *Mariner*, which aims to support comprehensive data protection while reducing the associated performance overhead to a minimum. More specifically, *Mariner* uses *local mirroring* to protect data from disk and server failures, and *remote replication* to protect data from site failures. In addition, *Mariner* keeps the before image of every disk update to protect data from software failures, human errors or malicious attacks.

A *Mariner* client, for example a file or a DBMS server, interacts with three iSCSI storage servers: a master storage server, a local mirror storage server and a logging server. When a *Mariner* client writes a data block, the write request is sent to these three servers. The data block is synchronously committed on the logging server, and then asynchronously committed on the master server and the local mirror server. In addition, the logging server is responsible for remote replication, which is also done asynchronously. When a *Mariner* client reads a data block, the read request is only sent to the master server, which services the request on its own.

Mariner supports block-level continuous data protection (CDP), which creates a new version for every disk write request and thus allows roll-back to any point in time. As more and more data corruption is caused by software bugs, human errors

and malicious attacks, CDP provides a powerful primitive for system administrators to correct the corrupted data. *Mariner*'s logging server is responsible for archiving the before image of every disk write within a period of time (called the *protection window*) so that it can undo the side effects of any disk write in the protection window.

To reduce the performance penalty associated with CDP and data replication, *Mariner* modifies the track-based logging (*Trail*) technique [12]. *Trail* was originally designed to reduce the write latency of locally attached disks and adapts the idea to the network storage environment where *Mariner* operates. The original *Trail* requires a log disk in addition to a normal disk, which hosts a write-optimized file system. By ensuring that the log disk's head is always on a free track, *Trail* could write the payload of a disk write request to wherever on the track the disk head happens to be. Once this write is completed, *Trail* returns a completion signal so that the high level software can proceed. Therefore, the latency of a synchronous disk write is reduced to only the sum of the *controller processing time* and the *data transfer delay*. However, the original *Trail* design is inadequate for *Mariner* for two reasons. First, the disk utilization efficiency of the design is too low to meet CDP's demanding log space requirement. Second, the design switches a disk's head to the next free track after servicing a request and thus incurs substantial disk switching costs. To address these problems, *Mariner* makes four modifications to *Trail*. First, after the payload of a logical disk write (W1) is written to a log disk, the payload is kept for a sufficiently long period of time so that the following write (W2) against the same data block can be undone T days after W2 is performed. Here T is the length of the protection window, and the payload of W1 is the before image of W2. Second, *Mariner* batches multiple logical disk write requests that arrive within an interval as much as possible into a physical disk write in order to amortize the fixed overhead associated with each physical disk write, and allows multiple physical disk writes to be written to a track until the track's utilization efficiency exceeds a certain threshold. Third, *Mariner* exploits an array of log disks to amplify both the log capacity and the effective logging throughput in terms of physical I/O rates. Finally, *Mariner* uses a modified version of two-phase commit protocol to propagate the effect of each logical disk write consistently to all replicas while minimizing the write latency visible to the software.

In addition to involving more servers, N-way data replication also introduces N times as much load on the storage client's CPU, memory and network interface. It is possible to move this load to the storage area network using special hardware such as SANTap [13], which replicates a disk write request coming into a SAN switch across multiple predefined ports in a way transparent to the storage client. However, this approach requires special and proprietary hardware support. *Mariner* uses a software-only approach called *Transparent Reliable link-layer Multicast (TRM)* to approximate the hardware-based in-network replication supported by SANTap. More specifically, TRM achieves in-network replication by exploiting link-layer tree-based multicast available in modern commodity Ethernet switches.

In *Mariner*, to provide flexible and efficient access to logged disk block versions,

it maintains two index structures: a TL index whose key is $\langle LBN, timestanp\rangle$ and an LT index whose key is $\langle timestamp, LBN\rangle$, where LBN and $timestamp$ are a block update's target logical block address and timestamp, respectively. When a host (i.e. a file server or a DBMS server) emits a block update request, the request is sent to the corresponding network storage server as well as the block-level CDP server that protects it. When a block-level CDP server receives a block update request, it logs the new block version to disk and inserts a new entry into each of these two index structures. Whenever a disk block version expires, i.e., falls outside the protection window, the corresponding LT and TL index entries should be removed.

Every block version corresponds to a point in a two-dimensional address space whose coordinate is the unique combination of its associated timestamp and target LBN. In general, the TL index is useful when users want to access all LBNs updated within a time range, whereas the LT index is useful when users want to access the most recent version of a LBN as of a particular point in time. In *Mariner*, the TL index is useful for its incremental file system consistency check mechanism [14] and for reclaiming LT and TL index entries, both of which need to know the set of block versions created within a time interval; the LT index is the basis for historical snapshot access, i.e., given a target LBN and a snapshot time T, the LT index returns the address of the physical block holding the most up-to-date version of LBN as of T.

Because these two indexes are too large to fit into memory for very large block-level CDP servers, they are required to persist on disk. Since timestamps of block updates are monotonically increasing and timestamps are the most significant part of the TL index's key, there is substantial locality in the updates to the TL index. Therefore, updating the on-disk TL index does not cause much of a performance problem. However, updates to the LT index do not exhibit such locality, because the most significant part of its key is the LBN and the target LBNs of temporally adjacent block updates are not necessarily close to one another. Without any optimization, a block update triggers three to four disk I/O operations in *Mariner*: logging the update itself (a sequential write), updating the TL index (a mostly sequential write) and updating the LT index (a random write possibly preceded by a read). Because *Mariner* employs a highly efficient disk logging scheme [14], which can complete a 4KB logging operation under 0.5 msec, index update becomes *Mariner*'s major performance bottleneck.

We employ BOSC that successfully solves the performance problem associated with index updates in *Mariner*. The BOSC scheme batches update operations to a persistent index and asynchronously commits them to disk using mostly sequential disk I/O. With BOSC, not only is the cost of bringing in each on-disk index page lowered, but such cost is also amortized over multiple index update operations. By employing BOSC, *Mariner*'s index update mechanism achieves the best of both worlds: strong consistency of synchronous index update and high throughput of sequential disk I/O.

Modified *Trail*, TRM and BOSC work together to enable *Mariner* to provide

comprehensive data protection at a performance cost that is almost negligible when compared with vanilla iSCSI storage servers without any protection. Detailed design and implementation details will be discussed in chapter 4.

The key advantage of block-level CDP is that its data protection service is readily available to a wide variety of file or DBMS servers from different vendors, because it applies update logging to a common access interface (i.e. iSCSI) shared by these servers. For the same reason, the high-level file system or DBMS operations that lead to individual disk writes are completely opaque to a block-level CDP system. This lack of knowledge of the high-level context behind each disk write limits the use of block-level CDPs to point-in-time block-level snapshots, and substantially decreases the practical appeal of block-level CDP systems.

To overcome the above limitations of existing block-level CDP systems, *Mariner* is designed to support file system-consistent point-in-time snapshots and provide users a file versioning capability similar to versioning file systems. Moreover, *Mariner* supports this file versioning capability in a way that is portable across all main-stream operating systems, including Linux, Solaris, and Windows XP. This paper describes the design, implementation and evaluation of *Mariner*'s user-level file versioning system, which is expected to bring forth the full value of block-level CDPs to their users.

In subsection 1.2.1, we sketch the design of *incremental File-System CheKer (iFSCK)*, a utility supporting file system-consistent point-in-time snapshots. In subsection 1.2.2, we introduce *User-level Versioning File System (UVFS)*, a utility to allow end users to navigate the data versions by leveraging the CDP capability.

1.2.1 Incremental File-System ChecKer (iFSCK)

Conventional data backup systems take a snapshot of the file/storage system periodically, so that the file/storage system can be restored to one of these snapshots in case it is corrupted. However, this approach is limited in terms of recovery point objective (RPO) and recovery time objective (RTO). That is, it cannot roll the file/storage system back to arbitrary points in time, and as a result its associated recovery time cannot be bounded because additional manual repair may be required after a programmatic roll-back. Continuous data protection (CDP) greatly improves the RTO and RPO of a data backup solution by keeping the before image of every update operation against a file/storage system for a period of time. Because CDP enables every update to be undoable, it supports arbitrary point-in-time roll-back.

A block-level CDP system [14–18] applies CDP at the disk access interface, i.e. every disk write operation from an application server (e.g. a file or DBMS server) to its back-end storage system within a *data protection window* [1] is logged and thus undoable. A key advantage of block-level CDP is that it can protect the data of arbitrary application servers without requiring any modifications to them.

[1] The data protection window is the time period within which every update is undoable, and is typically on the order of a week or a month.

Although existing block-level CDP systems can roll back the protected storage's image to arbitrary points in time in the past, none of them is able to guarantee that these images are consistent with respect to the metadata of the application servers whose data they are protecting. That is, when a user of a network file server whose data is protected by a block-level CDP system takes a point-in-time snapshot, the file system metadata in the returned snapshot may not be consistent with one another. As a result, even though block-level CDP supports flexible RTO for disk block-level images, it does not support flexible RTO for file-level images when the application server is a file server. This is a serious limitation of existing block-level CDP systems because it substantially decreases their practical utility and appeal.

Different application servers have different types of metadata, typically transparent to block-level CDP systems. To support arbitrary point-in-time metadata-consistent snapshots, exploiting application server-specific knowledge is inevitable. When the application server is a file server, one could apply a standard file system checker to block-level snapshots returned by a block-level CDP systems and transform them into a file system-consistent state. However, such checkers are too slow for some legacy file systems. This paper describes the design, implementation and evaluation of an *incremental* file system consistency checker called *iFSCK* that leverages file system-specific knowledge to convert a point-in-time snapshot into one that is guaranteed to be consistent with respect to file system metadata.

1.2.2 User-level Versioning File System (UVFS)

The lack of knowledge of the high-level context behind each disk write imposes two limitations on existing block-level CDP systems, which substantially decreases their practical appeal. First, when a user of a network file server protected by a block-level CDP system takes a point-in-time snapshot, the file system metadata in the returned snapshot may not be consistent with one another. This means that even though existing block-level CDP systems can support arbitrary point-in-time disk image snapshots within the data protection window, these snapshots are not necessarily file system-consistent. Second, none of the existing block-level CDP systems provide the same file versioning functionality as a versioning file system, because they lack the necessary file system metadata required to map disk blocks to files. As a result, a block-level CDP system cannot answer such questions as "what is the last version of `/a/b` before T", "how many versions of the file `/a/b` exist in $[T1, T2]$", etc.

We have developed a user-level file versioning system specifically designed to address these two limitations of existing block-level CDP systems. This system consists of a file versioning subsystem called *UVFS* (User-level Versioning File System) and an incremental file system consistency checker called *iFSCK* 1.2.1, which exploits file system-specific knowledge to convert a point-in-time disk image snapshot to be file system-consistent. This paper focuses specifically on the design, implementation and evaluation of *UVFS*, which augments a block-level CDP system with a file versioning

capability that is portable across all main-stream operating systems (Linux, Solaris, and Windows XP).

1.3 Write Optimization for Solid State Disk (SSD)

The recent commoditization of USB-based flash disks, mainly used in digital cameras, mobile music/video players and cell phones, has many pundits and technologists predict that flash memory-based disks will become the mass storage of choice on mainstream laptop computers in two to three years [19]. In fact, some of the ultra mobile PCs, such as AsusTek's Eee PC [20], already use flash disks as the only mass storage device. Given the much better performance characteristics and enormous economies of scale behind the flash disk technology, it appears inevitable that flash disks will replace magnetic disks as the main persistent storage technology, at least in some classes of computers.

Compared with magnetic disks, flash disks consume less power, take less space, and are more reliable because they don't include any mechanical parts. Moreover, flash disks offer much better latency and throughput in general because they work just like a RAM chip and do not incur any disk head positioning overhead. However, existing flash disk technology has two major drawbacks that render it largely a niche technology at this point. First, flash disk technology is still quite expensive, approximately $10/GByte, which is at least 20 times as expensive as magnetic disks. Indeed, at this price point, it is not uncommon that a flash disk costs as much as the computer it is installed on. Second, flash disk's performance is better than magnetic disk when the input workload consists of sequential reads, random reads, or sequential writes. Under a random write workload, flash disk's performance is comparable to that of magnetic disk at best, and in some cases actually worse. We believe the cost issue will diminish over time as the PC industry shifts its storage technology investment from magnetic to flash disks. However, flash disk's random write performance problem is rooted in the way flash memory cells are modified, and thus cannot be easily addressed. This paper describes the design and implementation of a log-structured flash storage manager (LFSM) that effectively solves the random write performance problem of commodity flash disks.

A flash memory chip is typically organized into a set of erasure units (typically 256 KBytes), each of which is the basic unit of erasure and in turn consists of a set of 512-byte sectors, which correspond to the basic units of read and write. Multiple sectors comprise a data block, which can be 4 KB or 8 KB in size. After an erasure unit is erased, writes to any of its sectors can proceed without triggering an erasure if their target addresses are disjoint. That is, after a sector is written and before it can be written the second time, it must be erased first. Because of this peculiar property of flash memory, random writes to a storage area mapped to an erasure unit may trigger repeated copying of the storage area to a free erasure unit and erasing of the original erasure unit holding the storage area, and thus result in significant performance overhead.

Moreover, flash disks typically come with a flash translation layer (FTL), which is implemented in firmware, maps logical disk sectors, which are exposed to the software, to physical disk sectors, and performs various optimizations such as wear leveling, which equalizes the physical write frequency of erasure units. This logical-to-physical map will require 64 million entries if it keeps track of individual 512-byte sectors on a 32-GB flash disk. To reduce this map's memory requirement, commodity flash disks increase the mapping granularity, sometimes to the level of an erasure unit. As a result of this coarser mapping granularity, two temporally separate writes to the same mapping unit, say an erasure unit, will trigger a copy and erasure operation if the target address of the second write is not larger than that of the first write, because a commodity flash disk cannot always tell whether a disk sector in an erasure unit has already been written previously or not. That is, if the N-th sector of a mapping unit is written, any attempt to write to any sector whose sector number is less than or equal to N will require an erasure, even if the target sector itself has not been written previously. Consequently, coarser mapping granularity further aggravates flash disk's random write performance problem.

To address the random write performance problem, LFSM converts all random writes into sequential writes to a log by introducing an additional level of indirection above the FTL. Because all commercial flash disks have good sequential write performance, LFSM effectively solves the random write performance problem for these disks in a uniform way without requiring any modifications to their hardware or FTL implementations. With this log-structured storage organization, LFSM needs to overcome two major challenges. First, LFSM still faces random writes because it needs to maintain a separate map for the level of indirection or translation it introduces and updates to this map are random. LFSM minimizes the performance overhead of the random writes to this map by using a technique called BUSC [21], *batching updates with sequential commit*. Second, to minimize the amount of copying whenever LFSM reclaims an erasure unit, it needs to allocate erasure units to logical blocks in such a way that logical blocks assigned to the same erasure unit have a similar life time and each erasure unit contains very few live logical blocks at the time when it is reclaimed.

1.4 Data Deduplication atop Block Change Tracking

Disk-based near-line storage systems are more and more popular with the increasing capacity and dropping price of the hard drives [22]. Different from online storage systems, near-line storage systems does not accommodate live data but only hosts snapshots of the live data, which allows flexible power-efficient management of the hard drives in the storage systems. In contrast to the off-line storage system, for example, tapes, near-line storage system can be accessed by the same interface as the on-line storage system. Applications of near-line storage systems include disk-to-disk

(D2D) backup [22] and storage archival [23]. For clarity, I denote the near-line storage system as the secondary storage system, and the online storage system as the primary storage system in the discussion throughout the paper.

Deduplication is an indispensable component for modern near-line storage systems for three reasons. First, data deduplication can reduce the space overhead. Enterprise users of near-line storage systems tend to store the same data over and over again onto the secondary storage systems. Without deduplication, redudant data can consume more than 500% of the storage space [24]. Second, data deduplication can improve the bandwidth efficiency in term of both network bandwidth and the disk I/O bandwidth. The bandwidth efficiency roots from two sources, 1) less backup traffic; 2) less replication traffic. In the former case, data duplicates do not need to be transferred to the secondary storage systems and only the sketch of the data is sent to the secondary storage systems [25]. In the latter case, replication of the secondary storage systems requires the bandwidth that is linear to the size of the storage, and data deduplication can greatly reduce the amount of replicated data. Third, lower number of disks ameliorates the power consumption because more disks need more power for powering the hard drives and cooling. As the power dominates the day-to-day expense of running a near-line storage systems [26], it becomes uttermost important to employ deduplication techniques to save the power.

For secondary storage systems, deduplication techniques at the block abstraction layer were proposed [27, 28] to deduplicate *streams* from different clients, where the full index for deduplication can not fit into memory. The backup clients blindly write out the whole snapshot images to the secondary storage systems. The secondary storage takes backups from the same client as a stream. The input stream is segmented into variable-sized chunks, and a hash called fingerprint is computed for each chunk. The secondary storage deduplicates chunks in the same stream and between different streams. Because deduplication is done against the whole snapshot image, the deduplication ratio can reach up to 20:1 as expected. Furthermore, because chunks are produced from a contiguous region of blocks within a region, fragmentation in the snapshot image can have a negative impact on the deduplication quality.

To reap all benefits of the data deduplication, it is inevitable to have a more intelligent client-side agent rather than the “dummy” client. For example, to avoid transferring duplicates to the secondary storage system, the client-side agent needs to segment the snapshot, transfer the fingerprints, figure out which chunks are duplicates and only transfer those chunks that are not duplicates. In general, pushing more intelligence to the client side has three benefits: 1) efficient bandwidth usage at the backup time, 2) reduced burden on the backup storage side because the client can assume preprocessing responsibility of the end-to-end deduplication computation, and 3) flexible deduplication strategies because the client has the full knowledge of the backup stream.

We explore new opportunities, challenges and solutions for data deduplication when the client-side agent has the non-obstructive knowledge of the backup streams. In concrete, we assume that the intelligent client-side agent has the following three

pieces of information regarding the backup stream. First, the client side can track changed blocks in a snapshot image since last snapshot (incremental change tracking). Second, the client can correlate changed blocks with the entity (i.e., a file) owning these blocks. Third, the high-level metadata (i.e., modification time of a file) is available when needed. The retrieval of these pieces of information is non-obstructive because all the information above can be retrieved by referring the high-level index structures [29, 30].

Two challenges arise in combining the client-side intelligence with the current deduplication techniques. First, because only incremental changes are pushed to the second storage system, it is not clear what is the exact source of data deduplication, and furthermore, what is the correct strategy to capture these deduplication sources effectively. Second, with all high-level information (i.e., files and related metadata), how to leverage all these information effectively so that we can improve both the deduplication quality and the deduplication throughput?

Ultimately, a good deduplication scheme should strike a reasonable balance between the deduplication quality and the deduplication throughput for arbitrary backup stream, including backup streams consisting of incremental updates. Data Domain's data deduplication technique [27] employs the full fingerprint index to achieve the maximal deduplication quality but does not provide any guarantee on the deduplication throughput. On the other hand, the sparse indexing scheme [28] can sustain a fixed deduplication throughput but the deduplication quality can decrease due to the sampling. In particular, if duplicates do not come with large chunks, a fixed sampling rate would miss some duplicates.

We propose to employ the temporal locality of incremental updates by varying the sampling rate based on the deduplication history. The hypothesis behind the proposal is that the incremental updates tend to be duplicates of recently-popular data from other clients as well as from the same client. If the hypothesis is true, we can have a higher sampling rate for the fingerprints of popular data, and have a lower sampling rate for those unpopular data. In particular, if stored data payload is popular recently, we'd rather have a sampling rate of 1 to achieve the highest degree of deduplication. In contrast, for unpopular data, the sampling rate can be dropped to a lower value because anyway there is less duplicate of the unpopular data.

To prevent the deduplication quality loss due to fragmentation of high-level entities (i.e., files), we propose to use the high-level entity (i.e., the file) as the basic unit of deduplication. Although file-level deduplication has a long history [31, 32] in deduplicating a whole file, deduplication of partial files containing only changed blocks has not yet been fully explored. Similar to the sampling method used in the sparse indexing scheme, we propose to sample fingerprints from a file, not from a block-level segment.

1.5 Contributions

In summary, this technical report and the research work it is based on make the following research contributions:

1. For *BOSC*, we developed a new disk access interface that supports disk update as a first-class primitive and enables the specification of application-specific update functions that are invoked by the disk I/O system,

2. For *BOSC*, we built a disk I/O system framework that effectively commits pending update requests in a batched fashion, and drastically improves the physical disk access efficiency by using only *sequential* disk I/O to bring in the requests' target disk blocks,

3. For *BOSC*, we have complete prototype implementation of the BOSC disk I/O system and four database index implementations that are built on top of BOSC, including B^+ tree, R tree, K-D-B tree, and Hash Table, and a comprehensive demonstration of the performance advantage of BOSC over conventional disk I/O systems without compromising data integrity, using both synthetic and realistic workloads,

4. For *CDP*, we developed a modified track-based logging technique that can simultaneously achieve low write latency, high write throughput and high disk space utilization efficiency using only commodity IDE/ATA drives,

5. For *CDP*, we modified two-phase commit protocol that exploits low-latency disk logging to hide the latency of local mirroring and remote replication without compromising the data integrity,

6. For *Solid State Disk*, we developed a novel combination of logging and BUSC to successfully convert all random writes into sequential writes,

7. For *Solid State Disk*, we crafted a fully operational prototype that demonstrates significant write latency reduction on a commodity *solid state disk* under realistic disk access workloads, and

8. For deduplication atop of Change Block Tracking, we explored the spatial and temporal locality of changed blocks in incremental backups by employing a *s*egment-based variable-frequency sampled finerprint index and *s*egment-based containers.

1.6 Outline

The rest of the technical report is organized as follows. In chapter 2, we survey the related work of efficient metadata update techniques, continuous data protection,

optimization for SSD, and deduplication techniques. Chapter 3 presents the design, implementation and evaluation of *BOSC* for stand-alone index structures. Chapter 4 describes the design, implementation and evaluation of track-based logging, modified two-phase commit, TRM, and metadata updating using BOSC, respectively. Chapter 5 detail the design, implementation and evaluation of *Log-structured Flash Storage Manager (LFSM)* for *solid state disks* by combining data logging and metadata updating with BOSC. In chapter 6, we present the design and evaluation of a deduplication scheme atop the change block tracking with BOSC to speed up the metadata update. Chapter 7 concludes the technical report and discusses the future work.

Chapter 2

Related Work

2.1 Related Work of Efficient Metadata Update

BOSC makes an innovation by combining 4 techniques from various research community related to metadata update. These four techniques are as below: (1) Buffering of individual queries, (2) Low-latency logging of queries to preserve data locality, (3) Sequential I/Os to maximize the I/O efficiency, (4) Decoupling of *insert/update/delete* queries from the actual commit. These 4 ideas are not new by its own, but the combination of these 4 techniques provides surprising performance improvement for random *insert/update/delete* queries.

In the following 6 subsection, I will discuss related work with technique (1), (2) and (3) among other related work about efficient metadata update. Because technique (4) focuses more on the data update rather than metadata update and is detailed in section 2.2, I will omit related work in this section. More concretely, subsection 2.1.1 discusses the general-purpose write buffering techniques in details. Subsection 2.1.2 surveys techniques in index structures and algorithms to leverage the sequential bandwidth as in technique (3) and intelligent record buffering and layout techniques to improve the performance of *insert/update/delete* queries. Subsection 2.1.3 surveys the principles and variations for efficient logging. Subsection 2.1.4 discusses several data structures and applications in leveraging the logging techniques. Subsection 2.1.5 surveys techniques used in commercial DBMSs to showcase the potential benefits of applying BOSC in state-of-art DBMSs. In subsection 2.1.6, I survey techniques that employ domain-specific knowledge to speedup the query performance, which is orthogonal to what BOSC can do.

2.1.1 Write Buffering

Dedicated write caching [33–35] hides the write latency by turning synchronous writes to disks into asynchronous ones. Writes are *destaged* from the cache to the disks to

achieve 2 goals, 1) exploring the spatial locality by ordering destaged writes to disks, and 2) distributing write load evenly to minimize interference with concurrent read requests. These write caching schemes improve the caching efficiency by examining 3 design dimensions, 1) leveraging the temporal locality by coalescing overwrites to reduce the number of writes to disks, 2) maintaining reasonable free space in the cache to accommodate bursts and to avoid synchronous writes, and 3) fulfilling read requests from the write cache if there is a hit. In particular, the requirement 2) is critical because too much free space in the cache hurts both the temporal and spatial locality, while too little free space in the cache hurts the very first purpose of write caching, that is, latency hiding.

The CLOCK [35] algorithm leverages the temporal locality by approximating the least recent write (LRW) policy, and is widely used in file systems [36]. However, the spatial locality is not accounted for in such algorithms.

Theodore explored [37] the combination of the spatial locality with the temporal locality in write caching for storage systems. The proposed algorithm balances the spatial locality and the temporal locality by combining the LRW policy [38–40] and the LST policy [37, 41]. However, these algorithm only work on a single disk and does not optimize for 1) the interference of read and write requests, and 2) maintaining the write cache in a smart way.

The Wise Ordering for Writes (WOW) algorithm decouples the destaging order from the techniques to explore temporal and spatial locality by grouping writes and ordering these write groups based on the LBA order. A recent bit is associated with each write group to indicate the temporal locality. Writes of any pages in the write group set the recent bit. Before destaging a write group, the recent bit is checked, if the recent bit is 1, the write group is skipped after the recent bit is cleared.

The Spatially and Temporally Optimized Writes (STOW) [33] schemes aims to maintain reasonable free space of the cache by controlling the destaging rate. Although the WOW scheme solves the problem of destaging order well, it does not examine the destaging rate policy to maintain reasonable free space in the cache. In STOW, writes are categorized into a random queue and a sequential queue with the write group as the basic queue unit. The sequential queue and the random queue are examined alternatively to destage write groups, and a hysteresis period is maintained to avoid frequent alternating between the sequential queue and the random queue. A modified Linear threshold [41] method is employed to adjust the destage rate. Writes in proximity of the concurrent reads are opportunistically scheduled.

The BOSC scheme differentiates itself from these write caching algorithms in two ways. First, BOSC employs efficient disk logging to ensure data durability. In contrast, these write caching algorithms either assume the existence of the relatively expensive NVRAM backed up by batteries [33], or do not consider the durability at all [42]. Second, BOSC buffers records instead of pages, while these algorithms take the page (e.g., 4 KB or 8 KB) as the basic buffer unit. It is not surprising because the write caching scheme has no knowledge of the application-level record.

However, these write caching scheme does provide helpful insights in improving

the BOSC in two ways. First, for workloads of read/write mix, BOSC can improve the commit efficiency by opportunistically scheduling the commit in a proximity of the read requests. Second and more important, the linear threshold scheme can be applied to BOSC to maintain reasonable free space in the cache, and therefore to improve the overall update efficiency.

2.1.2 Reorganization of Index Structures

In this subsection, I discuss the reorganization of index structures to improve the random insertion/update performance in the research community. For the sake of clarity, we formally define 5 types of index queries as below. First, an *insert* query inserts a record without querying if there is previously a record with the same key, which is usually used in streaming applications such as social networking. Second, an *update* query first queries the index to figure out if there exists a record with the same key. If so, the record is updated with the incoming record. If not, the *update* query inserts the record to the index. Third, a *delete* query first queries the index to find out if a record with the same key exists in the index. If so, the record is deleted. If not, nothing happens. Fourth, a *search* query queries the index for a particular key. If a record with the same key exists in the index, the record is returned. Otherwise, no record is returned. Fifth, a *range search* query asks the index for all records within the input range $\langle low-key, high-key \rangle$, where $low-key$ and $high-key$ is the lower bound and upper bound of the key, respectively. Also, in the following discussion, N represents the index size, and B as the block size in memory transfer.

Although employed widely for database indexes in industry, B^+ tree and its variants suffer from 3 performance problems. First, naive B^+ tree has poor performance for random *insert/update/delete* queries. For these three query types of *insert/update/delete*, the index is first queried to figure out the exact location of the record, the record is then inserted, updated and deleted accordingly. Second, parameter tuning is critical to achieve good performance. These parameters include but not limited to the size of blocks, the size of available main memory, and the policy to reorganizing the index. As noted in [43], parameter tuning becomes more and more complicated and mandates strong expertise to make it work. Third, traditional B^+ tree ages [43] over time and the *range search* query suffers due to poor locality of logically neighboring records. For example, for a traditional B^+ index, when a leaf block A is split, a new block B is grabbed from the free block pool, and block A and B can be remote from each other, causing the well-know fragmentation problem [44].

The line of research in cache-oblivious (CO) data algorithms [43, 45, 46] solve the second and third problem associated with traditional B^+ by leveraging two techniques. The first technique, the van Emde Boas (vEB) layout [47], is a building block to adapt a structure to be cache oblivious. In concrete, the vEB memory layout improves the caching efficiency by recursively clustering sub-trees of CO data structures. For example, for a B^+ tree containing N nodes with the height of h, the sub-tree rooted at the root node with a height of h/2 is put together and all other sqrt(N)

sub-trees are ordered in a sequential fashion. Recursively, each sub-tree is partitioned into smaller sub-trees. If a sub-tree S can fit into a block B, the whole sub-tree S can be accessed after the block is read into memory. In contrast, the traditional B^+ tree ordered the index tree in the breadth-first order. For example, each key in internal nodes of B^+ tree in put in the same block with its sibling keys, not its children. As a result, when a particular key is fetched in by *memory transfer*, siblings of the key are fetched in but not necessarily its children. Its children are very likely to be retrieved through another *memory transfer*, leading to excessive *memory transfer*.

CO data structures avoid the aging problem from the design by leveraging the *packed-memory array (PMA)* technique. More concretely, PMA is a *ordered* data item array with free space to accommodate updates, insertions and deletions. Each time the free space is not enough or too much, PMA is extended or shrunk by moving all data items from old PMA to the new PMA sequentially to amortize the per-item update cost. Because PMA is an ordered structure, the locality of logically neighboring leaf blocks is preserved regardless how PMA or the index evolves over time.

Recent advance in the cache-oblivious research [48] improves the performance of random *insert* queries without sacrificing the performance of *search* queries to mitigate the first problem of traditional B^+ tree. Incoming random inserted records are queued in a buffer associated with an internal node which is the root of the subtree containing the target leaves of the inserted records. When the buffer is full, the whole buffer are *shuttled* down to the subtrees recursively until no buffer is full.

The CO stream B^+ tree [48] leverages two ideas to achieve the high insertion throughput. The first idea is from the buffer repository tree [49],and the other idea attributes to recursive CO subtree structures [50]. A **BRT** tree is a strongly-balanced tree with each internal node associating with a buffer. Each insertion record is first queued in the buffer of the internal nodes if the buffer is not full. If the buffer is full, the records in the buffer are *shuttled* down to the lower subtrees, which can be recursive until the buffer of subtrees can hold the records. The intuition of the **BRT** tree is that the internal nodes higher in the tree have a looser division of records and can easily accumulate records for the subtree rooted at the particular internal node. The amortized cost of *shuttling* records down to lower subtrees in **BRT** can therefore be greatly reduced. The amortized insertion cost of **BRT** tree is $O(logN/B)$, where N is the size of the index and B is the size of the block. However, the worst-case cost of a *search* query is $O(logN)$ because the buffer associated with each internal node is flat without leveraging the vEB memory layout.

The CO stream B^+ tree improves the amortized cost of a *search* query from $O(logN)$ to $O(log_B N)$ by converting the flat buffer associated with each internal node to a self-similar fractal subtree. Because a set of subtrees replace the flat buffer, a *search* query involves recursive search into a *selected* branches in these subtrees. Because search of branches in subtrees can benefit from the vEB layout, the cost of a search query can be greatly reduced.

However, the proposed technique proposed in the CO stream B^+ tree does not

optimize for random *update* and *delete* queries because these two queries involve an implicit *search* query, which has an amortized cost of $O(log_B N)$. The *search* query exhibits poor performance when the search key is random. Although there are techniques called **lazy-BRT** [51] to delay the *search* query, the worst-case cost is not bounded. As a matter of fact, in the extreme case, the search can involve the scanning of the whole index. In contrast, BOSC provides a general framework to optimize for all random *update* and *delete* operations by introducing asynchrony, where the *update* queries and *delete* queries are queued and logged, and eventually processed by a background I/O thread.

BOSC can benefit from the research advance in the CO data structures in two aspects. First, BOSC can leverage PMA to ensure the locality of logically neighboring blocks. For example, for a B^+ tree, instead of choosing a new block from the free block pool each time a split occurs, the PMA can be employed to rebalance the whole leaf blocks in an amortized fashion. Second, BOSC can be used in tandem with CO structures for random *update* and *delete* queries. For example, BOSC can be employed to improve the random *update* and *delete* queries. But how to integrate the per-internal-node buffer in CO stream B^+ tree with the per-block queue in BOSC needs more work.

Two generalized algorithms have been proposed to improve the performance of random *insert* queries in attacking the first problem of traditional B^+ tree. First, the logarithmic method is employed to adapt the static data structures to their dynamic counterparts to achieve amortized asymptotic optimal performance for **insertions**. Given N elements in the data structure and disk block size B, the logarithmic method partitions the data structures into $log_B(N)$ subsets D_i of exponentially increasing size S_i ($S_i < B^i + 1$), i = 0,..., $log_B(N/2)$. Search queries are issued to all partitioned subsets and the result is a join of results from the subsets, and the search query incurs asymptotically $\omega((log_B(N))^2)$ disk I/Os. Roughly speaking, insertion queries are issued to the subset D_i which can hold it. When D_i is full, all subsets D_j(j = 0, ..., i) are moved to the subset $D_(i+1)$ in a batch. Because the insertion is usually done to a memory-resident partition and the moving cost of records is amortized among all inserted records, the logarithmic method have the asymptotically optimal I/O cost for insertions. However, the search has to go through each individual partitions and the I/O cost for search is not optimal.

Buffer trees [48, 49, 52] are the second category of generalized techniques to improve the performance of random *insert* queries. The basic idea is to buffer insert operations with certain nodes of the index structure. The buffered insertion operations are pushed down to the next deeper level once the buffer overflows in the current level. One important characteristics of the buffer tree technique is that the intrinsic properties of the underlying index structures are preserved. For example, for balanced B-tree with buffer tree technique [49], after associating with each internal node a fixed-sized buffer, the B-tree is still balanced. Between insertion and search queries, the buffer tree technique favors the insertion queries over the search queries as the search query has to go through every buffer of nodes along the root-to-leaf

path. Not surprisingly, the cost of a *search* query is $O(logN)$

Brodal and Fagerberg [53] proposed a generalized B-tree, called B^{ϵ} tree, to formally study the trade-off between the search and insertion of the B^{+} tree and its variants. For $0 \leq \epsilon \leq 1$, the amortized insertion I/O cost is $\Omega(\frac{log_{B^{\epsilon}+1}N}{B^{1-\epsilon}})$, while the amortized search I/O cost is $\Omega(log_{B^{\epsilon}+1}N)$. If $\epsilon = 1$, it matches the traditional B-tree, and if $\epsilon = 0$, it matches a buffered repository tree [49].

BOSC heads for a different direction, in which the performance of *search* is not sacrificed while the performance of *update/insert* is increased significantly by deferring the commit of update/insert until the corresponding target on-disk node is fetched into memory, where the fetching of target on-disk nodes are sequentially to leverage the benefit of sequential disk block transfer. One constrain applies, though, that is, the internal nodes are assumed to be memory-resident and the only I/O cost is the I/O cost related with those leaf nodes. From a practical perspective, the assumption holds because the size of internal nodes are 2-3 magnitudes smaller than the leaf nodes [54–56]. But regarding the absolute performance of insertions, the CO data structures can easily beat BOSC because they are not sensitive to the block size and caching effect, including the L2 cache size and the internal memory size.

2.1.3 Smart Data Placement and Management

In modern computer systems, hard disks are abstracted as a flat one-dimensional array. Although the abstraction simplifies the interaction between the disks and the applications (i.e., OS) atop it because only a logical block address (LBN) is used in read/write, the latency of a write can vary with the internal states of the disks (i.e., the disk head position, track density,etc.). In this section, I will discuss three categories of existing techniques to improve the write performance for disks.

Disk scheduling [57] is the first category of optimization techniques to improve the disk-based write performance. Given a queue of input requests, the disk scheduling algorithms minimize the total amount of write latency for all writes by scheduling those writes based on their positions. The write latency is comprised of seek time, rotational time, settle time and data transfer time, where the settle time and data transfer time are independent of the disk state, or in particularly the disk head position. To minimize the sum of seek time and rotational time for a bunch of requests, the information of disk head position needs to be accurate. As the disk has full knowledge of the disk state, including the disk geometry and disk head position, disk scheduling within disk itself such as *tagged command queuing* [58] is relatively straightforward. In contrast, disk scheduling out of disks is much more challenging because modern disks do not expose their disk geometry information and the disk head information to end users. As a result, the effective disk scheduling algorithms either run on top of a simulator [59, 60], or were twitted with extreme care [61–64]. Others [42] argue that disk scheduling with only LBA information can achieve reasonable good performance. Another branch of research [6, 65] includes useful background requests into the input queue to get those background requests done for free. However, the effectiveness of

disk scheduling heavily relies on the length of the input request queue. If there is only one request in the queue, there is no benefit of using disk scheduling algorithms.

Instead of minimizing write latency by using disk scheduling algorithms for a group of write requests, write indirection [14, 66, 67] aims to minimize the seek time and rotational latency for a single write request. Write payload is written to the "nearest" free space indicated by the disk head position and disk geometry. To manage the location information associated with each write request, some advocates to map the write to the location [68, 69], others embrace the idea of using advanced disk interface [70, 71]. The former involves maintaining a non-trivial amount of remapping metadata, and the latter needs significant changes to applications interacting with the disks. The mapping metadata can either be maintained in the file system [72, 73], or can be maintained at the block-level. The former fits more natural with the functionality of file systems: querying the location metadata and subsequently accessing the data. However, many legacy file systems (i.e., ext3) [74] do not support such arbitrary mapping. For these file systems, block-level remapping [68, 75] is a remedy.

Block-level remapping from logical block number to physical block number is not a panacea yet for three reasons. Firstly, the query of remapping metadata at payload read time and the updates of remapping metadata can incur significant performance overhead if not done properly. BOSC can solve the problem with unified logging ability by logging updates and metadata updates simultaneously. Secondly, garbage collection [76] is inevitable to defragment the free space as the disk ages, which turns out to incur non-trivial overhead [77] and implementation complexity. Thirdly, the applications above the block layer (i.e., file systems, database index) have the full knowledge of the written payload and it is their responsibility to manipulate the placement of data payload. The block layer does not have full knowledge to make a meaningful placement decision. For example, those blocks containing metadata accessed together should be placed in proximity. A line of research [30, 78] proposed different ways to feed the block layer with the application-level knowledge.

Although not a perfect solution under all scenarios, the block-level remapping fits nicely into one niche area, the data logging (i.e., WAL [79], undo/redo logging for RDBMS [80]). To resolve the performance overhead associated with metadata query and metadata update, the BOSC scheme can be used as below. Firstly, the logical-to-physical mapping itself can be logged together with the data payload to mitigate the overhead due to mapping update. Secondly, the metadata update is committed to the disks using sequential disk I/O in BOSC. Thirdly, after the logical-to-physical mapping record is committed, the corresponding log record can be safely discarded. Eventually, when the data payload is committed to the disk, the payload logging can also be safely discarded. Therefore, we get rid of the aging problem associated the block-level remapping scheme. A complete design for a case study is presented in section 4.

2.1.4 Logging and Batching of Individual Records

Logging techniques [81–83] are widely used for index structures to improve their update efficiency. The sequentiality nature of logging provides two-fold benefits, (1) efficient write performance, and (2) even utilization of all disk blocks. However, the benefits are not gratuitous, a remapping data structure is necessary to remember the physical location of a write, and disk blocks need to be recycled to be reused as the system ages [72, 73]. One more complication arises to apply logging techniques to the index structures, that is, the update of the index structures is significantly smaller than the on-disk block size. The in-page logging technique [82] mitigates the problem for NAND flash disks by appending individual updates to the end of an erasure unit containing the target page of the updates. In concrete, each EU is partitioned into two areas, the first area holds the page, and the second area holds the logging records. The page is reconstructed on the fly every time a read is issued to the page by merging the logging updates with the page. Although the in-page logging technique improves the update efficiency of index structures, each read involves a read of the data block and a read of all logging blocks because logging records of that page may scatter in the logging blocks. What is more, each time the logging blocks are full, every data page in the EU has to be moved to a new EU even if the data page itself does have related logging records.

Instead of logging records on the NAND flash disk, the NAND-based B-tree employs in-memory logging to improve the update efficiency of a traditional B-tree. The B-tree internal nodes are assumed to be memory-resident. Each leaf node of the B-tree is associated with a link list of sectors containing the logging updates of the leaf node, which allows flexibility in choosing which sector to log the updates. Similar to the in-page logging technique, the logging updates are merged with the leaf node each time the leaf node is accessed. To improve space utilization, the logging updates are buffered and lazily written to disk. This "lazy" update technique is also adopted in [83].

However, none of these techniques provide a full-fledged solution to the metadata update problem. For example, the durability of the remapping data structures is not considered in [82], the durability of the index structure is not guaranteed in [81, 83]. In contrast, the BOSC technique provides the durability guarantee through high-performance metadata logging for all index structures employing BOSC.

2.1.5 Insert/Update Optimizations of State-of-Art DBMSs

In practice, due to the complexity of the database internals, the database vendors are reluctant to adopt completely-new data structures and new index organization to achieve high performance. Instead, incremental improvement is more likely to succeed and in this section I summarize several techniques to improve performance for existing index structures in commercial database systems, several emerging database companies using new index structures, and future trends for commercial database

systems.

Graefe proposed several techniques [55, 84, 85] to improve the insertion and updating performance of B^+ trees with a focus on the ease of deployment. In [55], Graefe proposed a novel modification to improve the de-fragmentation and reorganization performance of B^+ tree. A logical pointer called *fence* instead of a physical pointer to sibling B^+ tree leaf nodes was proposed to limit the performance overhead of migrating B^+ tree leaf nodes. However, this scheme optimizes for inserts and does not work well for update-in-place operations because the latter need to fetch target leaf nodes first before modifying them. In [84], Graefe surveyed techniques to improve updating performance and quantitatively projected that an order of magnitude slow-down for the query performance can be traded for an improvement of inserting performance by two orders of magnitude. In [85], he proposed to add an artificial leading column to logically partition a single B^+ tree to several small B^+ trees. Similarly to the *buffer tree technique* [86], incoming updates 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. Moreover, such optimization techniques work well for streaming insertions as for collecting sensor data [86] but does not work well for update-in-place workloads such as OLTP workload. In contrast, BOSC is equally effective for insert-only and update-in-place workloads. Moreover, all these techniques focus only on B^+ tree and it is not clear if these techniques can be applied easily to other index data structures (i.e., R-tree), whereas BOSC is effective across multiple distinct database index structures.

The InnoDB storage engine in MySQL [87] adopts a technique called insert buffer tree to defer the insert/update until the corresponding B-tree leaf pages are fetched into cache. Insert/update requests are inserted to the memory-resident insert buffer with the key of (Index_ID, Key). The buffered insertion/update requests are committed to the corresponding B-tree leaf pages by a background thread. The background thread adopts a greedy policy to commit records. That is, each time it chooses the leaf page with the maximal number of queuing records as the target leaf page to commit queuing requests to. Our experiences show that the greedy policy does not consider the sequentiality of the disk access, and therefore is outperformed by the policy which sequentially fetches leaf pages. Although the insert buffer scheme does not achieve the optimal possible performance, it can achieve insert/update throughput up to 15 times larger than that does not use insert buffer. The improvement of the insert buffer scheme indicates that the BOSC idea can be applied to the commercial database with a significant performance boost.

TimesTen [88], Datablitz [89] and SolidDB [90] are examples of main-memory database products on the market. Although these products are optimized for in-memory access instead of block-based disk access, each database index still has an on-disk counterpart to preserve the durability of the database index. Recovery logs (i.e., the undo log and the redo log) are used to speed up the performance. For these

products, the BOSC scheme can be employed to improve the performance furthermore by scheduling the disk access more efficiently.

Tokutek Inc. commercializes the CO stream tree [48] as mentioned above by providing a storage engine called TokuDB [91] to MySQL DBMS. Because TokuDB allows efficient *insert* and *search* queries, the cost of update can be reduced for the following reason. Each update consists of (1) lookup of an index and (2) update to many other indexes. Traditional database DBMSs do not allow many **clustered** indexes due to slow *insert* queries. In contrast, TokuDB allow many **clustered** indexes because of its efficient insertion, which can greatly reduce the cost of lookup and therefore the overall cost of the *update* query.

Vertica Inc. [92] founded by Stonebraker et al. provides the column-based database solution optimizing for the warehousing data workload, where the query throughput is uttermost important. One important optimization technique for such workload is to use column-based tables instead of row-based tables to avoid the wasteful caching of unused columns in the queries [93]. This category of optimization is orthagonal to BOSC, and BOSC can work together with these optimization techniques to improve the queuing efficiency.

Stonbebrake et al. [94, 95] also proposed a more radical change to the traditional RDBMSs. As traditional disk-based RDBMSs entails significant performance overhead due to disk access (record access, undo logging, redo logging,etc), locking (two-phase commit locking), latching (concurrent access of latched data structures shared by multiple threads), and buffer management (access memory from the buffer pool), Stonebraker et al. proposed H-Store, a memory-resident RDBMS, to remove most of overhead in traditional disk-based RDBMSs. H-Store operates on a cluster of machine nodes, and these nodes collaborate to service the database queries. H-Store ensures data durability by replicating tables across different machine nodes, and does not persistently commit database changes to disks. Because memory access is faster than the disk access, H-Store employs the single-threaded model to complete each transaction one by one, and removes the overhead of locking and latching associated with multi-threaded model completely.

The assumption of operating a RDBMS in memory without disk backbone as in H-Store is suspicious for two reasons. Firstly, not all database workload can fit into main memory. For example, H-Store demonstrates that a collective 100 GB memory is enough to hold the OLTP workload. But other database workload, i.e., data mining workload of warehousing, can easily reach hundreds of terabytes. Secondly, even if the database workload can fit into memory, the inevitable software and hardware faults (bugs, hacks, crash) can easily corrupt the in-memory data structures. One way to mitigate the corruption problems is to increase the degree of replica. However, increasing the degree of replica will reduce the effective amount of memory holding database indexes, which may invalidate the assumption that the whole database workload can fit into main memory. In contrast, as long as disks are used, the BOSC scheme can benefit the database workloads without maintaining replicas in memory.

2.1.6 Insert and Update Optimizations Exploring Domain Specific Knowledge

It is well-established that the general-purpose RDBMSs do not perform well for many applications [94, 96, 97]. These applications include text processing (Google's Bigtable, Yahoo's PNUTS), warehousing (OLAP and data mining), scientific computation (i.e., geometry data processing), etc. Instead of using RDBMSs, specialized algorithms employing domain-specific knowledge were proposed for these applications to achieve reasonable performance. These specialized algorithms outperforms their counterpart using general-purpose RDBMSs by a factor of several magnitudes because the specialized algorithms usually abandon a subset of strict RDBMS properties to fit the need of their applications. In this section, we explore important common techniques employed in these domains. As the optimization in each domain is usually case-by-case, we skip some of the implementation details.

The Yuntis [98] search engine efficiently updates a number of metadata when processing a crawled page by queuing metadata update operations with their target disk blocks. Metadata disk blocks are brought into memory from disk in the decreasing order of the number of their pending updates, and are kept in memory as long as possible for maximum reuse. However, in-depth knowledge of the internals of Yuntis's data structures is necessary to convert it from a control-centric approach to a data-centric approach. In general, it takes a nontrivial amount of effort to reorganize an arbitrary application to benefit from the technique used in Yuntis. In contrast, BOSC is designed to be a general disk access mechanism that can be used by a wide range of applications.

Search engine giants [99, 100] develop text processing systems to suit the requirement of their applications. Key/value pair is the basic unit to access a record in Google's Bigtable [99]. Bigtable improves the I/O efficiency by logging every updates to a redo log. The most recent updates are kept memory-resident, while older updates are written out as update files. Periodically, both the memory-resident updates and updates in the files are merged with the original file accommodating old records. Reads are sacrificed to benefit writes because each read has to merge all related update files and memory-resident updates to form a consistent view to the applications. Although Bigtable employs the Bloom Filter [101] technique to reduce the number of disk access for frequently accessed key/value pairs, for the majority of the records, applications suffer in getting these records. Fortunately, many applications using Bigtable is not latency-sensitive application. But it is not clear whether Bigtable can fit the requirement of latency-sensitive applications.

Yahoo's PNUTS [100] takes a different perspective in crafting their design: consistently low latency for all requests. PNUTS relies heavily on a pub/sub system called Yahoo! Message Broker (YMB) to ensure the record durability and reliability. To achieve low-latency update, an update request is regarded as "committed" after the request is published to YMB, after which the application initiating the request can proceed. YMB publishes the update request by logging the request to disks on multiple servers. Eventually these committed records are propagated to their target

storage units by YMB. The way the insert/update request is serviced is similar to that in BOSC, the only difference is that BOSC now only works on local storage, not in a cluster environment. As presented in PNUTS, in a cluster environment, the record update is applied to the master copy of the record, while the record query can be serviced from different replicas, even the local stale replica. However, the latency to read the latest version of the record instead of the stale local copy is not well pronounced, which presumably involves merging of YMB's log records and the storage unit holding the record replica. Again, for most of their applications, there is no compelling demand to read the latest version of a record as long as YMB can propagate the updates to all replicas within a moderate time limit. However, the absence of the demand to read the latest version may be not a safe assumption for other applications [99, 102].

Netapp develops a utility called Spyglass [103] to manage metadata of the large-scale storage systems. Spyglass uses K-D trees [104] to map multiple attributes to files these attributes pertain to in a peta-scale distributed storage system consisting of commodity hardware (i.e., Ethernet, SATA disks, etc). Because file attributes have strong temporal and spatial locality [105], Spyglass partitions the global metadata index into hierarchic sub-partitions based on the namespace hierarchy. The partitioning provides two-fold benefits, namely, 1) record updates are confined within a single partition without triggering massive record movement as in the global metadata scheme (i.e., a row-based RDBMS table), 2) record search can be confined in local partitions with the help of a bloom filter signature. In concrete, each partition creates a bloom filter signature file for each attribute, which encodes the existence of files owning the attribute in the particular partition. A query fails to "hit" the bloom filter in the middle of the namespace hierarchy does not bother to traverse down, and those partitions in the subtree can be safely pruned from the search.

Spyglass is a big success in showing its effectiveness in Netapp-like environment, where updates are triggered periodically (i.e., hourly) by the background metadata crawler. However, for other workloads of different characteristics, Spyglass's methods may not play well for two reasons. Firstly, the workload does not exhibit so strong locality as that in Spyglass. Secondly, even if the workload exhibits strong locality, it is not easy to model the locality in a hierarchic way. For example. for block-level deduplication, the block-level does not have knowledge of the files owning these blocks, and therefore can not leverage the data locality using a hierarchic tree, which hurts the pruning efficiency of metadata search.

The optimization for data warehousing [93, 106, 107] workload has a different focus. Instead of optimizing for the updates, these optimizations stress more on the read queries. For example, the MonetDB [107] reduces the performance overhead of ad-hoc queries to large dataset by employing various cache-aware techniques. Because the disk I/Os are all sequential, the performance overhead roots from in-efficient use of main memory and cache. All optimization techniques center around the efficient use of main memory and CPU cache. One important optimization technique is to use column-based tables instead of row-based tables to avoid the wasteful caching

of unused columns in the queries [93]. As BOSC was proposed to speed up the insert/update performance, these techniques are orthogonal to the BOSC scheme and can be used in combination if the importance of the read performance is uttermost.

Scientific computing [97, 108–113] leverages computing-specific knowledge to organize and manipulate the index structures to intelligently accomplish the computing task. For example, the Weaver system [97] cultivated in CMU takes the raw data as input and generates unstructured octal meshes with billions of elements. The processing takes four phases, (1) construct phase to build unbalanced linear octree, (2) balance phase to balance the unbalanced linear octree, (3) extract phase to extract leaf node and internal nodes of the output octal mesh, (4) transform phase to glue leaf nodes an internals and to output a result file. The first impressive technique used in phase 2) is called balance by part (BBP), which partitions the whole linear octree into equal-sized 3D volume parts. Each part can fit into the memory and these volume parts are written out sequentially to take advantage of the sequential disk I/O. The second technique called *two-level bucket sort* appears in phase 3) to remove duplicate internal nodes. The *two-level bucket sort* technique maps each internal nodes to a per-region bucket, sorts nodes of each per-region bucket to eliminate duplicates, and eventually appends the result to the output internal node file. The insightful observation is that node extraction is an offline task instead of an online one, that is, the final position of each node is fixed given the input linear octree. The first optimization technique shares with the BOSC scheme to explore sequential disk bandwidth. The second optimization technique shares with BOSC to construct a local data structure to fit into main memory and yet to be written out sequentially.

2.2 Related Work of Asynchrony

To resolve the tension between data consistency and update performance, the idea of external data synchrony [114] has been proposed to approximate the performance of asynchronous disk I/O while providing the same consistency guarantee as seen by an external observer as synchronous disk I/O. Data updates are kept in a buffer and flushed to disk only when external outputs are generated. BOSC uses a special low-latency logging technique to reduce the performance cost of synchronous logging. As a result, BOSC is simpler because it does not need to be coordinated with other system events that produce externally visible outputs.

2.3 Related Work of CDP

Continuous Data Protection (CDP) backups the data on-the-fly as it is written to the disk. Therefore, every update is undoable. Traditional data protection system relies on file system backup, which is performed in a much coarser granularity than CDP. Using a CDP based solution in network storage could result in an increase in network and individual server resource load since all the data written to master storage node

have to be backed up to the local mirror storage nodes at the same time.

Parallax [115] also adopts block-level *Copy On Write(COW)* semantics to minimize data copies between different *Virtual Disk Image(VDI)*s in large-scale distributed environments. It focuses on the management of a large number of Virtual Machine(VM)s and does not consider preserving data over a long term. It employs a radix tree to provide ready access to one historical image and a group of radix trees are organized in a separate data structure. In contrast, Mariner uses an External B-Tree to preserve the full mapping of the logical address plus a timestamp to the physical address.

The idea of the disk head prediction used in Trail is not new. One example is the *Free block scheduling* [6, 116]. The objective of Free block scheduling is to piggy-back background media transfer with normal workload activity with little-to-no overhead by utilizing the rotational and seek latency of the requests belonging to the normal workload. A freeblock scheduler predicts the amount of the rotational latency before the next foreground media transfer and inserts background media transfer within the anticipated latency to minimize the impact on the foreground media transfer. The key point for the feasibility of free block scheduling is that the ordering of requests for background disk activities is not mandatory for background activities.

In the original Trail architecture [12], there is a normal disk, which holds the user data, and a log disk, which provides a fast staging buffer for disk writes. Given a disk write request, Trail first writes its payload to the log disk, and then completes the write to the normal disk asynchronously. Because Trail ensures that the disk head is always on an empty track and could accurately estimate the disk heads position in real time, it can write a piece of data to the log disk where the disk head happens to be at that instant. Consequently, each write operation incurs very little rotational latency and zero seek delay.

Fiber Channel (FC) is the predominant storage area networking technology. It evolved as an alternative data transfer technology to the low performance 10 Mbps Ethernet technology. With the advent of Gigabit Ethernet, the initial concerns of bandwidth and latency requirements of SANs no longer remain a core issue. The ultimate enabler of Ethernet-based SAN is the iSCSI [117] protocol, which defines semantics for block level SCSI I/O over any IP network. iSCSI is the FCP counterpart on Ethernet networks that maps the SCSI command set to the TCP/IP stack. The increase in the momentum of iSCSI is evident from its availability through several major operating system vendors and the availability of iSCSI enabled HBAs from major hardware vendors. Although it is debatable whether iSCSI/Ethernet will completely replace Fiber Channel SAN or not, the increasing momentum of Ethernet based SANs is undeniable.

The memory to memory approach to avoid data copy [118] [119] [120] has been widely applied. Remote Direct Memory Access(RDMA) is a zero-copy networking technique, permitting data to be transferred directly from application memory of one machine to that of another machine without involvement of host CPU processing, caches and context switches of host Operating Systems. These features are especially

important in highly parallel networking systems such as clustered computing [121]. Although technically superior to other alternatives, RDMA is not widely advocated in network storage systems because its most common underlying infrastructure is InfiniBand [122], a point-to-point switch fabric interconnect technology not widely used. In contrast, iSCSI operates seamlessly on Ethernet networks, which is the most widespread LAN technology in use till now.

Payload caching [123] and Network-centric buffer caching [124] share similarity with RDMA in the sense that they all aim to minimize the data copying. Payload caching caches payload in Network Interface Card (NIC) and reduces data traffic through host I/O bus. Network-centric buffer caching keeps a network friendly format in page/buffer cache to avoid data content copying and transformation overhead. As *TRM* aims to minimize traffic load on Ethernet network, these techniques are orthogonal with *TRM*. Compared with these techniques, *TRM* is much more favorable for applications that requires data replications.

Cisco SANTap [125] is a protocol that sits between the MDS switch and a storage application appliance. The SANTap service registers as both an initiator (host) and a target device (storage array) in the Fiber Channel name server. It allows the storage appliance to get a copy of the I/O exchange between the server and storage without compromising the primary I/O. The appliance is no longer in the data path. In addition, SANTap also needs to provide error recovery services to permit recovery in the event of appliance or port failure. However, special hardware support is required to use the SANTap approach. Our *TRM* approach achieves similar functionality on Ethernet switches and leverages TCP to make sure of reliable data replication.

The Viking project [126] revisits architecture features of Ethernet technology when Ethernet is applied to network storage and large scale Metropolitan Area Network(MAN). Viking overcomes one efficiency weakness of Ethernet technology by extending a single spanning tree to multiple spanning trees in forwarding routed data packets by leveraging on standard Virtual LAN technology. Viking improves the efficiency of underlying Ethernet-based network by making a better use of underlying data link capacities and reducing the down time of link failures . In contrast, *TRM* targets at minimizing network traffic passing through NICs and therefore gains better network efficiency.

Link-layer multicast could be implemented by exploiting IGMP snooping [127]. Traditionally, Ethernet switches treated link-layer multicast packets as broadcast packets. Network performance suffers as unnecessary packets are forwarded through the network. Fortunately, most modern Ethernet switches, particularly Gigabit switches, support a feature called IGMP snooping, which was designed to support IP multicast without using link-layer broadcasting. *TRM* makes a novel use of IGMP snooping to implement the link-layer multicast.

2.4 Related Work of File Versioning

Versioning file systems maintain versions of files at the file system level. File systems in Plan-9 [128], AFS [129], and WAFL [130] checkpoint the whole file systems periodically to support file versioning. For these systems, the temporal resolution of file versions is the checkpoint interval. Some file systems such as Elephant [131] and Versionfs [132] supports finer-grained versioning granularity and more flexible data retention policy. Elephant is a kernel-level versioning file system that creates a new version only when a file is closed. VersionFS is a versioning file system based on a stackable file system architecture [133]. As in Elephant, a new file version is created only on file close but the versioning policy is flexible. VersionFS also provides a friendly interface for users to access old versions and to customize the versioning policies. VersionFS still incurs non-negligible performance overhead - about 100% when measured by the Postmark benchmark. Neither Elephant nor VersionFS can distinguish between file updates that occur between a file open and file close operation.

Wayback [134] is a user-level comprehensive versioning file system that creates a new file version upon every file update. Ext3cow [135] is a comprehensive versioning file system that does not require any modification to ext3's interface of the kernel. It implements a $time-shifting$ interface in ext3 file system and employs a copy-on-write scheme to avoid polluting the file system cache. In ext3cow, only versions that are propagated to disk are retained. Block-level CDP systems take the same approach because they creates a new block version only when an updated block reaches the storage server. However, in ext3cow, individual file versions are attached to snapshots and file versions between consecutive snapshots are invisible to end users.

CVFS [136] is a kernel-level comprehensive versioning file system that is optimized for metadata logging efficiency. Journal-based meta-data is used for Inode/Indirect-block update and multiversion B-tree [137] is used for directory update.

2.5 Related Work of Consistency Check and Ensurance

Val Henson [138] proposed a file system-wide dirty bit to indicate whether the file system is being actively modified when the system crashes. When a file system crashes with the dirty bit not set, FSCK knows that it does not have to do a full FSCK when the system reboots. In addition, they proposed a technique called linked writes to identify a list of dirty Inodes to limit the scope of FSCK to checking only those dirty Inodes. Linked writes can also be viewed as a form of journaling where the journal entries are scattered across the disk and linked by the on disk dirty Inode list. It needs to link the Inode to the on disk dirty Inode list before the actual operation on this Inode takes place. Similarly, *iFSCK* scans updated blocks within a time window to limit the scope of FSCK. The difference is that *iFSCK* leverages file system-specific

knowledge to identify block updates that modify file system metadata and therefore works without any file system modification.

Other file systems, for example the ext3 file system under Linux, use a journaling architecture to give a transaction semantics to file system metadata updates through redo logging; it is relatively straightforward to identify disk updates that modify file system metadata because each of them appears in a separate log. In addition, the ext3 file system readily tags disk updates associated with their corresponding file system update operations. Therefore, its crash recovery code does not need to parse the disk updates to correlate them with their associated file system update operations.

CIMStore [139] exposes the history of all data written to stable storage for search or browsing. CIMStore marks the state of the system as being "quiescent" to speed up the recovery of the state of a storage application (e.g. a file system) and provides consistent point-in-time information. These "quiescent" points are located either by understanding high level data structures such as the "unmounted cleanly" flag of the superblock of a file system, or explicit notification from a client-side application monitoring storage application activity. This "quiescent" point detection is currently done manually in CIMStore. In contrast, *iFSCK* does not need "quiescent" points and transparently restore the storage volume to a consistent state.

J. Kent Peacock et al. [140] developed a fast consistency checking technique for Solaris file system. The fast FSCK provided with Netra NFS keeps track of the active portions of the file system and limits the scope of the file-system check to only that "working set." They need to modify the file system structure to add the state information and perform logging of certain transactions such as directory update. In contrast, *iFSCK* combines the write log from the CDP node and the file system-specific knowledge to infer the working set without modifying the file system itself.

2.6 Related Work of Write Optimization of SSD

2.6.1 Flash Translation Layer (FTL)

does not scale well, page-level mapping Two types of flash disk technologies exist [141, 142]. NOR flash disk is designed to replace erasable ROM (EPROM) and provides random data access in the same way as EPROM. The minimal addressable unit for NOR flash disk is a byte, the same as EPROM. In contrast, NAND flash disk is designed for mass storage. NAND provides a block access interface, and the minimal addressable unit is the same as a commodity hard drive, usually a 512-byte sector. A page in a NAND flash disk is usually a 2-KB page and can accommodate 4 sectors.

FTL is designed to mitigate the performance problems associated with the hardware limitations of flash disks [143]. First, overwriting an already written flash memory cell is physically not possible. Instead, FTL reads in the erasure unit containing the write's target sector, modifies it, erases the original erasure unit, and writes the modified version to the newly erased erasure unit. This read-erasure-write cycle incurs significant performance overhead because an erasure operation usually takes an

order of magnitude longer than a write operation. To reduce this performance overhead, FTL could write the in-memory copy to a separate erased erasure unit before erasing the original erasure unit. Second, because there is a limitation on the number of erasure operations that can be applied to an erasure unit [144], FTL employs wear-leveling techniques to distribute the writes evenly to all the erasure units in a flash disk so as to maximize its usable lifetime.

Page-based FTL maps a logical page to an appropriate physical page of a NAND flash disk [143, 145]. Andrew Birrell et al. [143] proposed a page-based mapping technique in which a logical page is remapped to the tail of the current erasure unit. A page is reserved at the end of each erasure unit to hold the mapping information and other metadata. The mapping from a logical page to its corresponding physical page is stored in volatile memory and every time the flash disk is restarted, the whole disk has to be scanned to reconstruct this mapping. This design incurs a long start-up delay when the flash disk is large. In contrast, LFSM maintains the address mapping information separately from the data log, and only those blocks whose corresponding mapping updates are still pending at the time of crash need to be scanned at the system recovery time. Therefore, LFSM is more scalable with respect to the increasing size of the flash disk. Moreover, commodity flash disks' FTL cannot afford page-based mapping because of the scarcity of on-board memory. To use 4-KB pages as the address mapping granularity on a 64-GB flash disk, it will require 16*36/8=72 MB, which is too much for most commodity flash disks.

Instead, unit-based FTL [146] maps a logical unit address to a physical unit address. Each logical page address thus consists of the address of the logical unit containing it and its offset within the unit. With unit-based FTL, repeated updates to a single sector in a unit may result in an erasure of the entire unit. The hybrid mapping scheme [147, 148] allows FTL to remap the offset of a page to a different offset within the same unit, and thus entails a smaller mapping memory requirement than page-based FTL. Units are categorized into log units and data units. Log units act as a write buffer to accommodate new writes while data units are designed to hold data with regular unit-based mapping. Log units only account for a small percentage of a flash disk's units.

eNVy [149] is a large non-volatile storage system built with Flash memory. Flash memory in eNVy has the same access interface as that of SDRAM, e.g., flash memory is accessed through the memory bus rather than the I/O channel such as IDE or SATA. To mitigate the performance problem associated with random writes, the target logical page of each write is remapped to a new physical page and the mapping information is updated accordingly. A battery-backed SRAM is employed to hold the mapping metadata. When the space in eNVy is used up, a cleaning process similar to Sprite's LFS [79] is triggered to recycle unused pages. In contrast, LFSM works with commodity flash disks that support block access interface and keeps the metadata on the same flash disk as the data, and is thus applicable to all PCs equipped with commodity flash disks, without requiring any expensive battery-backed SRAM.

CloudBurst [150] is a log-structure virtual disk subsystem designed specifically

around flash disks. Written virtual disk blocks are remapped to the end of a log. Data payload is fully compressed and a segment containing the metadata header and the compressed data payload is appended to the end of data log. In comparison, LFSM does not compress all the data payload, instead, but only a part of each write's payload to squeeze out space for the metadata header. CloudBurst's mapping structure is volatile in memory and is reconstructed from the flash disk at system startup time through scanning of the whole data log. In contrast, LFSM keeps a persistent mapping structure and only those blocks whose associated mapping updates are pending at the time of system crash need to be reconstructed at system startup time.

2.7 Related Work of Data De-duplication

Venti [151] pioneers the content-addressable storage (CAS) by computing the fingerprint (i.e., SHA1 hash value) of a data block and using the computed fingerprint instead of a logical block number to address the data block. Data blocks are de-duplicated because data blocks with the same content have the same fingerprint. However, Venti does not focus on de-duplicating data blocks, and more efforts are spent ensuring write-once-read-many property. In concrete, Venti does not address two performance problems associated with fingerprint-based de-duplication. Firstly, the access locality is lost because adjacent data blocks have very different fingerprint values. Secondly, for a large-scale storage system, the fingerprint index can not fit into main memory and the lookup of the fingerprint index during data de-duplication can incur extra disk I/Os, which can incur a significant performance overhead [27, 28].

Based-on whether the de-duplication steps into the critical I/O path, de-duplication techniques can be categorized into two camps, the inline de-duplication technique and the out-of-line de-duplication. In inline de-duplication [27, 28], each incoming write is checked for de-duplication purpose before it arrives at the disk. If previously there is write payload with the same content, the incoming write does not need to be written to disk. Otherwise, the new write payload is written to the disk. In contrast, out-of-line de-duplication techniques do not de-duplicate write payload on the fly. Instead, each data payload is first written to disk. A background procedure checks the newly written data payload for the de-duplication purpose.

Kai Li et al. [27] proposed an online de-duplication technique for a disk-to-disk (D2D) [22] data backup system. Because the fingerprint index can not fit into memory and resided on the disk, the paper proposed two techniques to minimize the overhead due to I/O access of the fingerprint index. Namely, the two techniques are (1) a bloom filter-based [152] summary vector to avoid unnecessary fingerprint lookup, and (2) a locality-preserving data placement scheme to leverage the spatial locality of the input fingerprints.

As the first optimization technique, the auxiliary bloom filter covers all fingerprint values of data blocks and accounts for a non-negligible amount of memory usage. On one side, a miss in the bloom filter indicates there is no such fingerprint value in the index structure. On the other hand, a hit in the bloom filter is not decisive in

determining if the fingerprint value of interest exists in the fingerprint index, and a search to the index structure is inevitable.

The second optimization technique preserves the locality by loading/evicting the fingerprints based on a container rather than a continuous range of fingerprint values because neighboring fingerprints do not reflect the data locality. The container corresponds to a continuous range of logical blocks from the input backup stream, and contains all metadata related to the continuous range, including the fingerprints and physical locations of these blocks. A query miss of one fingerprint in the container triggers the loading of all fingerprints in the container, predicting other fingerprints in the container will be queried in subsequent fingerpint queries. In most cases, the prediction is correct due to data locality in the input backup stream.

The sparse indexing scheme [28] uses a sampling fingerprint index instead of a whole fingerprint index to further reduce the memory usage of de-duplication in an online de-duplication system for a D2D data backup system. The insight of the proposed scheme is that duplicated data blocks tend to be in a consecutive range with a non-trivial length. A match of sampling fingerprint values in the range indicates the matching of the whole range with a high probability. Among all matched ranges, a champion is chosen to de-duplicate against. The sampling ratio can be used to trade-off the de-duplication quality and the memory usage. In one extreme, if all fingerprint values in the range are sampled, the is most efficient. In the other extreme, if only one fingerprint value in the range is sampled, the de-duplication algorithm can err in choosing the champion range and therefore the de-duplication efficiency drops. Although the segment match based on sampled fingerprint works well for examined workload, it is not clear how effective it is for other backup workloads, including changed blocks within a file or an email, which our de-duplication techniques focus on.

Many live storage systems [153–155] opt for the out-of-line de-duplication techniques because the performance overhead associated with online de-duplication is not acceptable for these systems. In these systems, the de-duplication functionality is cut out of the critical data write path, and the de-duplication is scheduled in the background. In particular, the data de-duplication improves the storage efficiency for distributed file systems [153–156] because all hosts in a cluster tend to host similar files or operating systems. For data de-duplication in distributed systems, the performance of metadata lookup and maintenance is not a serious concern because metadata can be distributed across all participating hosts in a distributed fashion.

HYDRAstor [157] is a content-addressable distributed near-line storage system with fault tolerance in the design. Backup images are chunked into segments, each segment is routed to a *peer* machine based on its fingerprint. Because each peer machine has the full hash key information for all segments stored on it, the peer machine can de-duplicate the segment in an online fashion. Different from Venti, segment is stored continuously on commodity storage devices to preserve data locality so that both read and writing of segments are I/O-efficient. HYDRAstor employs the mark-and-sweep garbage collection technique to reclaim physical blocks. Because

typical storage capacity of peer machines in HYDRAstor is in the scale of $\tilde{1}0$ TB, the counters of all physical blocks can still fit into memory and the marking process can be very fast. As the per-peer machine storage capacity grows, the mark-and-sweep garbage collection is not scalable because the counters of all physical blocks can not fit into memory anymore and the I/O becomes the performance bottleneck.

However, HYDRAstor showcases complicated practical consideration in designing a robust garbage collection and de-duplication scheme against failures if the failure of machines is a norm in a distributed environment. For example, during the garbage collection, the marking of deleted blocks needs to survive machine failures to prevent future writes to take the to-be-deleted blocks as a stored duplicate. Their solution for this problem is to make the system read-only first, marking all to-be-deleted blocks in one shot, and then make the system read-and-write. For our deduplication system, we can learn how they deal with each corner cases in fighting against failures.

The Foundation [158] leverages commodity USB external hard drives to archive digital files in a similar fashion to Venti. Different from Venti but similar to the Data Domain scheme, a 16 MB segment is stored continuously to preserve locality for sequential read and fingerprint caching. For each fresh write, up to 3 disk accesses are encountered: (1) the lookup of fingerprint, (2) appending the data payload to the end of the data log, and (3)fingerprint updates to the on-disk fingerprint index. Bloom filter is employed to filter out unnecessary lookup in (1). For (3), in updating the fingerprint store, similar to BOSC, a buffer is employed to accommodate fingerprint updates and a single sequential scan is used to commit updates when the buffer is full, which shows the effectiveness of BOSC in de-duplication. However, the fingerprint updates are not logged to the disk and their durability is not ensured, which can cause problem when the fingerprint update buffer is not filled for a long time.

Online de-duplication techniques in backup storage systems (i.e., D2D backup systems) and out-of-line de-duplication techniques in live storage systems are different in 4 aspects. Firstly, online de-duplication can save disk space in the first place because data de-duplication is done on the fly. In contrast, out-of-line de-duplication first stores duplicates and de-duplicate data in the background. Secondly, online de-duplication incurs performance overhead on the critical data path, while out-of-line de-duplication chooses to offload the performance overhead in the background. Thirdly, online de-duplication in backup storage systems is conducted periodically, while out-of-line de-duplication techniques need to deal with duplicates produced by dynamic changes. Fourthly, out-of-line de-duplication techniques employs file as the basic unit of data de-duplication, while online de-duplication techniques in the backup storage systems use block streams as the basic unit of data de-duplication.

Although different in many aspects, out-of-line and online de-duplicatio fit well in their corresponding arenas. For example, because the primary concern of a live storage system is to provide low-latency access to live data and temporary space utilization is not the top concern, deferring the data strikes a reasonable balance between performance overhead and space utilization. Conversely, backup systems focus more on backup throughput than the latency of individual backup requests,

and the storage utilization is the first priority. Therefore, online de-duplication is preferable for backup systems.

Our proposed de-duplication technique differs from that of Data domain paper and that of sparse indexing paper in three aspects. Firstly, the input of the backup stream consists of only changed blocks since last backup. Secondly, the spatial locality is captured based on the file. Thirdly, the sampling rate of fingerprint selection is varied based on the de-duplication history to capture the temporal locality of input backup stream. The first difference requires our technique to squeeze out the duplicates by fully exploring their spatial locality and temporal locality.

In [159], de-duplication is based on files. Each file has a representative fingerprint. Entries in the fingerprint index are distributed to K nodes one by one based on modular operation, or other distributed hash table functions. The whole container corresponding to a fingerprint index entry is distributed to the same node as the fingerprint index entry. If two fingerprint index entries happen to have the same container but distributed to two different nodes, the same containers are duplicated on two nodes. When a file is backed up, only the representative fingerprint is used to route the file to a node, all other fingerprints of the file are not used for routing purpose. In contrast, in our proposed technique, all fingerprints of a segment are used to query the fingerprint index on the corresponding node. More importantly, containers are also distributed to all K nodes using the hash of fingerprints, eliminating the duplicate of containers.

In [159], each file has a whole-file fingerprint, which means that a match of the whole-file fingerprint indicates that the whole file is a duplicate. In our proposed de-duplication technique, each segment has a whole-segment fingerprint, which is more flexible. However, to save RAM space, it is desirable to have the file-level information, including the file identifier and file offsets, to represent the segment instead of individual physical block addresses.

Chapter 3

Batching mOdification and Sequential Commit (BOSC)

3.1 Update-Aware Disk Access Interface

The conventional disk access interface supports `read(target_block_addr, dest_buf_addr)` and `write(target_block_addr, src_buf_addr)`, for applications to read and write disk blocks, respectively. Under this interface, existing disk I/O systems optimize the physical disk access efficiency by delaying and/or scheduling disk write requests, but by servicing read requests as soon as possible in order to decrease the critical path delay. In addition, after a disk read request is serviced, control is passed back to the process issuing the request and the additional processing on the requested block is done in the context of the issuing process.

Logically, an update to a disk block involves a disk read of the target disk block and a disk write of the same block after the block is brought into memory. If a disk I/O system could treat each disk update request as an atomic operation, 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 disk I/O system must be able to perform the request's associated update operation in a way that is independent of the semantics of the application issuing the request. To allow an application of a disk I/O system to explicitly declare a disk access request as a disk update request and supply the necessary information for the disk I/O system to service it atomically, we propose an *update-aware* disk access primitive specifically 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 information related to the requested modification, and a pointer to an application-specific function that commits the actual modification to disk. This primitive is sufficiently general to accommodate

common disk update requests from such storage applications as a database index manager, 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 disk I/O 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. In fact, the disk I/O system does not even need to differentiate among *create*, *update*, or *delete* operations. As a result, the update-aware disk access interface gives a disk I/O system the same flexibility of scheduling disk update requests as disk write requests, and enables significant improvement in physical disk access efficiency. Finally, the proposed disk update primitive seamlessly complements the standard disk read and write primitives, which are still needed for supporting legacy applications.

3.2 Batching Modifications with Sequential Commit

Figure 3.1 shows how BOSC implements the update-aware disk access interface. It applies a space-efficient low-latency disk logging technique to log each disk update request, queues it in an in-memory queue, commits updates to disk asynchronously using mostly sequential disk I/O, and recovers from failures efficiently. The following subsections describe its design in more detail.

3.2.1 Low-Latency Synchronous Logging

Logging an update request to disk synchronously and then performing whatever operations triggered by the update asynchronously is a well-known technique. BOSC takes the same approach to deliver high sustained disk update throughput and the same durability guarantee as synchronous disk updates. In BOSC, each log record for a disk update request contains the following information:

- A copy of the data structure pointed to by `ptr_modification`,
- A global sequence number for the current disk update request,
- A *back pointer* to the disk location of the log record that is temporally immediate before this record,

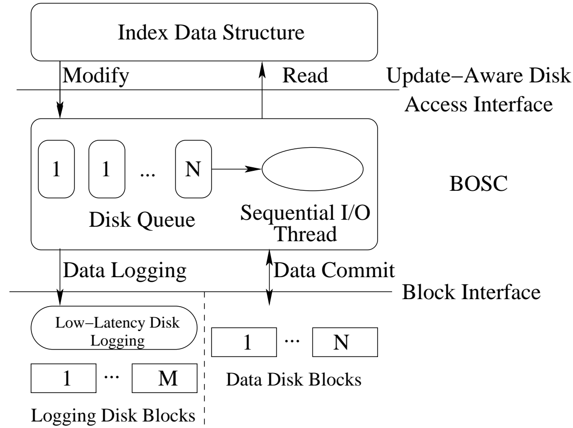


Figure 3.1: *BOSC associates with each disk block an in-memory update request queue. When BOSC receives a disk update request, it logs the request to disk, queues the request in the update request queue associated with its target disk block, and then performs the update operation only when the target disk block is brought into memory. BOSC fetches target disk blocks using sequential disk I/O.*

- A *global frontier*, which corresponds to the global sequence number for the *youngest* disk update request before which all disk update requests have been committed to disk, and
- A *local frontier*, which corresponds to the global sequence number for the *youngest* disk update request before which all disk update requests to the target (*local*) disk block of the current disk update request have been committed to disk.

Upon receiving a disk update request, BOSC increments the current global sequence number and assigns the result to the request, and prepares its log record by extracting the data structure pointed to by `ptr_modification` and copying the *global frontier*, the *local frontier* associated with the specified target disk block, and the disk location of the last log record. To illustrate how BOSC maintains the system-wide global frontier and the per-block local frontiers, let's consider the following update request sequence: 1(10), 2(15), 3(10), 4(2000), 5(30), 6(10), 7(15), where the numbers outside the parentheses are global sequence numbers of disk update requests and those inside are their target disk block numbers. Suppose a system failure occurs immediately after Request 7 is logged to disk, and at that point only the effects of Requests 1, 2, 3, 5 and 6 are committed to disk. So at that instant, the global frontier is 4, the local frontiers for Blocks 10, 15, 30 and 2000 are 6, 2, 5, and 0, respectively. BOSC uses the global frontier to determine which log records can be recycled at run time and to reduce the number of log records that need to be examined at recovery time. The per-block local frontiers can further cut down the recovery processing load,

as is explained in Section 3.2.3.

Because the end-to-end throughput of BOSC is bounded by its synchronous disk logging performance, BOSC extends a low-latency disk logging technique [12] to be both low-latency and space-efficient, to support aggressive disk request batching, and to work on a commodity disk array [160]. The key idea in this low-latency logging technique is to write a disk block to where the disk head happens to be. More concretely, BOSC maintains a separate disk request queue for each disk in the log disk array. At any point in time, one of the log disks serves as the *active* disk. In the beginning, BOSC randomly chooses one of the log disks as the active disk. Once a log disk becomes the active disk, it remains as the active disk until the waiting time of the oldest pending request in its queue exceeds a threshold, T_{wait}. Whenever a new disk write request arrives, BOSC inserts the request to the active disk's queue as long as the waiting time of its oldest pending request is smaller than T_{wait} and there is enough free space in the current track to accommodate the new request; otherwise BOSC dispatches the request batch currently in the active disk's queue, chooses another log disk as the active disk and inserts the new request to its queue.

To select a new active disk for an incoming write request, BOSC computes the earliest time at which the write request could be written to each log disk, and selects the one that can write the request to disk at the earliest. When computing a write request's write time on a log disk, BOSC takes into account the current position of the log disk's head and the possibility of batching the new request with others already in the disk's queue. For those log disks that are currently idle, BOSC only needs to consider the delay due to batching. A key design decision in BOSC is to dispatch a new write request to a log disk that allows batching of as many disk write requests into one physical disk write operation as possible, rather than to one with the earliest write time for that request. However, BOSC uses T_{wait} to limit the size of batching and to ensure that the experienced latency of each incoming disk write request is always bounded.

Because of batching, multiple log records could be merged into a physical disk request when they are written to disk. Also, the actual disk location of each log record is only known at the last moment, i.e., right when they are written to disk, and BOSC keeps track of this information accurately to chain log records together through their back pointers.

To track the log disks' disk head position, BOSC statically extracts the physical disk geometry information from every log disk, and constantly keeps track of each log disk's disk head position at run time. More concretely, after a physical disk write is completed, BOSC records the *LBA (Logical Block Address)* of its last sector, LBA_0, and its completion timestamp T_0. Assuming the disk head stays in the same track, when the next write arrives at T_1, BOSC estimates the disk head's current position $CurrentLBA$ using the following formula:

$$CurrentLBA = SPT \cdot \frac{(T_1 - T_0)\ mod\ RoTime}{RoTime} + LBA_0 \tag{3.1}$$

where SPT is the number of sectors in the current track, $RoTime$ is the disk's full rotation time. The final predicted position, $DestinationLBA$, is $CurrentLBA +$

Lookahead, where *Lookahead* is an empirical value chosen to account for such delays as the controller delay and avoid a full rotation delay due to tracking errors. Detailed design and analysis of low-latency synchronous logging can be found in chapter 4.

3.2.2 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 queues, one for each 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 the log disk array; 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 1 msec.

In the background a separate thread of BOSC constantly reads the data disks sequentially to fetch into memory those disk blocks whose pending request queue is non-empty. This thread goes from the beginning to the end of the data disks, and repeats the cycle. This is referred to as the *sequential commit cycle*. Every time the background BOSC thread brings in a disk block, it applies all the pending updates to the disk block and writes the block back to the 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, BOSC physically reads and writes disk block runs, and commits pending updates on a run by run basis.

A disk block run corresponds to a contiguous sequence of disk blocks the percentage of which having a non-empty pending request queue is above a threshold (currently set to 0.5) and whose length is no greater than another threshold (currently set to 32). The first and last disk block in a run must have a non-empty pending request queue, and runs are disjoint. As one run is being fetched from a data disk, the background thread applies pending updates to another run that has previously been brought into memory, and pushes a third run, whose pending updates have already been applied, to another data disk. By pipelining the processing of runs, BOSC is able to eliminate unnecessary disk seek delays and reduce the number of disk rotations required to commit pending updates to a sequence of disk blocks to 2.

Although the general idea of batching disk I/O requests to amortize the associated disk access overhead is well known, there are several differences between BOSC and similar efforts in the past. First, BOSC exploits the flexibility afforded by the update-aware disk access interface to schedule disk update requests as if they are disk write requests. Second, BOSC incorporates a disk geometry-aware low-latency disk logging technique to deliver the same integrity as synchronous disk updates while reducing the application-perceived latency and the performance cost associated with synchronous logging to the minimum. Finally, BOSC uses *sequential* disk I/O to commit pending updates and greatly improves the overall disk update throughput.

3.2.3 Recovery Processing

After a system crash, BOSC parses the log to discover uncommitted disk update requests, reconstructs the in-memory per-disk-block update request queues that exist immediately before the crash, and resumes its normal processing. Note that BOSC chooses *not* to commit all uncommitted disk update requests to disk. 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. Searching the log for the log record with the largest global sequence number.
2. Determining the *replay window* in the log that contains log records related to the reconstruction of update request queues.
3. Parsing the log records in the replay window to reconstruct the per-block request queues.

To speed up Step (1), BOSC 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. A major problem in this binary search is how to identify log records on the disks. Each disk log record is self-describing and contains a separate disk sector for metadata. One of the fields in this metadata disk sector is a one-byte *signature* field, which is a unique bit pattern that allows BOSC's recovery subsystem to identify each log record on the disks. However, the same signature bit pattern may also appear in the user data. To solve this problem, BOSC uses `0xFF` as the signature bit pattern, copies the first byte of every user data sector in a write request to the metadata sector, and changes the first byte of every user data sector to `0x00` before logging them to disk. This design guarantees that the first byte of each sector on the log disks is either `0xFF` or `0x00`, and those starting with `0xFF` start a log record.

Finding the *beginning* of the replay window is trivial because it actually corresponds to the *global frontier* field of the youngest log record, because all updates associated with log records before the global frontier 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. Given a log record, BOSC first assigns the value in its *local frontier* field to its target block's *local frontier* if the request queue of its target block is empty; then BOSC inserts the update request in the log record into its target block's request queue in the global sequence number order if its global sequence number is larger than the *local frontier*. By leveraging the global and local frontier information in log records, BOSC can avoid inserting a significant portion of the log records in the replay window into per-block request queues. After BOSC reconstructs per-block update request queues, it resume normal-mode processing.

3.2.4 Extensions

A straightforward way for a BOSC application like a database index manager to query if a record of certain qualification exists in a disk block is to explicitly read in the block and scan it for the target record. 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 potential inefficiency, 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 scan the pending update requests in memory-resident queues and/or the target disk block if BOSC needs to fetch it into memory. This API allows BOSC to double the in-memory request queues as a cache for the associated disk blocks.

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 are 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 is contained in an application-level transaction that the application's recovery manager thinks should be aborted, the application's recovery manager will explicitly undo the update request with a compensating update request.

Currently, the log used by BOSC's recovery manager is not exposed to the applications using BOSC. That is, if a BOSC application such as a database index manager is built on BOSC, it needs to perform its own write-ahead logging in addition to BOSC's synchronous disk logging. This application-level logging step could be optimized away if BOSC provides a *unified logging* API [160] that allows high-level applications to specify their log records associated with high-level updates and register call-back functions to be invoked at recovery time. With these information, BOSC will be able to perform a single physical disk logging operation that effectively encompasses multiple logical logging operations from different software components.

3.3 BOSC-Based B^+ Tree

We have successfully ported three tree-based database index implementations (B^+ tree, R tree and K-D-B tree) from TPIE [161, 162] and an existing hash table implementation [163] 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 the face of very large data sets.

Common steps shared by porting efforts of these database index implementations

are

- Allocating and de-allocating disk blocks,
- Constructing a data structure that contains all the necessary information required to modify and to query a target disk block,
- Developing an update commit function that performs a requested modification, which could be a delete, an insert or an update operation, on a disk block that is brought into memory, and
- Developing a query function that scans the per-block request queues before retrieving target disk blocks when servicing a query request.

Of course, the actual data structure layout and internal logic for commit/scan functions are different for different index implementations. However, in general they can be easily adapted from their original implementations without significant changes.

We will focus on the B^+ implementation, and other index structures are similar to adapt to use BOSC.

To service a modification (write) command, a database index implementation first determines the disk block holding the target index page, then constructs an update request record and finally calls BOSC's update API with the target disk block's address, the associated update request record and its commit function as input arguments. To service a query (read) command, a database index implementation first determines the disk block holding the target index page, and then it constructs a query request record and calls BOSC's query API with the target disk block's address, the query request record and its query function as input arguments.

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, whereas 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 the BOSC-based 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 number of index records in each leaf node, and proactively triggers the split of a leaf or internal node when the number of records in a tree node exceeds a threshold. 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.

Take the case of B^+ tree for example. Its update commit function includes a component that examines the target disk block's pending update request queue to determine whether an update request will trigger a structural change or not, and if so, enacts the change by generating additional update requests if necessary, for instance, allocating a new block, modifying another block to point to the new block, copying some part of the current block to the new block, etc. Note that B^+ tree's commit function does not need to perform any disk I/O while enacting a structural change associated with a disk update request, not even fetching the disk update request's target disk block This is possible only if additional application-specific metadata about a disk block is stored with its update request queue, for example, the remaining free capacity of a disk block in the case of B^+ tree.

To support structural changes to a database index triggered by a modification command, the commit function of each update request comprises two components: the first component modifies the target disk block and is invoked when the target disk block is brought into memory, and the second component performs synchronous structural modification triggered by an update request and is invoked at the time when the request is queued. The second component ensures that all additional update requests generated by an update request X are reflected to their associated queues immediately after X is queued. For example, for an insert operation to a B^+ tree, its first component updates the index page into which the new record is inserted, and its second component is responsible for splitting an index page into two when the number of its pending update requests exceeds the capacity of the index page. For an insert operation to a hash table, the first component updates the page containing the target bucket, and the second component re-queues the request if the page that is supposed to hold the target bucket is already full.

3.4 Performance Evaluation

3.4.1 Evaluation Methodology

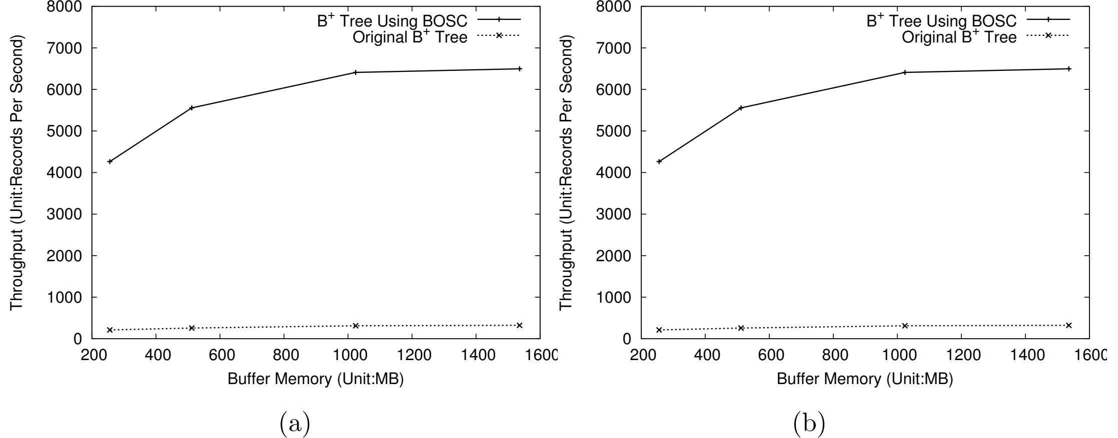


Figure 3.2: **(a):** *Comparison between the record insertion throughput of a BOSC-based B^+ tree implementation and a vanilla B^+ tree implementation based on the conventional disk read/write interface under the random insert workload when the total amount of buffer memory is varied from 256 MB to 1.5 GB. The leaf block size is 64 KB, the record size is 64 B, and the initial index size is 16 GB. The scan size is 64 MB.* **(b):** *The same as in* **(a)** *except the workload is the random update workload.*

We have built a complete BOSC prototype based on Fedora Core 3 with Linux 2.6.11 as the kernel. 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.

To evaluate the efficiency of the BOSC-based B^+ tree, we used the following four synthetic workloads: (1) *sequential insert* workload, (2) *random insert* workload, (3) *clustered insert* workload, and (4) *random update* workload. In the sequential insert workload, records with sequentially increasing key values between 0 and 2^{60} are inserted into an initialized index. 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 clustered insert workload consists of a group of fixed-sized (32 by default) clusters of record insertions. Within each cluster, records with sequentially increasing key values are inserted. However, the key of the start record for each cluster is randomly generated. 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 80-GB IBM Deskstar DTLA-307030

disks with on-disk cache disabled, four of which store the B^+ tree index records and one of which is dedicated to low-latency logging. We turned off on-disk caching to ensure durability of data written to disk.

In practice, 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 many different initialized B^+ trees in the entire performance 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 experiments, 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 properly initialized. During the experiment, 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 50 seconds.

3.4.2 Overall Performance Improvement

Figure 3.2 shows the throughputs of a vanilla B^+ tree implementation on a conventional disk read/write interface and a BOSC-based B^+ tree implementation under the random insert and random update workload. 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.

The performance of the BOSC-based B^+ tree implementation under the random update workload is almost the same as that under the random insert, because

in both workloads the accesses to the index pages are random and consequently their performance is bottlenecked by disk I/O. Figure 3.2(b) shows that the throughput improvement of the BOSC-based B^+ tree implementation over the vanilla B^+ tree implementation is the same as in Figure 3.2(a). 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 [84, 164, 165] 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 the 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 the 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 [166], where the number of warehouses is set to 20, 40, 60 and 80. Each trace entry includes the access type (e.g. read, update, delete and insert) and the key/data information in each request to the index engine. The average record sizes are 69 bytes, 102 bytes, 112 bytes and 128 bytes for warehouse 20, 40, 60 and 80, respectively. 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 3.3 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

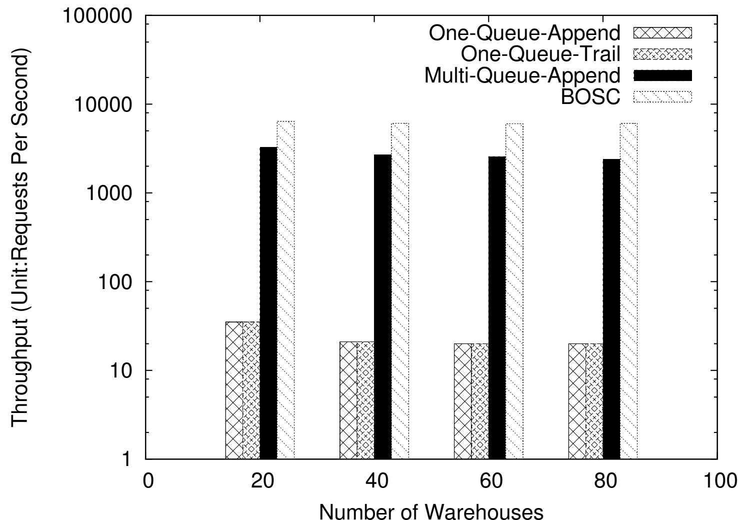


Figure 3.3: *Throughput comparison among the BOSC-based B^+ tree implementation, the B^+ tree implementation with multiple request queues and append-only logging, the B^+ tree implementation with one request queue and append-only logging, and the B^+ tree implementation with one request queue and low-latency 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. 3 data disks and 2 logging disks are used. The leaf node size is 64 KB and the buffer memory is 1 GB.*

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

Figure 3.3 also shows that the throughput of the BOSC-based B^+ tree implementation slightly decreases with the number of warehouse numbers, because larger warehouse number corresponds to larger database index size and thus tends to lower the average number of pending update requests per disk block when they are committed. Because the accesses in the TPC-C workload still exhibit some locality, the

performance impact of larger index size is less dramatic than expected. The vanilla B^+ tree implementation, on the other hand, is largely unaffected by the number of warehouses, because most index access requests require random disk I/Os anyway, regardless of the index size. Overall, under the TPC-C workload, the BOSC-based B^+ tree implementation is 300 times faster than that of the vanilla B^+ tree implementation when there are 80 warehouses, and is 180 times faster when there are 20 warehouses.

3.4.3 Sensitivity Study

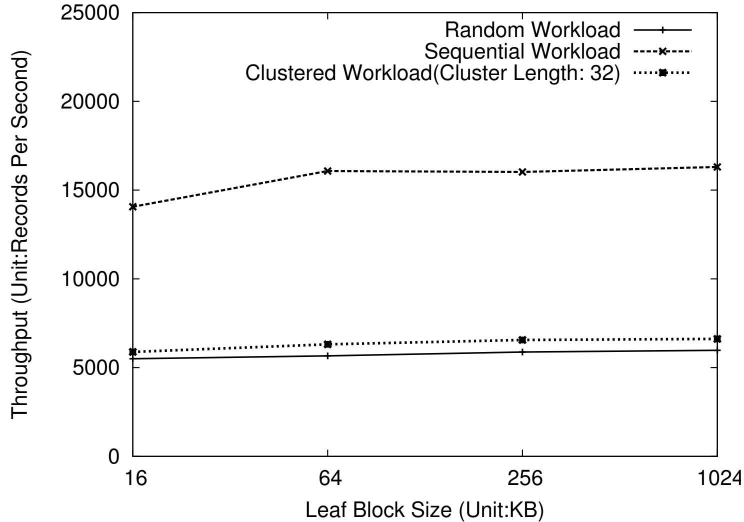


Figure 3.4: *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the leaf node size is varied from 16 KB to 1024 KB. The X axis is log-scale. The Y axis is the number of new records inserted per second. 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.*

The default setting for the B^+ tree experiments whose results are reported in this section is as follows. The key field of each B^+ tree index record is 16 bytes long, and the record size can vary. The total memory available for BOSC's per-block request queues is up to 1536MB. The B^+ tree's internal nodes are pinned down in physical memory and require about 200 MB for a 128-GB index with a 16-KB leaf block size and 64B record size. In the sequential insert workload, the key values of the newly inserted index records increase from 0 to 2^{60} sequentially. In the random insert workload, the key values of the newly inserted index records are uniformly distributed between 0 and 2^d, where d depends on the index size, the record size and also how the test B^+ tree index is initialized.

If the index size is A , the record size is B and the incremental step in initializing the B^+ tree index is C, then $d = log(C * 0.5 * A/B)$, where 0.5 is the fill factor of the initial B^+ tree index. For example, if the index is 128 GB, the record is 64 B and $C = 1000$, then d is 40.

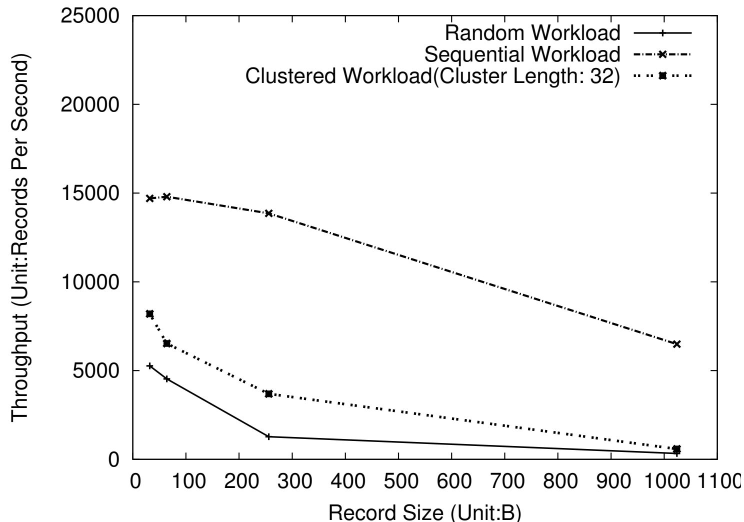


Figure 3.5: *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the index record size is varied from 64 B to 1024 B. The Y axis shows the number of new records inserted per second. The memory allocated for all per-block request queues is 1 GB, the leaf block size is 64 KB, and the initial index size is 128 GB.*

In the clustered insert workload, key values of the newly inserted index records within a cluster are increasing consecutively and each cluster starts with a random key value. The cluster size is fixed at 32 in all experiments if not specified otherwise. In the random update workload, the key values of the updated index records are randomly chosen from key values previously inserted by the random insert workload.

In this section, we evaluate the impact of the leaf node size, the index record size, the index size and the buffer memory size on the performance of the BOSC-based B^+ tree. Four data disks and one log disk are used. 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 I/O 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 3.4.

In general, the throughput performance of the BOSC-based B^+ tree index under the sequential insert workload is much higher than that under the random insert workload for two reasons. First, the average number of pending requests in each

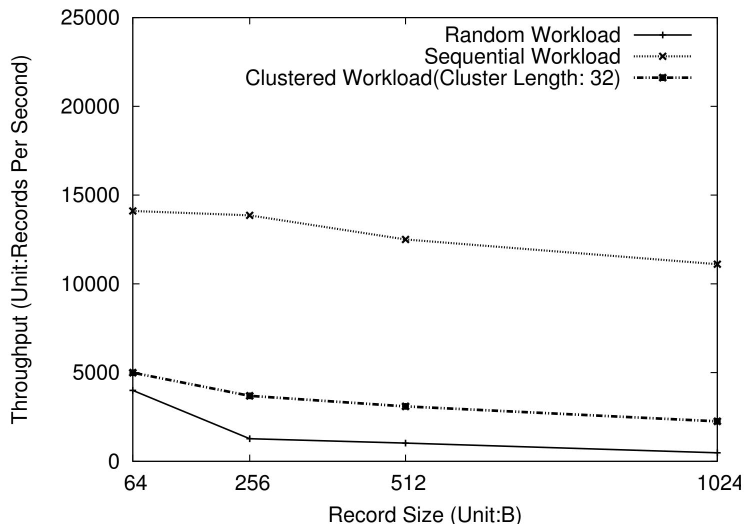


Figure 3.6: *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the index record size increases from 64 B to 1 KB, and the leaf node size also varies proportionally so that the ratio between the two is fixed. The Y axis shows the number of new records inserted per second. The memory allocated for all per-block request queues is 1 GB, and the index size is 128 GB.*

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 3.4. 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 3.5 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.

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 3.6.

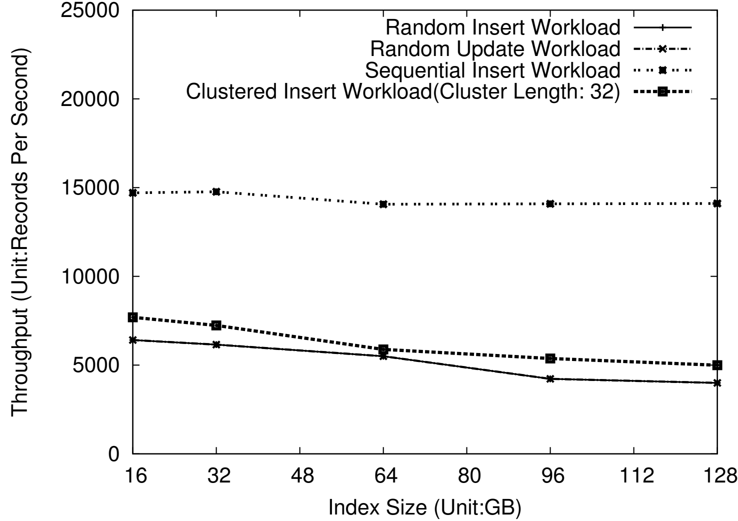


Figure 3.7: *Record insertion/update throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, random insertion and random update workload when the initial index size is varied from 16 GB to 128 GB. The Y axis shows the number of new records inserted/updated per second. The memory allocated for all per-block request queues is 1 GB, the record size is 64 B, and the leaf block size is 16 KB.*

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 3.7. 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 3.7 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 3.8. The performance impact of buffer memory size is minimal for the sequential insert workload because its degree of batching is largely unaffected by the buffer memory size.

For the random workload, there is a significant increase in throughput when the total queue memory is increased from 512MB to 1GB because IO dominates. But from 1G to 1.5G, the increase is not linear because the throughput is reaching its maximum. For the cluster workload, the increase is not linear because the average queue size does not increase linearly. For the sequential workload, the throughput does not change much because the memory is not the performance bottleneck.

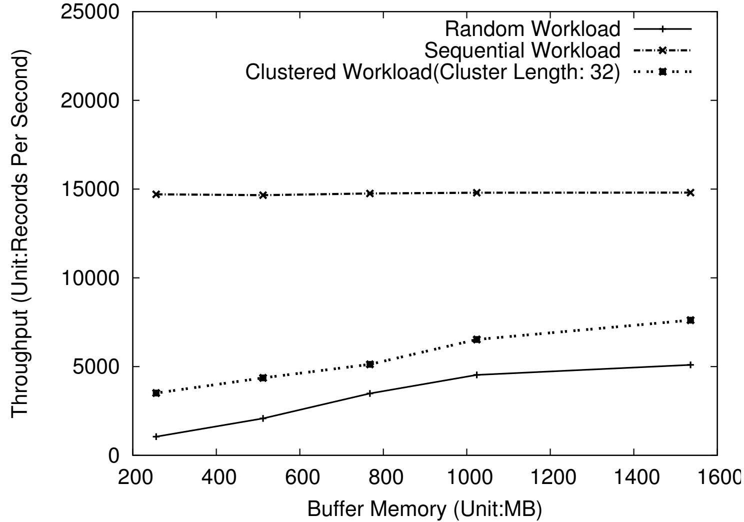


Figure 3.8: *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the BOSC's buffer memory is varied from 512 MB to 1536 MB. The Y axis shows the number of new records inserted per second. The record size is 64 B, the leaf block size is 64 KB, and the initial index size is 128 GB.*

3.4.4 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, including a 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 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 physical disk I/O size to 256 KB or 4 KB. Figure 3.9(a) 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 256-KB disk I/O unit 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. For the BOSC-based hash table implementation with 4 KB disk I/O, its throughput keeps increasing with the buffer memory size because larger memory size leads to

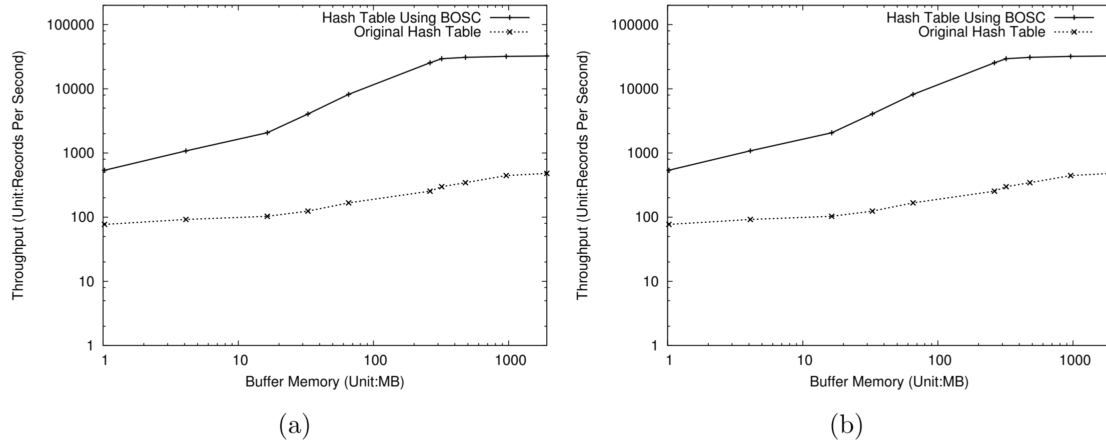


Figure 3.9: **(a):** *Comparison between the record insertion throughput of a BOSC-based Hash Table implementation and a vanilla Hash Table implementation based on the conventional disk read/write interface under the random query workload when the total amount of buffer memory is varied from 1 MB to 2 GB. Both X and Y axes are log-scale. The physical disk I/O sizes used in sequential commit are 256 KB and 4 KB.* **(b):** *The same as in* **(a)** *except the input workload consists of record updates rather than record inserts.*

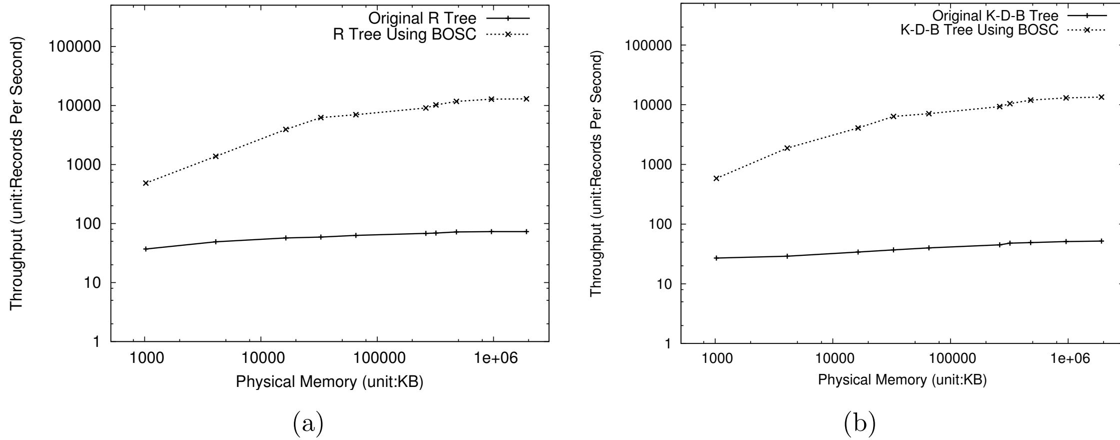
better batching efficiency for each physical disk I/O unit. When the buffer memory size is smaller than 64 MB, the average queue length of the BOSC-based hash table implementation is 1 and the performance gain of BOSC originates mainly from sequential disk I/O.

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 disk I/O unit is 256 KB and 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 3.9(b) 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 workload. 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.

3.4.5 R Tree and K-D-B Tree

The buffer memory is varied in size and is used to cache an R tree's leaf pages in the case of the vanilla R tree implementation and to hold per-block request queues in the case of the BOSC-based R tree implementation. The physical disk I/O size used in



(a) (b)

Figure 3.10: **(a):** *Comparison between the record insertion throughput of a BOSC-based R Tree implementation and that of a vanilla R Tree implementation based on the conventional disk read/write interface under the random 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 physical disk I/O size used in sequential commit is 32 KB.* **(b):** *The same as in* **(a)** *except the index structure is K-D-B Tree and the disk I/O size is 128 KB.*

sequential commit is 16 KB. The random insert workload, which inserts a series of squares into an initially empty R tree index, where the X and Y coordinates of their lower-left vertex and their size are all randomly distributed between 0 and 2^{61}, is used in this experiment. Each experiment run starts with an empty R tree and continues with new index record insertions until the R tree's size reaches 8 GB. The memory for holding all of the R tree's internal nodes is pinned down and is 16 MB in size.

Figure 3.10(a) shows the BOSC-based R tree implementation is more capable of exploiting the size increase in buffer memory than the vanilla R tree implementation, and when the buffer memory size is 960 MB, the throughput of the BOSC-based R tree implementation reaches around 12800 requests/second, which is more than 150 time higher than that of the vanilla R tree implementation (fewer than 80 records/second).

The set-up for the K-D-B tree experiment is the same as that for the R tree experiment, except the input workload inserts a series of two-dimensional points, where both their X and Y coordinate are randomly distributed between 0 and 2^{61}, and the physical disk I/O size used in sequential commit is 128 KB. Similar to R tree, Figure 3.10(b) shows that the BOSC-based K-D-B tree implementation also benefits more from the size increase in buffer memory than the vanilla K-D-B tree implementation, and when the buffer memory size is 960 MB, the throughput of the BOSC-based K-D-B tree implementation under the random insert workload reaches around 13000 requests/second, which is more than 250 times higher than the vanilla K-D-B tree implementation (fewer than 50 records/second).

3.4.6 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
R Tree	-	-	17.82	17.79
K-D-B Tree	10.52	10.50	15.32	15.34
Hash Table	10.78	10.79	-	-

Table 3.1: *The average latency of Point and Range queries for the B^+ tree implementations with and without BOSC, R tree, K-D-B tree and Hash Table. All four types of index structures are initiated by varied sequential insert workload as described in corresponding subsections. Two types of read queries are tested, Find and Range Query. The leaf block size for all index structures is 4 KB, and the buffer memory is 256 MB. The current R tree implementation supports only range queries, the current Hash Table implementation does not support range queries.*

Table 3.1 shows the average latency of Point and Range queries for the B^+ tree implementations with and without BOSC. The B^+ tree is initialized by a sequential insert workload, and then immediately used to service a set of 100 Point and Range queries. 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. 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. 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 [84, 164, 165].

3.4.7 Logging and Recovery Performance

BOSC relies on low-latency logging to provide the same durability guarantee as synchronous disk updates. The average latency of logging a 4-Kbyte block to an IDE disk array is under 0.5 msec, about an order of magnitude smaller than conventional disk logging implementations and the fastest ever reported in the literature. In addition,

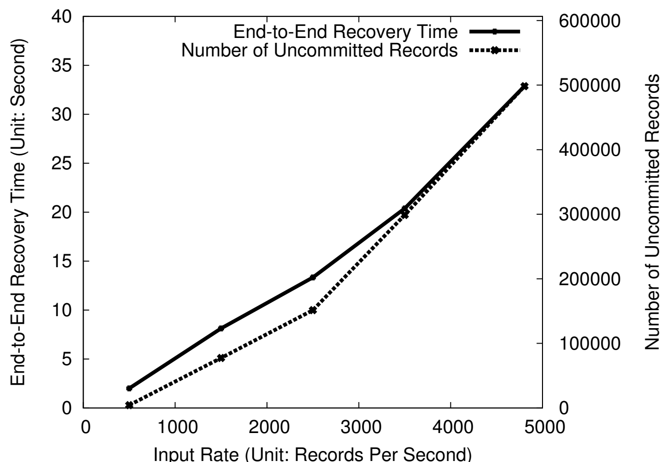


Figure 3.11: *The total recovery time for a 64-GB B^+ index and the number of uncommitted pending update requests in the replay window as the input update request rate is varied before the crash.*

Index Structure	Locating the Youngest Log Record (second)	Reconstructing Per-Block RQs (second)	Number of Log Records in Replay Window	Number of Log Records Put Into RQs
B^+ Tree	0.87	32	498370	448504
R Tree	0.9	27	376753	338909
K-D-B Tree	0.91	29	609834	548758
Hash Table	0.86	23	1897640	1364983

Table 3.2: *The break-down of the recovery processing time for four database index implementations, and the number of log records that are in the replay window and that are actually put into per-block Request Queues (RQ).*

through aggressive disk request batching, BOSC is able to log more than 50000 per-insertion-request log records per second, or about 20 μs per log record. Finally, even with such high logging efficiency, BOSC is able to keep the log disks' space utilization above 70%.

There are two major steps in BOSC's recovery procedure: (1) identifying the youngest log record and (2) reconstructing the in-memory per-block request queues by analyzing the log records between the youngest log record and its associated global frontier. Because Step (1) uses a binary search through the logging disk array, it typically takes between 0.8 to 0.9 seconds to complete.

Table 3.2 shows the time required by each of these two steps when recovering four database index implementations. In addition, Table 3.2 shows the number of log records in the replay window between the youngest log record and its global frontier,

and the number of log records that are actually put into the per-block request queues. The difference between the two is the number of log records in the replay window that have already been committed before the crash.

The time required by Step (2) depends on the number of uncommitted pending updates, which in turn depends on the input request rate. To evaluate how the total recovery time scales with the input rate, we ran a random update workload with varying input request rates to update records in a 64-GB B^+ tree with the following configuration: 256-MB buffer memory, and 16-byte index record. In each run, we issued about 64 million update requests, shut down the B^+ tree machine, restarted it and measured its recovery time.

Figure 3.11 shows that the total recovery time of a BOSC-based B^+ tree implementation indeed increases with the input request rate, because higher input request rate populates the per-block request queues faster and accumulates more uncommitted pending updates in the request queues when the system is shut down. These pending updates need to be scanned and reconstructed in Step (2) of the recovery process. As expected, increase in the total recovery time is roughly linearly proportional to increase in the number of uncommitted pending updates, as shown in the right Y axis of Figure 3.11.

Chapter 4

Continuous Data Protection (CDP)

4.1 System Architecture

As shown in Figure 4.1, a *Mariner* storage system consists of six types of storage nodes. A *client* node, which could be a file or database server, accesses data in a virtual storage device through the iSCSI protocol. The current data of a virtual storage device is stored on a master *storage* node, and replicated on a local mirror *storage* node. The virtual storage device's historical versions are maintained on a *logging* node (called Trail node from this point on), which also serves as a control gateway for remote replication. Data writes are first committed to *remote logging* nodes and then propagated to *remote storage* nodes. *Manager* node is used for system configuration, administration, monitoring and failure recovery. A typical *Mariner* system contains multiple client nodes, storage nodes, Trail nodes, remote logging nodes and remote storage nodes, but only one manager node. A Trail node can be shared by multiple master and mirror nodes.

With CDP, *Mariner* allows users to roll back a virtual storage device to any point within the protection window. Users can only read and write the current or read any historical snapshot of a virtual storage device. To maintain the file system consistency for a particular point-in-time storage snapshot, *Mariner* may need to perform a fsck-like recovery procedure on the snapshot to return a storage view with consistent file system metadata. This recovery procedure needs to modify a historical storage snapshot, but the associated disk writes are held in a temporary buffer and are thrown away when the snapshot is no longer needed.

Read requests for the current data on a virtual storage device are serviced by its associated storage nodes. Write requests for the current data on a virtual storage device are serviced by its associated Trail node and storage nodes. More specifically, a logical disk write request is first sent to the corresponding Trail node, which logs it

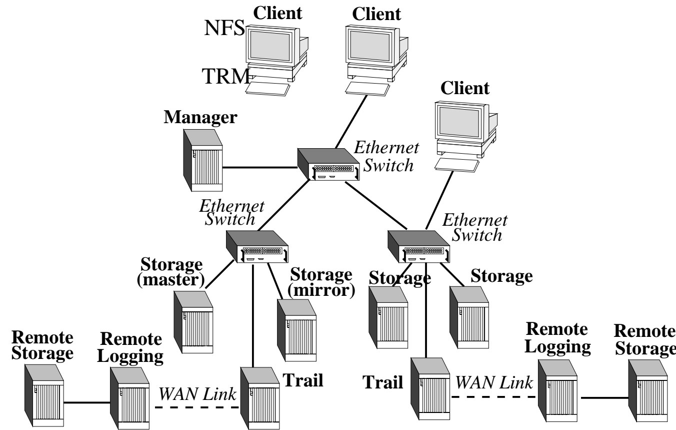


Figure 4.1: *A Mariner storage system consists of six types of nodes:* client *nodes that issue data access requests,* manager *nodes for system configuration and administration,* storage *nodes that hold local replicas of current data,* Trail *or* logging *nodes that maintain historical data and serve as a gateway for remote replication, and remote logging/storage nodes that keep a remote copy of current data.*

to disk and returns an OK reply to the requesting client. Then the client writes it to one or multiple storage nodes, depending on the degree of local mirroring supported. As far as a *Mariner* client is concerned, a disk write is completed when it receives an OK reply from the Trail node. Because of track-based logging, *Mariner* clients experience very low disk write latency. To reduce the performance penalty associated with sending a disk write's payload to multiple nodes, *Mariner* uses TRM to duplicate the payload packet in the network.

The Trail node of a virtual storage device services all read and write requests for that device's historical data, and batches multiple disk writes to replicate them to a remote site more efficiently. Because of space constraints, the details of remote replication are omitted in this technical report.

4.2 Low-Latency Disk Array Logging and Logging-based Replication

The original Trail design [12] moves the log disk's head to the next track after each write operation to ensure that the disk head is always on an empty track. Therefore, the log records are contiguous on a track-by-track rather than byte-by-byte basis, hence the name track-based logging. This per-write disk head movement incurs a track-to-track seek delay for every write operation, and results in low disk space utilization. The modified Trail design allows multiple physical writes per track and uses an array of log disks to further mask track-to-track seek delays.

Mariner maintains a disk request queue for each log disk. At any point in time, one of the log disks serves as the active disk. In the beginning, *Mariner* randomly

chooses one of the log disks as the active disk. Once a log disk becomes the active disk, it remains as the active disk until the waiting time of the oldest pending request exceeds a threshold, T_{wait}. Whenever a new logical disk write request arrives at a Trail node, *Mariner* inserts the request to the active disk's queue as long as the waiting time of its oldest pending request is smaller than T_{wait} and there is enough free space in the current track to accommodate the new request; otherwise *Mariner* dispatches the request batch in the active disk's queue, and chooses another log disk as the active disk and inserts the request to its queue.

To choose a new active disk for an incoming write request, *Mariner* computes the time at which the write request could be written to each log disk, and selects the one that can write the request to disk at the earliest. When computing an incoming disk write request's write time on a log disk, *Mariner* takes into account the current position of the log disk's head and the possibility of batching the request with others already in the disk's queue. For those log disks that are currently idle, *Mariner* only needs to consider the delay due to batching.

A key design decision in *Mariner* is to encourage batching of multiple logical disk writes into one physical disk by dispatching a new write request to the active disk, rather than to the disk with the earliest write time for that request. As we will show in Section 7, this design choice significantly increases *Mariner*'s batching efficiency and thus effective throughput.

For every logical disk write, *Mariner* creates a log record that contains the write's Logical Block Address (LBA), timestamp and payload, and writes it to the log disk chosen for the request. To facilitate accesses to historical data, *Mariner* maintains an index structure to map a disk block's logical block number and a timestamp to the physical block number of the corresponding historical version. This index data structure is maintained by a user-level daemon and organized as a B^{+}tree residing on a different disk, and contains only the log records of those logical disk writes in the protection window. Because the log record of each logical disk write is self-contained, *Mariner* can reconstruct the index tree by scanning the log disks. Therefore, *Mariner* can afford to batch updates to the index tree due to disk writes and perform them asynchronously.

Trail is currently implemented under the Linux 2.6 kernel as a virtual device driver between the file system and the physical disk driver, as shown in Figure 4.2. It dispatches logical disk write requests to the per-log-disk request queues, maintains a disk block buffer cache to facilitate the service of current data accesses, and a B-tree cache to facilitate the look-up of historical versions of disk blocks.

To implement track-based logging, *Mariner* statically extracts the physical disk geometry information from every log disk, and then uses a disk head position estimation algorithm to predict each log disk's disk head position at run time. More concretely, after a physical disk write is completed, *Mariner* records the LBA of its last sector, LBA_0, and its completion timestamp T_0. Assuming the disk head stays in the same track, when the next write arrives at T_1, *Mariner* estimates the disk head's

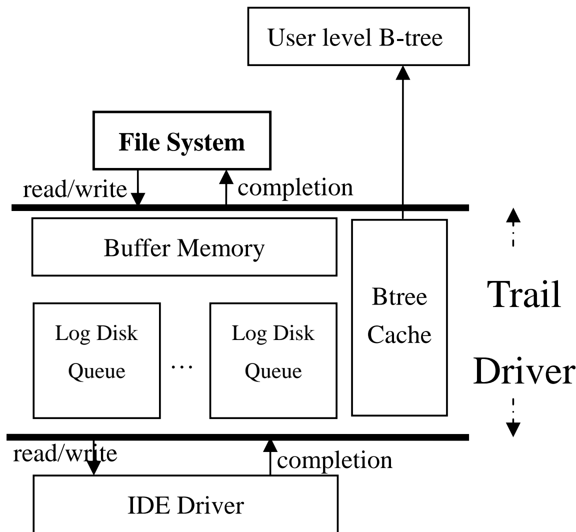


Figure 4.2: *The software architecture of* Mariner*'s Trail node. The Trail module, which sits between the file system and the physical disk driver, manages a disk block buffer cache, a B-tree cache, and a set of disk request queues, one for each log disk. The user-level B-tree daemon maintains the index tree for mapping a disk block's LBA and timestamp to its corresponding physical block.*

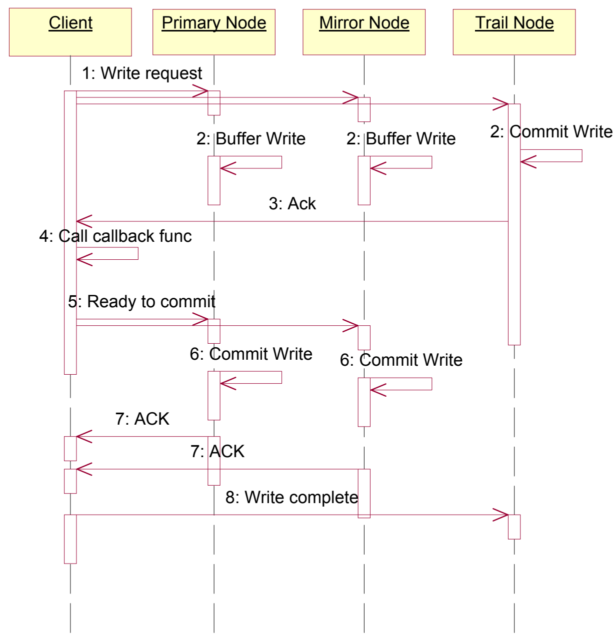


Figure 4.3: The message sequence used in the modified two-phase commit protocol when there is no device failure.

current position $CurrentLBA$ using the following formula:

$$CurrentLBA = SPT \cdot \frac{(T_1 - T_0)\ mod\ RoTime}{RoTime} + LBA_0 \tag{4.1}$$

where SPT is the number of sectors in the current track, $RoTime$ is the disk's full rotation time. The final predicted position, $DestinationLBA$, is $CurrentLBA + Lookahead$ to account for such delay as the controller delay. $Lookahead$ is an empirical value chosen to avoid a full rotation. For the IBM Deskstar DTLA-307030 disk, this value is set to be 22 sectors. The accuracy of the above disk head position estimation algorithm decreases with the value of $T_1 - T_0$. To ensure the algorithm's accuracy is always adequate, *Mariner* issues additional dummy disk reads to guarantee that $T_1 - T_0$ is always below a threshold, T_{idle}, even when the input load is low.

To satisfy CDP's log space requirement, *Mariner* allows multiple physical writes to go to the same track in order to use the log disks' space more efficiently. However, higher log disk space utilization efficiency means longer rotation latency because it is less likely that when a new write request arrives at a log disk's queue, the disk's head happens to be on a sufficiently large free region that can hold it. To determine when to switch a log disk's head to the next free track, *Mariner* uses the following metric to gauge the degree of fragmentation of the current track:

$$F = \frac{ServicedReqNum}{10 \cdot (1 - Utilization)} \tag{4.2}$$

where $ServiceReqNum$ is the number of write requests already written to the current track and $Utilization$ is the percentage of the current track that is already occupied. The larger the values of $ServiceReqNum$ and $Utilization$, the more fragmented the current track. As *Mariner* can self-describe its data logging, $ServiceReqNum$ can be extracted from the hard drive in case of crashes. After a log disk services a physical write request, *Mariner* computes its current track's fragmentation metric. If the metric's value exceeds a pre-defined threshold, T_{switch}, and its request queue is empty, *Mariner* issues a seek command to move the disk's head to the next track. To minimize the delay of the track-to-track seek, the destination LBA of the seek command (which is a write command for IDE drives because IDE drives only support two operations, read and write) is set to $CurrentLBA + CurrentSPT$, where $CurrentSPT$ is SPT of the current track.

Mariner leverages Trail's low-latency disk write capability and a modified two-phase commit protocol to replicate data asynchronously without compromising data integrity. Figure 4.3 shows the message sequence used in this modified two-phase commit protocol. The *Mariner* client issuing a logical disk write request serves as the coordinator, and the Trail, master and local mirror nodes are the participants. The client first sends the write request to the Trail, master and local mirror nodes of its virtual storage device. Upon receiving this request, the Trail node immediately commits the request to its log disk and sends an ACK back to the client after it is

done, but the master and local mirror nodes simply buffer this request, waiting for further instruction. When the client receives the Trail node's ACK, it notifies the master and local mirror nodes to commit the buffered write request, and resumes the thread that issues the write request by invoking the associated call-back function. The master (local mirror) node sends back an ACK after completing the write request to disk. Finally, the client asynchronously informs the Trail node about each write request's completion status on the master and local mirror nodes, so that the Trail node can keep track of their progress. Whenever possible, the messages of this modified two-phase commit protocol are piggy-backed with normal iSCSI command packets. In addition, the protocol has built in extensive retry mechanisms to deal with such failures as packet loss, message corruption, TCP connection time-out and iSCSI connection time-out.

This modified two-phase commit protocol is different from the standard two-phase commit protocol because its goal is to commit a write request on as many participant nodes as possible, rather than to achieve all-or-nothing consistency among participants. Therefore, the coordinator does not need to collect ACKs from all participants before committing a write request. Instead, it keeps a record of who has committed which requests so that after a failed node recovers, the system knows how to replay which write requests to bring it to synchronization with others. As the client informs the Trail node about the other two nodes' write progress, the other two nodes snoop the network and also each keep a local write progress log about others. When the Trail node is alive, it is the Trail node's write progress log that serves as the ground truth. When the Trail node is dead, it is master node's write progress log that serves as the ground truth.

When the master node dies, the local mirror node becomes the master node, and each write request is sent to the new master node and the Trail node; after the old master node recovers, it contacts the Trail node, which keeps track of each node's write progress, to replay missing write requests, and becomes the local mirror node. When the local mirror node dies, each write request is sent to the master node and the Trail node; after the local mirror node recovers, it contacts the Trail node to replay missing write requests, and continues to be the local mirror node. When the Trail node dies, CDP and remote replication cease to function, and each write request is sent to the master and local mirror nodes; after the Trail node recovers, it contacts the master node for synchronization and continues to act as a logging disk.

4.3 Transparent Reliable Multicast (TRM)

When a *Mariner* client sends a write request to the Trail, master and local mirror nodes in the modified two-phase commit protocol, it uses TRM to reliably multicast the write request and achieve almost the same network efficiency as the no-replication case.

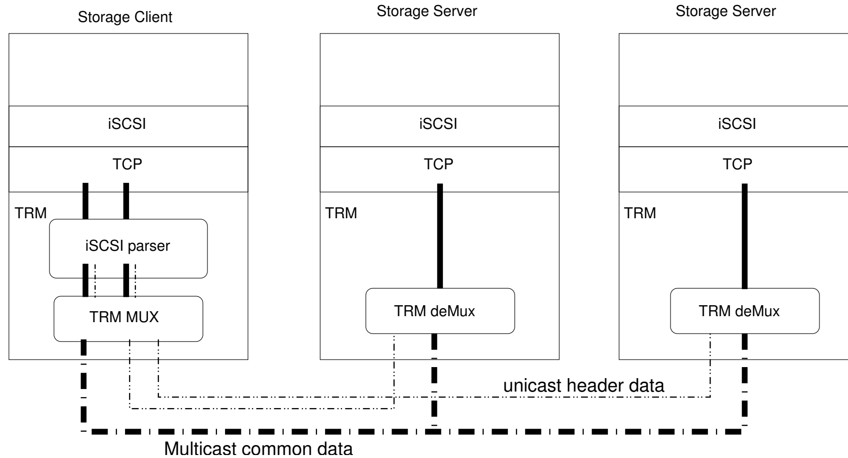


Figure 4.4: Data flow of an iSCSI-based TRM system supporting 2-way replication. An iSCSI protocol parser keeps track of contents in TCP connections corresponding to the two iSCSI sessions involved in data replication. The first iSCSI copy associated with each SCSI write request is sent as multicast packets, whereas the headers of the second copy are merged and sent as a unicast packet. The TRM layer at the receivers reconstructs each individual TCP stream based on the received unicast and multicast data.

4.3.1 Multicast Transmission of Common Payloads

Logically, TRM is a software layer residing below the TCP/IP stack that constantly monitors the contents of outgoing TCP connections to look for common bytes. When packets from a set of TCP connections share common bytes, TRM sends only one of them as an Ethernet multicast packet to the destination nodes associated with these connections. The software architecture of iSCSI-based TRM is shown in Figure 4.4. There are two key components in TRM: (a) the client side component monitoring TCP connections for common data payload and constructing multicast packets that carry these common payload, and (b) the server side component reconstructing the original TCP streams based on the payloads and headers received.

The client-side TRM component of the current *Mariner* prototype includes an iSCSI parser that tracks iSCSI commands in iSCSI-carrying TCP connections. Once detecting common write payloads among the three iSCSI connections to a virtual storage device's Trail, master and local mirror nodes, it asynchronously merges the three write requests sharing the same payload by sending the earliest-arriving copy of each iSCSI write request using multicast and the headers of the other two copies as unicast packets to their corresponding nodes. Asynchronous merging does not require the three TCP connections involved in data replication to be strictly synchronized.

The server-side TRM component reconstructs the individual TCP streams by taking the common payloads, which are received as multicast packets, and headers, which are received as unicast packets, putting them together into original unicast

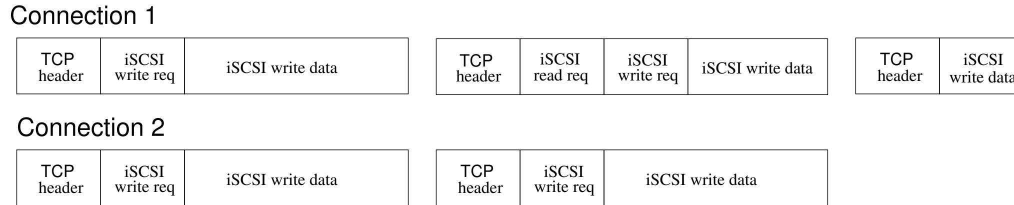


Figure 4.5: Because Connection 1 contains both READ and WRITE commands and Connection 2 contains only WRITE commands, packet-by-packet comparison between these two connections cannot detect the payloads of WRITE commands.

TCP packets, and passing them up to the TCP/IP stack for further processing.

When there is packet lost, TRM relies on TCP to retransmit the lost packets and therefore does not require any additional machinery to support reliable transmission. Retransmitted packets are always transmitted as unicast packets. As a result, packet retransmission may cause the TCP connections being merged to become desynchronized.

4.3.2 Common Payload Detection

A *Mariner* client sends each iSCSI read request only to the master node, but sends each iSCSI write request to the Trail, master and local mirror nodes. Therefore, the TCP connection associated with the master storage node contains more iSCSI commands than the two TCP connections associated with the local mirror and Trail node. Because TCP is a stream protocol and does not preserve application-level packet boundaries, packet-by-packet comparison may not be able to reliably detect all common payloads among connections, as shown in Figure 4.5, which calls for a more expensive byte-by-byte comparison approach to detect common payloads.

To reduce the performance overhead associated with common payload detection, *Mariner* exploits protocol-specific knowledge. More concretely, the current *Mariner* prototype parses the iSCSI commands in each of the three TCP connections and is able to pinpoint the precise location of the payload portion of each iSCSI write request. From these locations, the *TRM* layer can easily detect common payloads without resorting to expensive byte-by-byte comparison.

4.3.3 Tree-Based Link-Layer Multicast

Mariner uses Ethernet as its storage area network, and leverages Ethernet's link-layer multicast to deliver the common payloads of duplicated iSCSI write requests. Link-layer multicast on Ethernet networks is implemented as a broadcast-and-filter mechanism. That is, an Ethernet switch treats a link-layer multicast packet as a broadcast packet, and broadcasts it to all of its ports except the one from which the packet comes. When a node receives a link-layer multicast packet, it filters out the

packet and drops it if the packet's destination MAC address is not registered on the node's network interface card (NIC). Although the broadcast-and-filter mechanism is easy to implement, it imposes a serious load on the switches as each multicast packet appears in every active link of the layer-2 network in which the packet source resides.

To eliminate the performance problem associated with the broadcast-and-filter approach to link-layer multicast, *Mariner* exploits the Virtual LAN (VLAN) technology [167, 168] available in commodity Ethernet switches, which was originally designed to divide a layer-2 network into multiple broadcast domains and limit the scope of broadcast packets. Operationally, each Ethernet packet is tagged with a distinct VLAN ID, and each switch maintains a separate routing table for each VLAN going through it. When an Ethernet switch receives a packet, it first identifies the routing table corresponding to its VLAN ID and routes the packet based on the corresponding routing table. A VLAN is identical to a layer-2 network in every aspect, including owing its spanning tree, i.e., per-VLAN spanning tree (PVST). The multiple spanning tree (MST) [169, 170] protocol allows one to set up multiple spanning trees on a layer-2 network, each associated with a distinct VLAN.

A standard Ethernet network builds up its spanning tree using a distributed spanning tree construction algorithm (IEEE 802.1D) [171]. The VLAN protocol allows the network administrator to explicitly specify the spanning tree connecting a VLAN's members by configuring the priority of using each physical link in constructing the VLAN. All modern Ethernet switches provide an SNMP interface for this priority configuration, which makes it possible to programmatically set up and tear down VLANs and their spanning trees at run time. The number of VLAN groups that a commodity Ethernet switch supports is between 256 and 1000.

Using PVST, *Mariner* puts a storage client and the Trail, master and local mirror nodes of its virtual storage device in a VLAN, and constructs a spanning tree to connect them. Whenever a client needs to multicast the common payload of duplicated iSCSI write requests, it sends the payload as a broadcast packet on the associated VLAN, which reaches the corresponding nodes along the VLAN's spanning tree.

4.4 Incremental File System ChecKing (iFSCK)

4.4.1 Snapshot Access

A block-level CDP system typically provides a programming interface to access any point-in-time disk image snapshot in the data protection window. To provide end users a file versioning view on top of this interface in a NFS-like environment is challenging, especially if modifications to the infrastructure must be minimized. This subsection describes how an end user can access a point-in-time disk image snapshot under NFS. Similar snapshot access mechanisms can be developed for other network file access protocols such as CIFS. *iFSCK* is independent of snapshot access mechanisms.

In a typical NFS environment, the end user accesses files from an NFS client, which uses NFS to communicate with an NFS server whose data is protected by a block-level CDP system. To access a particular directory of a particular point-in-time snapshot of her file system, the user specifies the directory's pathname (P) and the target timestamp (T) in a request, which is sent to a dedicated daemon in the NFS server. Upon receiving such a request, the daemon requests the associated block-level CDP system to create a disk image snapshot at T as an iSCSI target, instructs the NFS server to bind this iSCSI target to a local iSCSI initiator device, mounts a local directory on the iSCSI initiator device, and exports this local directory. Finally, another daemon on the NFS client mounts the target directory P within the NFS server's exported directory to a local directory it creates. The client-side daemon communicates with the server-side daemon through a proprietary control protocol. The above snapshot access procedure interoperates with the standard NFS/iSCSI protocol without requiring any modification.

To reduce the set-up time required to access a file system snapshot, at the block-level CDP system a pool of virtual iSCSI targets is created in advance, and the NFS server binds them to its local virtual devices. This way, upon receiving a snapshot access request the server-side daemon can directly allocate one of these virtual devices on the NFS server to the requested snapshot.The client-side daemon mounts the target directory within the chosen virtual device to the user-specified local directory. When a snapshot is no longer needed, the corresponding virtual devices are returned to the pool.

4.4.2 Ensuring File System Consistency

Because of file system caching, file system-level updates do not immediately trigger disk block-level operations. By default in Linux, the *pdflush* daemon wakes up every 5 seconds and flushes to disk those dirty buffer cache pages that are older than 30 seconds. Therefore, when a user asks for a point-in-time snapshot of a storage volume at time T, the returned snapshot may not capture all file system-level updates that take place before T, and more importantly it may not be even file system-consistent.

In addition to user data blocks, a file system also includes a set of metadata for managing its disk storage space and the relationships and attributes of its files. When a user application modifies a file system object, the modify operation may trigger multiple updates to the file system's metadata. In theory, a file system update operation and the file system metadata updates it triggers should be performed *atomically*, as if they are batched into a transaction, so that the file system state is always consistent. However, for performance reasons, existing file systems, for example, the *ext2* file system under Linux, choose not to implement these updates as transactions. Instead, they resort to periodic buffer flushing to amortize the disk I/O cost of making file system metadata persistent. Under such file systems, how to quickly convert a point-in-time storage volume snapshot into its corresponding file system-consistent snapshot is a technical challenge for block-level CDP systems.

Given a disk image snapshot for a timestamp T, *iFSCK* is designed to identify all disk-level updates after T that correspond to file system-level update operations before and at T, and replay them against the snapshot to ensure that all file system-level updates before T are completed successfully. The key implementation challenge of this approach is how to accurately correlate block-level disk updates with their associated file system-level updates. *iFSCK* solves this problem by assuming that it knows the disk locations of file system metadata and the internal structures of these metadata. Given a timestamp T and its corresponding disk snapshot V, *iFSCK* transforms V into a file system-consistent snapshot by using the following algorithm, assuming the host file system is an ext2 file system:

File Operation	Related Updates
Creation	Inode Bitmap update, Inode Table update (new Inode and its parent Inode), parent Dentry update, Group Descriptor update (free Inode count), Superblock update(free Inode count)
Deletion	Inode Bitmap update, Inode Table update (deleted Inode and its parent Inode), Parent Dentry update, Block Bitmap update, Group Descriptor update (free Inode count, free block count), Superblock update(free Inode count, free block count)
Renaming	Inode Table update(parent Inode), parent Dentry update
Truncation	Inode Table update, Data Bitmap update, Data Block update, Group Descriptor update (free block count) , Superblock update (free block count)
File Write	Inode Table update, Data Block update
File Append	Inode Table update, Data Bitmap update, Data Block update, Group Descriptor update (free block count), Superblock update(free block count)

Table 4.1: File operations and the corresponding related metadata updates.

1. *iFSCK* scans disk block updates that took place within a time window $[T - LB$ $, T + UB]$, where $T - LB$ is the lower bound of the time window and $T + UB$ is the upper bound of the time window, and classifies them into the following types: *Block Bitmap* updates, *Inode Bitmap* updates, *Inode* updates and *Data Block* updates. By examining Inode updates in more detail, one can further sub-divide Data Block updates into *User Data Block* updates and *Directory Block* updates. Each of the file system metadata updates listed above modifies a distinct range of V's block address space. From *Directory Block* updates, *iFSCK* identifies all file/directory creation, deletion and renaming operations within $[T - LB, T + UB]$.

2. *iFSCK* extracts from each disk block update individual metadata update operations triggered by file-level update operations. More specifically, for each updated block, *iFSCK* retrieves its previous version, and performs a byte-by-byte comparison to determine which part of the block and which metadata entries (e.g. Inodes or bitmap entries) in that block are modified. Therefore, even if multiple file system-level update operations result in a single block update, *iFSCK* can correctly identify each of them. In addition, *iFSCK* can also identify modifications to indirect blocks, which are no different than normal data blocks, because whenever *iFSCK* recognizes an Inode update, it follows the Inode to track down its indirect, doubly indirect and triply indirect blocks, and checks if they appear in the list of updated blocks within $[T-LB, T+UB]$.
3. *iFSCK* partitions the metadata update operations within $[T - LB, T + UB]$ to groups, each of which corresponds to a file-level update operation, according to a pre-computed table (shown in Table 4.1) that lists the set of metadata update operations for file deletion, file renaming, file truncation, file write, and file append. For example, when a new file is created, a new Inode is allocated to the file (Inode Bitmap update), this Inode is properly initialized (Inode Table update), some data blocks may also be allocated to the file (Block Bitmap update) and modified (Data Block update), the directory holding this new file is modified (parent Dentry update), and so is the directory's Inode (Inode Table update). *iFSCK* detects file truncation operations by examining the file length field in the Inodes.
4. For every group that has at least one metadata update operation occurring between $T - LB$ and T, *iFSCK* includes into the *redo* list all the group's constituent metadata update operations that appear after T to make it complete, replays the final redo list to the snapshot at T, and eventually produces a file system-consistent snapshot that is after and closest to T.

In Step (3), we make the assumption that a file system update operation that logically starts before T completes all its disk-level updates before $T + UB$. Because pdflush wakes up every 5 seconds, we assume the disk updates associated with a file system update operation span at most 5 seconds. Therefore, for a file system update operation that occurs exactly at time T, the latest disk update operations associated with it must occur before $T+5$, and these disk updates must be flushed to disk before $T + 35$. However, because the pdflush's wake-up timing may be mis-aligned, in the worst case, they must be flushed before $T+40$. Therefore, in general UB is set to be $T_{wake-up}+T_{flush}+T_{span}$, where $T_{wake-up}$ corresponds to the periodic wake-up interval of pdflsuh (5 seconds in Linux), T_{flush} to the flushing threshold (30 seconds in Linux) and T_{span} to the time span of a file system update's disk-level update operations (assumed to be 5 seconds). On the other hand, LB has to be large enough so that for every file system update some of whose disk-level updates occur before $T - LB$, there is at least one of its disk-level updates takes place in $[T-LB, T]$. This prevents the anomaly that even though some disk-level updates associated with a file system

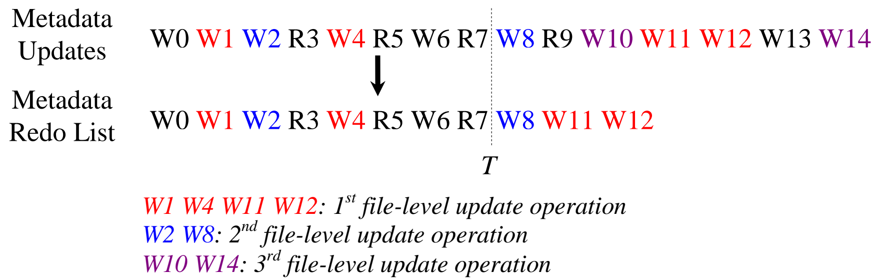


Figure 4.6: An example showing how *iFSCK* extracts a metadata redo list from the metadata updates within a time period. T is the target timestamp.

update occur before T, *iFSCK* mistakes it for one whose first disk-level update occurs after T. As in UB, LB is also set to $T_{wake-up} + T_{flush} + T_{span}$, i.e. 40 seconds.

Figure 4.6 illustrates how *iFSCK* extracts metadata redo list from the metadata updates within a period. R and W stand for metadata read and metadata write, respectively. iFSCK includes file system metadata update operations that are part of file-level update operations and occur before the target time stamp T, which is right after R7. In this case, the first and second file-level update operations occur before T and therefore should be redone, whereas the third file-level update operation occurs after T and therefore should be undone.

iFSCK supports three file system consistency levels, each incurring a different performance overhead. The strongest consistency level guarantees consistency of all file system metadata, including Block bitmap, Inode bitmap, directory contents, etc. This consistency level is also known as crash consistency as used in standard file system checkers, and is useful in creating a read/write snapshot. The second strongest consistency level guarantees consistency only for a selected subset of file system metadata (e.g. excluding allocation bitmap information) and directory contents. This level is useful for creating a read-only snapshot. The weakest consistent level provides the same consistency guarantee as the second strongest level except that its scope is restricted to a particular directory rather than the entire file system. For the weakest consistency level, *iFSCK* iterates through each directory entry in the target directory, making sure that the Inode number corresponding to each directory entry refers to a valid in-use Inode, i.e. the Inode is within a valid range, is allocated and the Inode link count is not zero, etc. The performance cost of weaker consistency levels is lower because disk updates associated with certain file system metadata updates (e.g. Block and Inode bitmap) can be ignored.

Although the above algorithm is designed for ext2 file systems, its underlying principle can be easily applied to other Unix file systems (e.g. Soloris UFS) that have similar file system structure to ext2's. For journaling file systems such as ext3 and NTFS, the journal already explicitly contains the file system metadata updates and their grouping information, so most of the analyses in *iFSCK* are no longer necessary. For example, ext3's file system consistency check tool first applies its metadata journal, and then continues with a ext2-style full file system check if the

file system's superblock indicates that further checking is required. We have ported *iFSCK* to ext3 so as to derive the redo list directly from its journal rather than from scanning disk updates within $[T-LB, T+UB]$. In the case that an ext3 file system's journal is corrupted, *iFSCK* invokes its own incremental checker rather than ext2's file system checker.

4.5 User-level Versioning File System (UVFS)

4.5.1 Overview

In *UVFS*, a file system object is uniquely defined by its pathname, rather than by its internal representation, such as an Inode. Because of hard links, an Inode can have multiple pathnames, each of which still corresponds to a unique file system object. If a file system object is renamed, it becomes a different file system object. An *incarnation* of a pathname corresponds to a file system object with that pathname from its creation to its deletion (including rename). If a file system object is created and deleted multiple times, it has multiple incarnations. Different incarnations of a file system object can use different Inodes. An incarnation can have multiple *versions*, each corresponding to a distinct modification within the incarnation. Logically, versions associated with one incarnation are unrelated to versions associated with another incarnation with the same pathname. The second column in Table 4.1 shows the file incarnation/version modifications associated with different file update operations.

Given a point-in-time disk snapshot, *UVFS* leverages the original host file system, from which the snapshot is taken, to properly interpret its contents, and makes only two assumptions about the file system: (1) support for *last modify time* field and a system call to access it such as *stat* in Linux and (2) support for a system call that accesses the contents of a directory file such as *readdir* in Linux. Because all mainstream operating systems, including Linux [172], BSD Unix [173], Solaris [174], AIX and Windows XP/Vista, support these two features, and that *UVFS* is implemented completely at the user level, *UVFS* is portable across different operating systems. However, for ease of exposition, the following description assumes an ext2/ext3 file system. Because the time resolution of the *stat* system call on Linux kernel 2.6 is 1 second, the current *UVFS* prototype cannot distinguish file versions in the same second.

UVFS provides users the following file version query operations that are commonly supported by existing versioning file systems:

- *File/directory snapshot access:* accessing a particular snapshot described by a time point and a pathname.
- *Searching for versions associated with an incarnation within a time range:* listing all the versions associated with an incarnation with a given file pathname within a specified time range.

- *Searching for incarnations within a time range:* listing all incarnations with a given file pathname within a specified time range.

- *Version search across incarnations within a time range:* listing all versions associated with all incarnations with a given file pathname within a specified time range.

- *Searching for all file/directory versions under a directory within a time range:* listing all versions of files and subdirectories that ever existed under a given directory within a specified time range.

4.5.2 Version Query Processing Algorithms

When an application on an NFS client issues a file version query, a *UVFS* agent on the client services the query by executing the following algorithms against the NFS server and the block-level CDP server. The basic primitives used in the query processing are (a) set-up/tear-down of snapshot images, (b) traversal of the file systems associated with snapshots, and (c) internal processing in the form of comparison of directory contents or timestamps.

Versions Associated with an Incarnation

When a new version of a file incarnation is created, the *last modify time* field of the incarnation's Inode must be modified. Therefore, to discover the versions of an incarnation that exist within a time range, one just needs to identify the time points at which the incarnation's *last modify time* field is modified, and to access the incarnation's snapshots at these time points. More concretely, given a pathname P and a time range $[T1, T2]$, *UVFS* first accesses P's snapshot corresponding to the time point $T2 - \delta$ and retrieves that snapshot's *last modify time*, say T. If $T \geq T1$, *UVFS* repeats the same procedure to locate the version immediately prior to the version corresponding to T, etc. If $T < T1$, then *UVFS* has found all the versions of the incarnation P within $[T1, T2]$ and the process stops. The parameter δ is chosen in such a way to ensure that whatever modifications to the file system before and at $T2$ should already be reflected to the snapshot at time $T2$ - δ.

Incarnations Associated with a File

A file pathname may refer to multiple incarnations within a time period. Each of these incarnations corresponds to a pair of creation and deletion of a file system object with that file pathname. When an incarnation with a particular pathname is created or deleted, the immediate parent directory containing the file pathname must be modified, as is its *last modify time* field. To discover all incarnations of a given pathname P within a time period $[T1, T2]$, *UVFS* first extracts the pathname for P's immediate parent directory, say Q, identifies all versions of Q within $[T1, T2]$, and compares adjacent versions of Q to determine if the difference between them is

related to the creation or deletion of P. Every time a new instance of P appears in a new Q version, a new incarnation of P is created; every time an existing instance of P disappears in a new Q version, the corresponding incarnation of P is considered over.

However, identifying all versions of Q within $[T1, T2]$ itself is non-trivial, because it requires identifying all incarnations of Q within $[T1, T2]$; this in turn requires identifying all versions and incarnations of Q's immediate parent directory within $[T1, T2]$, all versions and incarnations of the immediate parent directory of Q's immediate parent directory within $[T1, T2]$, etc. Fortunately, this recursive process eventually stops because by definition, there is only one incarnation for the root directory of every file system.

Versions Associated with a File

This operation is simply built upon the above two operations. Given a pathname P and a time period $[T1, T2]$, *UVFS* first discovers all incarnations of P within the specified time range, and then extracts all the versions associated with each of these incarnations.

All File Versions Under a Directory

To service this type of file versioning queries, *UVFS* first locates all versions of the specified directory, then extracts all versions of every pathname that ever appears in any version of the specified directory, and finally outputs a union of all the file and subdirectory versions found.

4.5.3 Optimizations

To service a file versioning query, a *UVFS* agent needs to set up snapshots, traverse file systems associated with these snapshots, and perform some internal processing such as timestamp or directory content comparison. Typically the performance cost of internal processing is negligible. The cost of setting up snapshot consists of two components: (a) establishing NFS and iSCSI connections and (b) invoking *iFSCK* to fix an established disk snapshot. To reduce the cost of (a), *UVFS* reuses NFS/iSCSI connections and virtual devices created at the block-level CDP server that are set up for accesses to subsequent snapshots. To reduce the cost of (b), *UVFS* takes an optimistic approach by assuming that most point-in-time disk snapshots are consistent, and invokes *iFSCK* lazily, specifically only when either *readdir* or *stat* returns with an unexpected error during file system traversal. In addition, when *UVFS* does invoke *iFSCK*, it applies consistency check only to the path from the root to the target file or directory, and focuses only on their last modified times and directory contents while ignoring other types of file system metadata.

To reduce the performance cost associated with file system traversal, *UVFS* employs various forms of caching to reuse efforts invested in previous snapshot accesses.

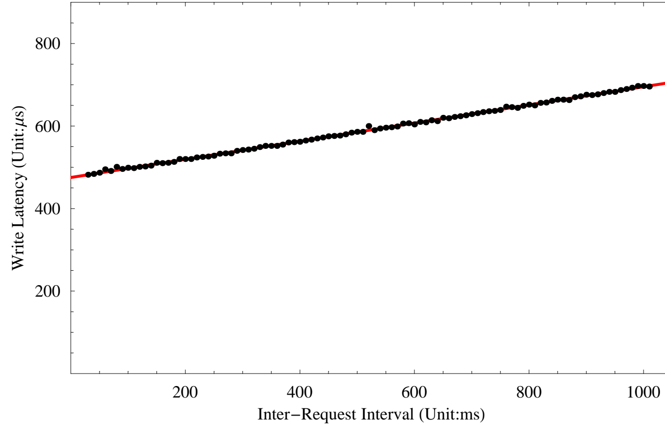


Figure 4.7: The impact of the inter-request interval on the measured write latency as seen by the Trail device driver.

UVFS caches the *last modify times* for files and directories in a snapshot, and reuses them for subsequent file versioning queries that need to access the same snapshot. Moreover, to exploit the significant redundancy among the snapshots that are established during the service of a file versioning query, *UVFS* adds a simple caching mechanism on the CDP server that caches disk blocks which are recently accessed and associated with previously established snapshots. This disk block caching mechanism is meant to reduce the disk I/O cost associated with traversal of temporally adjacent snapshots that overlap with each other significantly.

4.6 Performance Results and Analysis

4.6.1 Evaluation Methodology

We first evaluate each component of *Mariner*, including Trail, modified two-phase commit, and TRM, and then the entire system as a whole. We then evaluate the *UVFS* and *iFSCK* tools.

We use synthetic workloads to stress-test each *Mariner* component and real traces to evaluate *Mariner*'s end-to-end performance. The four traces used in this study include both file system and database workload:

1. IO Trace

    - Lair62b The original Lair62b is an NFS RPC trace collected on an NFS server by the SOS project of Harvard University [175]. This trace is converted into a block-level disk access trace through an FFS-like file system simulator, which models the I-node and data blocks and ignores other meta-data [176]. The block size is 4KB and the trace is a one-day long trace with 12631475 requests, 2816401 of which are writes.

- OLTP(On-Line Transaction Processing) OLTP trace is a database buffer cache access trace collected on an IBM DB2 database running IBM's TPCC benchmark of 1,000 warehouses [176]. The trace is featured by a large amount of random access. The block size is 4KB.
- DSS(Decision Support System) DSS trace is another database buffer cache access trace collected on an IBM DB2 database running IBM's TPCH benchmark [176]. The trace contains several large sequential scan of a big table. The block size is 4KB.
- Cello99 Cello99 is a low-level disk I/O trace collected from a HP UNIX platform. Since the trace is filtered by the file system cache, the spatial locality is quite poor. The block size is 8KB.
- MS-SQL-Large I/O trace MS-SQL-Large trace is a disk I/O trace collected from a Microsoft SQL database server running the standard TPC-C benchmark for two hours. The TPC-C database consists of 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The trace is filtered by a 1 GByte SQL server cache. The block size is 4KB and the trace has 5390743 requests, 866029 of which are write ones.
- MS-SQL-Small I/O trace This trace is collected with the same setup as the previous trace except that the server cache is 64 MB.

2. File-level Trace

   - Postmark Postmark [177] is a file system benchmark emulating very heavy small file workload. The benchmark creates a specified number of files, performs various file system operations and finally deletes those files. For all runs, we run Postmark with 10,000 files, 1000 subdirectories and 50,000 transactions.
   - Lair Trace played by TBBT [178] trace player. NFS server is the Trail client side. It is the same trace as Lair62b, the only difference is it is played at NFS level. It is a one-day trace on Oct 21, 2001 worth of 2GB data in total.

The testbed used in this study consists of one client node, a Trail node, a master node and a local mirror node, all of which are connected by a Netgear GS508T 8-port Gigabit Ethernet switch. The Trail node is a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a 400 MHz front-side bus, an embedded Gigabit Ethernet Card, and up to five ATA/IDE hard disks, each of which is a 80-GB IBM Deskstar DTLA-307030 disk. The master node, the local mirror node and the client node are PowerEdge SC1425 machines with an Intel 3.8 GHz CPU, 1 GB memory, a 800 MHz front-side bus and four embedded Gigabit Ethernet Cards. We use UNH iSCSI implementation (version 1.6.0) [179] on the iSCSI initiator side and Linux's iSCSI Enterprise Target (*IET*) implementation on the iSCSI target side. Note that in the `fileio` mode of the *IET* implementation, each write request is synchronous

as a sync-like function is called after each write operation. In terms of performance metrics, we measure the average write latency and the I/O rate of each test run.

In this study, we first evaluate the basic track-based logging technique as this is the first time this technique is implemented on a commodity IDE/ATA drive. Then we examine the write latency of a Trail node that uses an array of log disks and supports multiple writes per track, and impacts of various configuration parameters. Next, we evaluate the effectiveness of TRM in terms of its savings in network load. Thirdly, we measure the end-to-end write latency of a logical disk write request under *Mariner*, which includes the effects of Trail, two-phase commit and TRM. Finally, we demonstrate the effectiveness of *UVFS* and *iFSCK* , respectively.

4.6.2 Low-Latency Disk Logging

In this section, we evaluate the efficiency of the first IDE/ATA implementation of Trail. In particular, we quantify the impact of disk idle time on the accuracy of the disk head position prediction algorithm, and give a detailed break-down of the disk write latency into controller processing time, rotational latency and data transmission time.

Accuracy of Disk Head Position Prediction

We issue 20000 4KB disk write requests at a fixed inter-request interval to an ATA/IDE disk and measure the average end-to-end latency as seen by the Trail device driver. Figure 4.7 shows the average end-to-end write latency increases approximately linearly with the inter-request interval, because the accuracy of disk head position prediction decreases when the temporal distance between measurement samples increases. The disk head position prediction algorithm in Section 4 assumes that a disk's platter rotates at a constant speed. However, in practice, a disk's rotation speed fluctuates dynamically, and the deviation from the constant-rotation-speed model increases over time because of accumulation.

From Figure 4.7, the minimal latency for a 4KB disk write is around 0.475msec. This is the lowest disk write latency measurement ever recorded on an IDE/ATA drive as far as we know.

One way to solve the prediction accuracy problem is to pro-actively issue disk access commands so as to ensure that the time interval between consecutive measurements used in disk head position prediction is below a threshold, T_{idle}. Although a smaller value of T_{idle} could improve the accuracy of disk head position prediction, it could also impose too much load on the disks and negatively impact the write latency of user requests. How to achieve the minimal write latency by striking an optimal tradeoff between disk head position prediction accuracy and adverse impacts of additional loads will be discussed in Section 4.6.4.

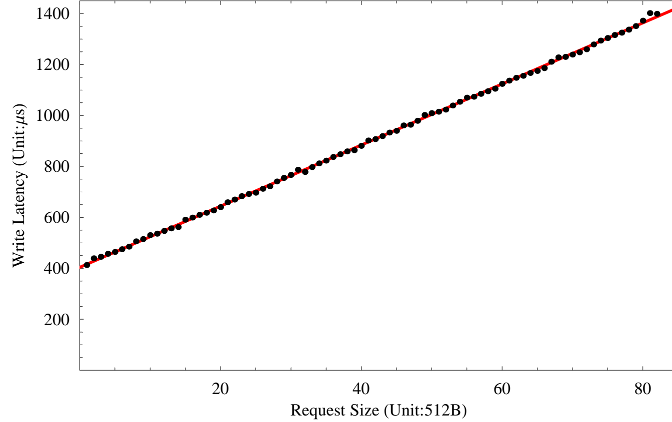


Figure 4.8: The impact of the disk write request size on the measured write latency as seen by the Trail device driver.

Analysis of Disk Write Latency

From the Trail device driver, a disk write's latency consists of a fixed controller processing delay T_c, repositioning latency T_{repo}, the seek time T_{seek} and the data transmission time T_{trans}. That is,

$$T_{latency} = T_c + T_{seek} + T_{repo} + T_{trans} \tag{4.3}$$

T_{seek} is zero in the Trail architecture because the disk head stays on the same track. T_{trans} can be calculated based on the request size and the disk's physical transfer bandwidth. To measure the fixed controller processing delay T_c, we enable the on-disk write cache to remove the rotational latency and the internal data transmission time because a write request is completed once its payload reaches on-disk write cache. We issue a sequence of 4KB requests from the device driver to eliminate all software queuing delay. The inter-request interval is set to be large enough (e.g. 1 sec) to avoid overflowing the on-disk write cache. The measured average write latency in this set-up is 0.104msec, which corresponds to T_c.

The repositioning latency consists of a rotational latency $T_{rotation}$ and the disk head settling time T_{settle}, both of which are independent of the request size. To measure $T_{rotation}$, we issue a sequence of fixed-sized write requests, whose target is set to 20 sectors plus and minus of the predicted disk head position, and compute the minimum of their write latency. The on-disk write cache is disabled in this case. In each run, a new request is issued only when the previous request is finished. Figure 4.8 shows the minimal measured write latency versus the request size of the write request sequence. When the request size is zero, the measured latency is the sum of the controller processing delay and the repositioning latency. Therefore, T_{repo} is approximately 0.4msec (Y intercept) - 0.1msec (controller processing delay) = 0.3msec.

Wait Time Limit (msec)	**Batch Size (KB)**	**Write Latency (msec)**
0	12	6.7
0.12	12.7	5.8
0.24	15	3.1
0.36	20	1.9
0.48	22	2.1
0.60	24	2.3
0.72	28	2.4
1	32	3.0

Table 4.2: *Impact of wait time limit on the batching efficiency and average write latency, where the request size is 4KB and* $T_{switch} = 6$.

4.6.3 Array of Logging Disks

Mariner's Trail node uses an array of log disks, rather than a single log disk. This subsection evaluates the effectiveness of *Mariner*'s disk request dispatching algorithm in exploiting request batching to improve the I/O rate without compromising the write latency. In this experiment, there are five log disks and each log disk is a commodity IDE/ATA hard drive connected via an independent ATA/IDE channel from the Promise Ultra100 TX2 IDE controller. We issue additional disk access requests to guarantee that the maximal time interval between consecutive accesses to each disk is at most 50msec. In addition, the maximum waiting time in the disk request queue is 0.3msec, T_{switch} is set to 6 to achieve reasonable disk utilization efficiency. Under this setup, a stand-alone Trail device can deliver 1.8msec write latency and achieve 70% disk space utilization under an input workload of 12500 writes/sec and 4KB per write request.

Batching of multiple logical writes into a physical write improves *Mariner*'s physical write efficiency and thus its effective throughput. Batching is especially useful in the face of a burst of write requests. However, batching increases the write latency because it forces those requests that arrive early to wait even when the disk is idle. To resolve this issue, *Mariner* sets a limit on a request's wait time (T_{wait}) when it batches logical write requests.

Table 4.2 shows the impact of T_{wait} on the log disk array's write latency. The workload used in this experiment is a synthetic workload that consists of 4KB write requests with a fixed inter-request interval, 60usec. When T_{wait} is set to zero, there is not much room for request batching, and the batch size, i.e., the average number of logical writes per physical write, is small, around 3 or 12KB. Smaller batch size leads to lower I/O rate for the log disk array, and eventually causes subsequent write requests to queue up and experience higher latency. On the other hand, when T_{wait} is set to 0.36msec, the resulting batch size is larger, the log disk array's I/O rate

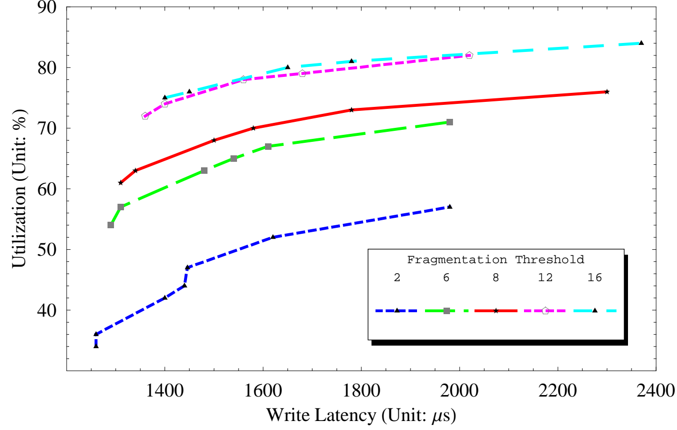


Figure 4.9: Performance impact of the choice of T_{switch} on the log disks' disk space utilization efficiency and write latency.

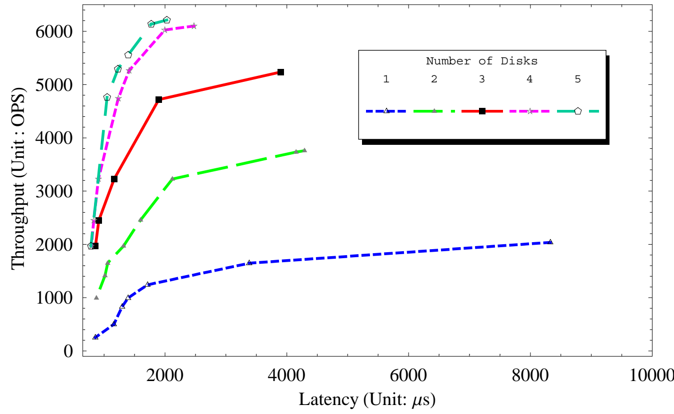


Figure 4.10: Performance impact of the number of log disks on the log disks' throughput and write latency.

improves, and the average write latency actually decreases. This result demonstrates that it is better to force requests to wait a little bit longer out front in order to improve the batching efficiency and eventually decrease the write latency for everybody. However, as T_{wait} is increased beyond 0.36msec, the write latency starts to increase again, because each request is likely to wait longer and each physical write also takes longer to complete.

4.6.4 Sensitivity Study

In this subsection, we study the performance impact of each configuration parameter in *Mariner*'s track-based logging design. There are 4 configuration parameters: (1) the threshold of the fragmentation metric T_{switch}, (2) the disk head recalibration interval (T_{idle}), (3) the wait time limit for batching (T_{wait}) and the number of log disks in the array. Unless specified otherwise, the following parameter settings are

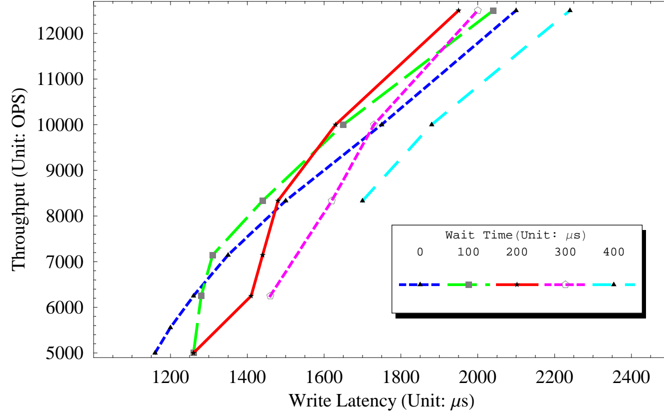


Figure 4.11: Performance impact of wait time limt (T_{wait}) on the log disks' throughput and write latency.

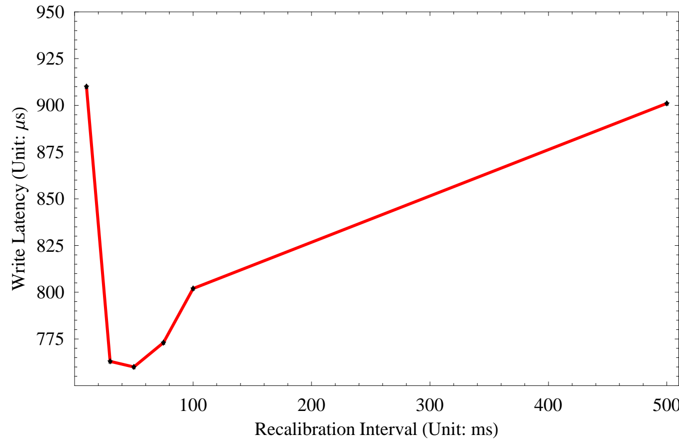


Figure 4.12: Impact of the choice of the recalibration frequency or interval on the log disks' write latency.

used by default: $T_{idle} = 50msec, T_{switch} = 12, T_{wait} = 0.2msec$, and the number of disks is 5.

We use a synthetic workload to feed into *Mariner*'s Trail node by varying the inter-request interval until reaching the maximum throughput of the log disk array. The synthetic workload contains $20,000$ write requests of 4 KBytes and there is no read request, and the write latency from the device driver is measured. We use six different inter-request interval values to generate six different input request rates: 0.08msec, 0.1msec, 0.12msec, 0.14msec, 0.16msec and 0.18msec.

T_{switch} determines when to switch a disk's head to the next track and thus plays an important role in the trade-off between disk write latency and disk space utilization efficiency. Every curve in Figure 4.9 has up to six measurements, which from left to right correspond to the six inter-request intervals in decreasing order. We stop decreasing the inter-request interval as soon as the measured latency exceeds 2msec.

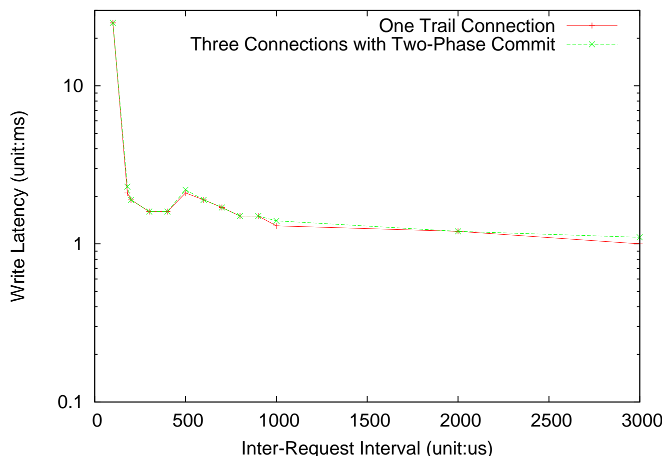


Figure 4.13: *The impact of the inter-request interval of the input write request sequence on the average write latency. Both the iSCSI initiator and target set their queue length to 2048. T_{wait} is set to 0.36 msec and T_{switch} is set to 2. One vanilla iSCSI connection with only Trail node is compared with two-phase commit implementation.*

For a given T_{switch}, as the input request rate increases (or inter-request interval decreases), each physical write batches more logical writes, and the disk utilization efficiency improves because Equation 4.2 is based on the number of physical writes and the same number of physical writes can pack more bytes because batching is more effective. However, improved disk utilization efficiency worsens the average write latency, because each physical write is larger and takes longer to complete. For a given input request rate, as T_{switch} increases, the disk utilization efficiency improves significantly without degrading the average write latency too much. This result empirically justifies one of the key design decisions in *Mariner*: allowing multiple physical disk writes per track.

Each curve in Figure 4.10 shows the latency and throughput of a given number of log disks when the inter-request interval decreases from 0.18msec to 0.08msec from left to right. For a fixed number of log disks, increase in the input request rate increases both their throughput and latency because batching is more effective and the size of each physical write is bigger. For a given input request rate, as the number of log disks increases, the throughput increases linearly and the latency remains largely unchanged. For example, when the inter-request interval is 0.1msec, the 1-disk configuration can achieve a throughput of 1000 disk writes operations per second (OPS) with an average write latency of 1.4msec, and the 2-disk configuration can can achieve a throughput of 2000 disk writes operations per second (OPS) with the same average write latency. This linear improvement comes from the fact that multiple disks can mask the disk head switch delays of individual disks as well as provide higher aggregate raw transfer bandwidth.

Again each curve in Figure 4.11 has up to six measurements, which from left

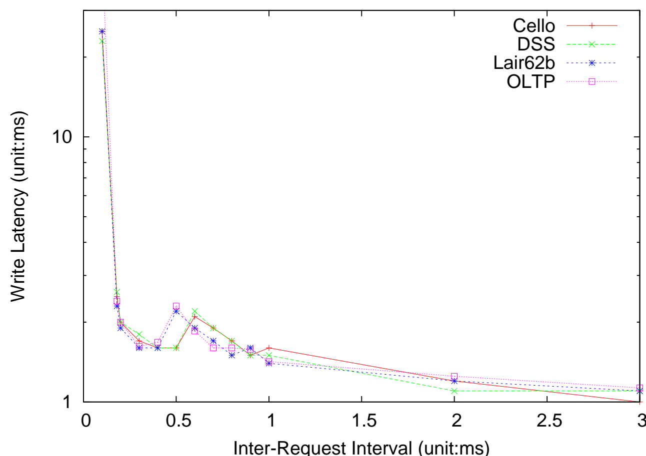


Figure 4.14: The measured end-to-end latency from a *Mariner* client under four different types of workload. The burst size is set to 1000 and inter-burst interval is set to 1 second.

to right correspond to the six inter-request intervals in decreasing order, and we stop decreasing the inter-request interval as soon as the measured latency exceeds 2msec. For a given T_{wait}, as the input request rate increases, the throughput of the log disks increases because batching is more effective, and the average write latency grows because each physical write is larger and takes longer to complete. For a given input request rate, increase in T_{wait} improves the batching efficiency, which in turn increases the throughput and the average write latency of the log disks. From the results, 0.2msec seems to be a good choice for T_{wait} to achieve a reasonable tradeoff between disk write latency and incurring reasonable write latency overhead.

Figure 4.12 shows the performance impact of the recalibration frequency on the average write latency. Increase in the recalibration frequency improves the accuracy of disk head position prediction and thus reduces the rotational latency of disk writes. However, increase in the recalibration frequency also introduces additional load to the log disks and may actually delay the disk writes requests from users. Therefore, for a given workload, there is an optimal recalibration frequency that balances these two performance factors, as shown in Figure 4.12. For a workload consisting of 4 KB large requests, a recalibration interval of 50msec is the optimal value.

4.6.5 Impact of iSCSI Processing

This subsection evaluates the write throughput and latency of *Mariner*'s Trail node as seen from an iSCSI client. The test environment contains one iSCSI connection from an iSCSI client to an iSCSI target that uses *Mariner*'s Trail node as the underlying storage device. We use a synthetic workload that keeps sending write requests of size 4KB at a fixed inter-request interval and for each run, we measure the average write latency. The queue length of both the iSCSI initiator and the iSCSI target is set to

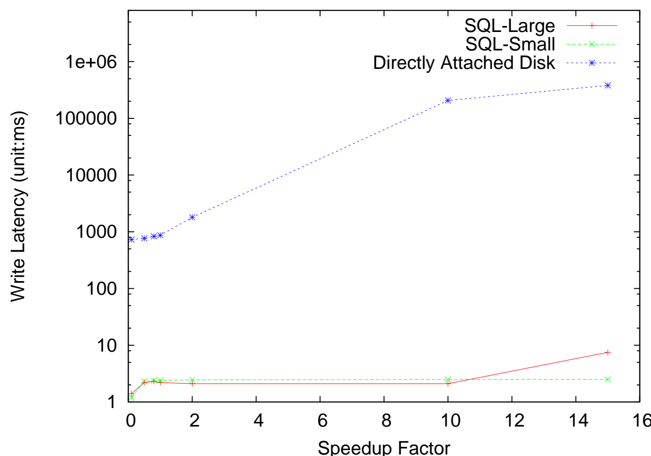


Figure 4.15: The measured end-to-end latency from a *Mariner* client under two real traces at different speedup factors. Y axis is in log scale. Both iSCSI initiator and target set their queue length to 2048. T_{wait} is set to 0.36msec and T_{switch} is 2.

2048 to accommodate large bursts.

We modify the *IET* iSCSI implementation in the following ways to improve its write latency. The first modification is to avoid going through the file-system-related APIs as used by the `fileio` mode of the *IET* implementation. Instead, we call *generic_make_request* directly, a standard interface between the block device and other components of the kernel. The second modification is to simplify the software architecture. The original *IET* iSCSI implementation has two categories of threads: a network thread and a pool of worker threads to issue requests to the underlying storage entities. These two threads relay data through an iSCSI command queue. Our implementation eliminates the iSCSI command queue and directly places write requests into the per-log-disk request queue.

Figure 4.13 shows how the iSCSI-level write latency varies with the inter-request interval. The iSCSI-level write latency increases dramatically when the inter-request interval falls below 0.18msec, because the input load corresponding to the inter-request interval of 0.18msec hits the capacity of the log disks. The IET iSCSI target implementation could process an iSCSI command every 0.08msec. The average batch size is around 8KB, which takes a fixed processing overhead of 0.1msec. The Trail implementation in the *Mariner* prototype introduces a small overhead (around 0.07msec), which comes from decision logic that determines which log disk to use, and post-processing after each physical I/O completion.

Figure 4.13 also shows that the iSCSI-level write latency increases as the inter-request interval increases from 0.3msec to 0.5msec. This is because a request is delivered to the disk controller under two scenarios: either the request's wait time exceeds T_{wait} when the next request comes in or there is no queued request and a free log disk is ready to be used. These two conditions conflict with each other: a wait time exceeding T_{wait} indicates there have been queued requests and future requests

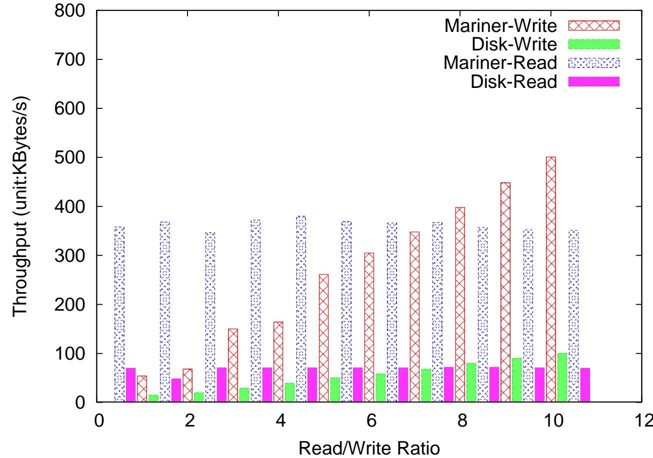


Figure 4.16: The measured Postmark throughput for directly-attached disk and *Mariner* 's storage architecture, respectively with different read/write ratio. We create 100,000 files and run 500,000 transactions. Both the iSCSI device and the locally-attached disk are synchronously mounted to the working directory of Postmark.

will get queued. Therefore, a time point exists to reach the worst case: the time just falls within T_{wait} and forces subsequent requests to believe there have been queued requests and experience the queuing delay in the same way. For T_{wait} of 0.36 msec and 5 log disks, this time point happens to be 0.5 msec. After reaching a peak value at 0.5 msec, the write latency drops down as the inter-request interval increases because input request rate is far below *Mariner* 's capacity and no request needs to be queued.

4.6.6 Impact of Modified Two-Phase Commit Protocol

In this section, we study the performance impact of the modified two-phase commit protocol on the write latency. An iSCSI client is connected to a Trail node, a master node and a local mirror node. The queue length of both the iSCSI initiator and the iSCSI target is set to 2048 to accommodate large bursts. We use both synthetic workload and real disk access traces in this experiment. The synthetic workload consists of multiple request bursts with a sufficiently long time interval between two consecutive bursts to allow the master and local mirror node to finish the previous burst. The target block address of each disk write request in the synthetic trace is borrowed from the Lair62b trace. Both the master node and local mirror node could complete a burst size of 1000 requests within 100msec when the on-disk cache is turned on. The burst size in the synthetic workload is set to 1000 and the interval between two consecutive bursts is 200msec.

Figure 4.14 shows the write latency versus the inter-request interval within a write burst. Compared the one connection case with that of modified two-phase commit, it is clear that the modified two-phase commit protocol does not introduce

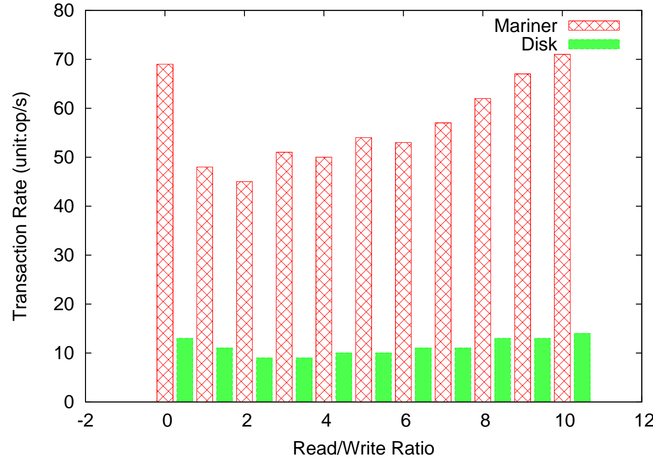


Figure 4.17: The measured Postmark transaction rates for directly-attached disk and *Mariner* 's storage architecture with different read/write ratio. For *Mariner* , both iSCSI initiator and target set their queue length to 2048. T_{wait} is set to 0.36msec and T_{switch} is 2. For directly-attached hard drive, we turn off their on-disk write cache.

any noticeable penalty on the write latency. There are two reasons. First, because of TRM, the additional payload transfer due to data replication does not incur additional networking overhead. Second, the latency of a modified two-phase commit transaction ends when the write to the Trail node is completed, which is exactly the same as the iSCSI-level write latency reported in the previous subsection. Figure 4.15 shows the measured write latency under disk access traces played back at different speedup factors. Because the MS-SQL-Large trace is collected on a server with a large buffer cache, it contains larger bursts and the average inter-request interval is small.

When the speedup factor is smaller than 1, the write latency increases because the inter-request interval increases and the additional batching delay of the current *Mariner* prototype kicks in and when the requests are sparse, the additional batching delay disappears. When the speedup factor is large, the write latency also increases because of the larger input load. For the MS-SQL-Large trace, increase in the speedup factor eventually exceeds the throughput capacity of the log disks and result in very long write latency, most of which is queuing delays at the *Mariner* client and Trail node.

To illustrate the performance improvement, we setup a vanilla storage server where writes are only propagated to one storage node consisting of a vanilla hard drive. The vanilla hard drive has their on-disk cache turned off. Storage client and server are attached locally. We use MS-SQL-Large and MS-SQL-Small trace to drive the comparison. Figure 4.15 shows the performance improvement. For all speedup factors, our scheme beats the vanilla configuration by at least a factor of 500.

Figure 4.16 illustrates that both the write and read throughput are improved by a factor of 5. This is because the throughput of Postmark is sensitive to per-request latency especially when writes are synchronous. Postmark is a single-threaded

NFS OP	Avg Elapsed Time for Disk (msec)	Avg Elapsed Time for *Mariner* (msec)
setattr	33.3	4.8
write	121	12
create	82.7	9
remove	64.5	7
rename	57.3	9.0
link	83.2	8.9
mkdir	161	13.4

Table 4.3: *Elapsed time improvement for different write-related NFS operations, where T_{switch} = 2 and T_{wait}= 0.36 msec. Both direct-attached disk and* Mariner *'s iSCSI device are mounted synchronously on the NFS server directory.*

benchmark and each synchronous operation will prevent future requests from being sent out. As a result, a reduction in the per-request elapsed time by N will lead to a N times increase in the throughput. The average per-write elapsed time is 3 msec for *Mariner* client above the file system and 20 msec for directly-attached disks.

Figure 4.17 shows the transaction rate is also improved by a factor of 5. As all writes are synchronous, a throughput improvement of 5 indicates a per-request latency improvement of 5. This is backed by the average per-write latencies of both directly-attached disks and *Mariner* 's storage system. The per-request latency on directly-attached disk is around 20 msec and the per-request latency on *Mariner* 's storage system is 3.2 msec when read/write ratio is zero. However, under this workload, the advantage of request batching can not be shown very clearly as at any point in time, there is only one outstanding write request and there is no batching.

Table 4.3 shows the per-request latency improvement for different NFS operations by playing the Lair trace with TBBT trace player. TBBT trace player is at the NFS client side. Playing one-hour trace of the whole day trace(2:00am on Oct 21st, 2001) in full speed takes only 3 seconds for *Mariner* and 21 seconds for direct-attached disk. Per-write NFS operation latency improves by a factor of between 6 and 10. This is because there are multiple outstanding requests for batching. The latency at the iSCSI layer is 2 msec and .

4.6.7 TRM Evaluation

In this experiment, data traffic is injected through the raw iSCSI device in order to by-pass buffer and file system caching. In addition, the number of outstanding SCSI commands parameter is set to 64 to ensure the pipeline of iSCSI commands is filled up during the tests [180]. We run all tests from the initiator's memory to the target's memory without involving any disk accesses,. so that we can truly stress test the performance of the storage area network.

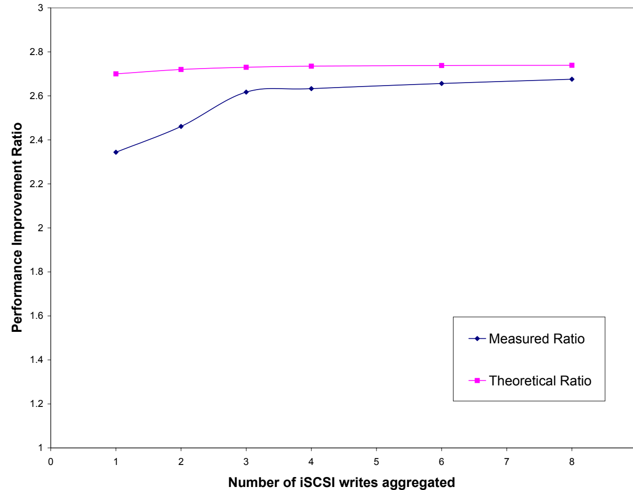


Figure 4.18: Effect of degree of packet aggregation on *TRM* throughput improvement under iSCSI. Each iSCSI write request is duplicated to two other iSCSI devices, *TRM* extracts the common payload among the TCP connections underlying these three iSCSI sessions and sends them as multicast packets. Without *TRM*, the iSCSI throughput is 768Mbps. *TRM* increases the throughput by a factor of 2.68.

During the experiment, an application transfers a file from the client node's memory to the three storage nodes' memory using the iSCSI protocol. Unless stated otherwise, the default file size used is 400Mbytes. When the connecting switch works at 100Mbps, the iSCSI write throughput without *TRM* is 93Mbps, while with *TRM*, this throughput increases by a factor of 2.7 to 251Mbps. The theoretical improvement ratio in this case is 2.72. We calculate the theoretical throughput as total bytes transfered on wire by client in non-TRM case divided by that in *TRM* case. When the connecting switch operates at 1Gbps, the iSCSI write throughput without *TRM* is 768Mbps. Because the UNH iSCSI implementation is not able to generate traffic at a rate higher than 1.08Gbps, we capture an iSCSI request trace beforehand, and replay it as fast as possible. This replay approach completely bypasses the iSCSI layer and its overhead, and thus could generate iSCSI traffic at a sufficiently high rate to stress-test the system.

Figure 4.18 shows the effect of degree of packet aggregation on *TRM* throughput under iSCSI traffic. In this case, *TRM* transmits iSCSI write requests in the TCP connection to the Trail node via multicast. It merges into a single packet headers of multiple iSCSI write requests in the TCP connection to the master storage node, and performs similar processing for packets in the TCP connection to the mirror storage node. When each iSCSI write size is 4KByte and every 8 iSCSI writes are aggregated in the TCP connection to master and local mirror storage node, the iSCSI write throughput is 768Mbps without *TRM*. *TRM* increases this throughput by a factor of 2.68 to 2.06GBit/s. The theoretical throughput improvement ratio is 2.73. We conjecture the gap between these two ratios is partly due to per-packet transmission

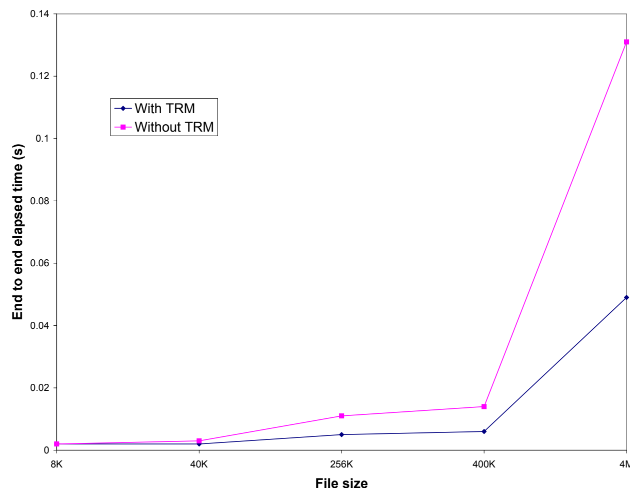


Figure 4.19: Effect of transferred file size on *TRM*'s throughput. Each iSCSI write's size is 8KByte and the degree of packet aggregation is set to 1 iSCSI Write. In each run, *TRM* starts transferring the file using the iSCSI protocol after fully warming up the TCP connections.

overhead, which is not accounted for in the calculation of theoretical throughput improvement ratio. Thus when increasing the number of iSCSI writes aggregated, the measured ratio gets close to the theoretical ratio. In addition, the gap is also due to that ACKs from the storage nodes are not merged.

Figure 4.19 shows the impact of the file size used in the experiment on *TRM*'s throughput. When the file size is more than 4MBytes, *TRM* is able to reduce the end-to-end elapsed time of transferring a file by a factor of 2.6. When the file size is 256KByte, the throughput improvement ratio decreases to 2.2. When the file size is smaller than 40KByte, *TRM* produces no visible throughput improvement because it takes at least one RTT to complete a file transfer and the RTT of the testbed network is around 0.2msec. The file size in this experiment really corresponds to the total size of a continuous stream of write requests. Because in real world storage clients play the role of a file or DBMS server, they could easily generate a stream of write requests at the transfer rate of tens of Mbps when they are fully loaded.

4.6.8 Effectiveness of BOSC

To facilitate historical snapshot access, *Mariner* supports two indexes to translate a logical disk block at a particular time point to its physical disk block: One index (TL index) is based on the time stamps at which new versions of logical disk blocks are created whereas the other (LT index) is based on the addresses of logical disk blocks. Insertions into the TL index has higher locality than those into the LT index, because time stamps are incremented monotonically. As *Mariner* uses the same logging technique described in Section 4.2, it is able to log new versions of

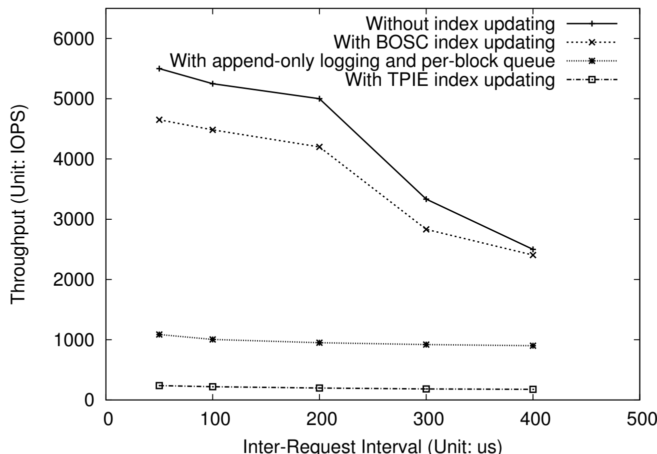


Figure 4.20: Throughput comparison among four versions of a block-level CDP system under the random insert workload: the baseline configuration without index updating, index updating using BOSC, index updating using BOSC with append-like logging, and index updating using TPIE.

updated disk blocks (4KB each) at more than 5000 blocks/sec, the main bottleneck of *Mariner* thus lies in the update of the two indexes, particularly the LT index.

To measure *Mariner*'s end-to-end block logging throughput, we ran a kernel thread on the CDP server to continuously generate new versions of existing logical disk blocks, each of which is logged to disk and triggers a new entry to be inserted into the LT and TL index. The address of each updated logical disk block is randomly generated and is uniformly distributed in $[0, 2^{61}]$. In each run, there are totally 20 GB worth of new block versions logged and 935 MB ($20GB * \frac{24}{512}$) worth of index records inserted. The buffer memory for BOSC's per-page request queuing is set to 64 MB, and the leaf index page cache for TPIE is also set to 64 MB. We varied the rate at which this kernel thread generates new block versions and measured the the number of updated blocks that *Mariner* is able to process, including logging them to disk and updating their LT/TL index entries, per second.

The throughput of *Mariner* when index updating is disabled measures its ability to log new disk block versions to disk, and thus represents an upper bound on *Mariner*'s throughput. Figure 4.20 compares the sustained throughput of *Mariner* with and without index updating. The difference between them thus represents the performance cost associated with index update. As the inter-request interval decreases, *Mariner*'s throughput increases because the input load increases. To bound the latency of each disk write, we imposed a constraint on the latency of each disk write request by setting T_{wait} to 1.6 msec. With this constraint, *Mariner* cannot sustain an input load with an inter-request interval lower than 50 μsec even when index update is disabled.

We experimented with three different ways of updating the LT index in *Mariner*: TPIE, BOSC and BOSC with append-like logging. TPIE does not support logging or sequential commit. BOSC supports both sequential commit and low-latency logging. BOSC with append-like logging supports sequential commit but uses append-like logging, which treats each disk in the log disk array as a separate log file and always writes to the end of each log file rather than where the disk head happens to be.

Compared with the baseline configuration without index updating, the index update overheads introduced by TPIE, BOSC, and BOSC with append-like logging are more than 95%, less than 15%, and more than 85%, respectively. In terms of absolute performance, the end-to-end logging throughput of a BOSC-based *Mariner* is more than 45 times better than a TPIE-based *Mariner*, and more than 6 times better than an append logging-based *Mariner*. As the inter-request interval increases, the input load decreases and the relative performance impact of index updating also decreases. As a result the throughput gap between the baseline configuration without index updating and the BOSC configuration with index updating also decreases with the increase in inter-request interval.

4.6.9 Performance Evaluation of UVFS

In this section, we evaluate the effectiveness and performance of *UVFS*'s file version searching capability. From the perspective of an end user, the ability to interactively navigate through historical versions of files enables her to quickly zoom into file versions of interest. We use the elapsed time for servicing a file versioning query as the metric for evaluating *UVFS*'s performance.

Methodology

The testbed used in this study consists of an NFS client node, an NFS server node, and a file update logging node based on an experimental block-level CDP system called Mariner [14], all of which are connected by a Netgear GS508T Gigabit Ethernet switch. The CDP server is a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a Gigabit Ethernet card, and five IBM Deskstar ATA/IDE hard disks, four of which are log disks and one of which holds data. Other testbed nodes are Dell PowerEdge SC1425 servers with an Intel 2.8 GHz CPU, a Gigabit Ethernet card, and 1024 MB memory. The operating system is Fedora Core 3 with Linux kernel 2.6.11. The test file system used is an ext3 file system unless specified otherwise. When caching of disk data blocks on the CDP server is turned on, the amount of memory dedicated to caching is 4 Mbytes. We ran the following workloads to create historical images on the CDP node, and used the resulting images to evaluate *UVFS*. All experiment runs were conducted on machines with cold cache.

Synthetic Workload: The synthetic workload starts with a file system with only an empty root directory. It creates N subdirectories under the root directory, picks one of the subdirectories, say `/a`, and creates N subdirectories under it, picks

one of the subdirectories of `/a`, say `/a/b`, and creates N subdirectories under it, and recursively applies the same set of operations until it creates a directory of a certain depth (D). At that point, the workload creates N files under one of the most recently created batch of directories (called the *leaf directory*, e.g. `/a/b/c/d/e` for $D = 5$), and for each file creates C incarnations, each of which in turn has K versions. Then it deletes all files in the leaf directories, all subdirectories in the leaf directory's parent directory, all subdirectories in the parent directory of the leaf directory's parent directory, and recursively upwards until the file system becomes an empty root directory again. So an instance of the synthetic workload is characterized by four parameters: N, D, C and K.

Lair Trace: The Lair trace is an NFS trace collected from the EECS NFS server (EECS) of Harvard University over two months [181]. The EECS trace grows by 2GB every day. We use the Trace-Based file system Benchmarking Tool [178] to replay this trace against the NFS server on the testbed.

SPECsfs: SPECsfs is a general-purpose benchmark for NFS servers [182]. It bypasses the NFS client and accesses the NFS server directly. We ran SPECsfs with 1 server process, 1 NFS client and set the operation rate at 100 OPS to age the file system image.

Three types of file versioning queries are used in this performance study: a *version search* query asking for all the versions of all the incarnations associated with a given pathname, an *incarnation search* query asking for all the incarnations associated with a given pathname, and a *directory search* query asking for all the versions of all the incarnations associated with all pathnames under a given directory. Each file versioning query is serviced by a special agent on an NFS client, which accesses historical snapshots on a block-level CDP server through an NFS server.

Correctness of File Versioning Algorithm

To verify the correctness of *UVFS*, we ran the Postmark workload and compared the versions discovered by *UVFS* with those that were derived from a comprehensive file-level update trace collected during the run. To collect this trace, we instrumented the source code of Postmark to record every file-level update operation, including *open*, *write*, *close*, *unlink*, *mkdir*, *rmdir*, etc. The file system was mounted with the *dirsync* and *sync* flag. During each run, we turned off all file caching, including Postmark's own buffering, to ensure that all file-level updates are propagated immediately down to the block level.

After a Postmark run with 1,000 files, 1,000 subdirectories and 10,000 transactions, we deduced from the resulting file-level update trace that the run produced in total 16,637 versions of 5,900 files/directories. Because the temporal resolution of the file-level update trace is millisecond, we consolidated all updates to the same file block within the same second into one update, so as to match the temporal resolution of *UVFS*. It takes *UVFS* 305 seconds to complete servicing a *directory search* query starting from the root directory, and the result it returns matches exactly with those derived from the file-level update trace.

NFS Op	getattr	lookup	readdir	access	readdirplus
Cost	1	D	N	1	N

Table 4.4: The cost of each type of NFS operation used in file version searching in terms of numbers of RPCs required.

Synthetic Workload

We ran the synthetic workload with the *dirsync* flag turned on to force to disk synchronously every file system metadata update, including Inode bitmap, Inodes, directory entries, etc. Under this configuration, *UVFS* never needs to invoke *iFSCK* because every snapshot it accesses is always file system-consistent. As a result, the main performance cost associated with snapshot access comes from NFS and iSCSI connection set-up.

The performance cost associated with file system traversal mainly comes from the set of NFS operations used in the traversal. The performance cost of each NFS operation in turn is determined by the number of associated remote procedure calls (RPC). Table 4.4 lists the number of RPCs required by each type of NFS operation used in file version searching under a synthetic workload characterized by $\langle N, D, C, K \rangle$. The actual performance overhead of each RPC is dominated by the disk accesses it requires.

Assume the iSCSI/NFS connections required by the service of a file versioning query are pre-established and therefore their set-ups incur zero cost, the query processing cost is dominated by the file system traversal cost, which in turn is determined by the number of RPCs required. Processing of a *version search* query for a file f starts with an *incarnation search* of f to find all incarnations of f, which in turn triggers a *version search* of f's parent directory. This recursive procedure continues until it reaches the root directory. Therefore, the total number of RPCs required by a version search query is thus

$$VS_{RPC} = \sum_{i=1}^{D-1} 2 * N * (i+2) + C * K * (D+2) \quad (4.4)$$

Accordingly, the per-version discovery time of a *version search* query is $\frac{VS_{RPC} * AvgRPCTime}{C * K}$.

Similarly, the total number of RPCs required by an *incarnation search* query is:

$$IS_{RPC} = \sum_{i=1}^{D-1} 2 * N * (i+2) + C * (D+2) \quad (4.5)$$

Therefore, the per-incarnation discovery time of an *incarnation search* query is $\frac{IS_{RPC} * AvgRPCTime}{C}$.

A *directory search* of a directory d first finds all versions associated with d. A *readdir* is then issued to read all directory entries of each version of d. For each file f under each version of d, a *version search* is initiated to locate all versions of f. Therefore, the total number of RPCs required by a *directory search* query is:

$$DS_{RPC} = \sum_{i=1}^{D-1} 2 * N * (i+2) + N + N * C * K * (D+2) \quad (4.6)$$

and the per-version discovery time of a *directory search* query is

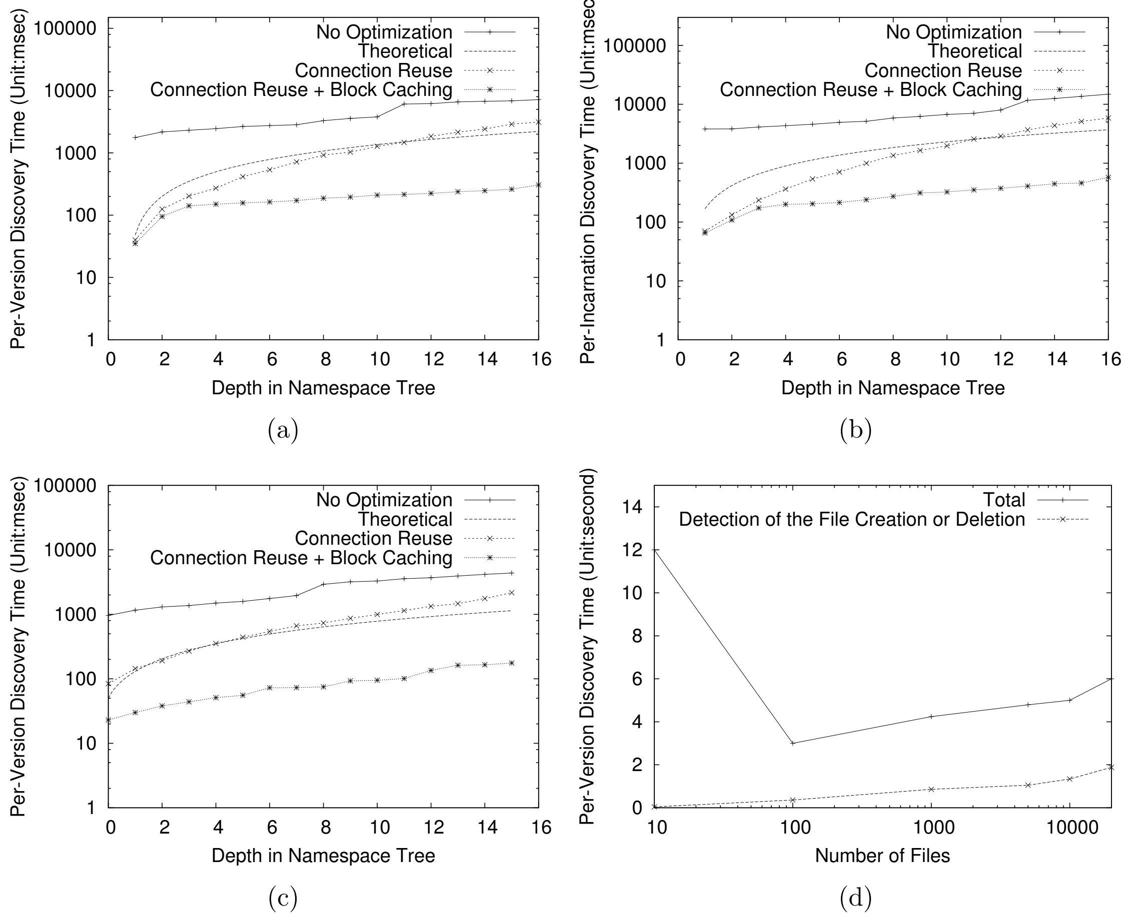


Figure 4.21: **(a)** The average per-version discovery time for a *version search* query against a file in the leaf directory when N=10, C=2, K=2, and D varied from 1 to 16. The **no optimization** curve corresponds to a vanilla *UVFS* implementation without any optimization, in which iSCSI/NFS connections are set up on demand. The **theoretical** curve is derived from Equation (4.4) with $AvgRPCTime = 16$ msec and connection reuse turned on. The **connection reuse** curve corresponds to an implementation with connection reuse optimization, in which iSCSI/NFS connections are reused and thus pre-established in most cases. The **block caching+connection reuse** curve corresponds to an implementation with both connection reuse and block caching optimizations, in which disk block caching on the CDP server is enabled. The Y axis is in log scale. **(b)** The average per-incarnation discovery time for an *incarnation search* query against a file in the leaf directory. Other parameters are the same as those in **(a)**. **(c)** The average per-version discovery time for a *directory search* query against the leaf directory when N=10, C=2, K=2, and D varied from 0 to 16. Other parameters are the same as those in **(a)**. **(d)** The average per-version discovery time for a *directory search* query against the root directory and the portion related to detection of incarnation deletion and creation when D=0, C=1, K=2 and N is varied from 10 to 20,000. No optimization is enabled. The X axis is in log scale.

$\frac{DS_{RPC} * AvgRPCTime}{N * C * K}$.

If the iSCSI/NFS connections required by the service of a file versioning query are not pre-established, the number of historical snapshots needed during a file versioning query plays an important role in the query processing cost. The numbers of snapshots required in an *incarnation search*, *version search*, and *directory search* are $2 * N * D$, $2 * N * D + C * K$, and $2 * N * D + C * K * N$, respectively. As an example, if $D = 4$, $C = 2$, $K = 2$, $N = 10$, and the target file is `/a/b/c/f1`, a *version search* needs to examine all 20 versions of `/` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a`, all 20 versions of `/a` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a/b`, all 20 versions of `/a/b` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a/b/c`, and all 40 versions of `/a/b/c` (20 files creations and 20 file deletions) to determine that there are two incarnations of `/a/b/c/f1`, from which it then locates the two versions of each incarnation based on its *last modify time*. In total, it needs to set up 100 (20 + 20 + 20 + 40) snapshots.

We set $N = 10, C = 2, K = 2$, and varied the file system tree depth parameter D from 0 to 16 for the *directory search* query, or from 1 to 16 for the *version search* query and the *incarnation search* query. Figure 4.21(a) shows the average per-version discovery time of a *version search* query whose target is a randomly selected file f in the leaf directory. In each run, four file versions are returned to each *version search* query. Figure 4.21(b) shows that the average per-incarnation discovery time of an *incarnation search* query whose target is a randomly selected file in the leaf directory. In each run, two incarnations are returned to each *incarnation search* query. Figure 4.21(c) shows the average per-version discovery time of a *directory search* query whose target is a randomly selected leaf directory. In each run, 40 file versions are returned to each *directory search* query. There are four curves in each figure, corresponding to the theoretical performance cost when NFS/iSCSI connections are pre-established, the measured performance cost when NFS/iSCSI connections are pre-established, the measured performance cost when NFS/iSCSI connections are set up on demand, and the measured performance cost when NFS/iSCSI connections are pre-established and disk block caching on the CDP server is turned on.

Regardless of whether optimizations are enabled, the per-version or per-incarnation discovery time generally increases with the tree depth of the target file, because pathname lookup cost is a significant component, and NFS decomposes the lookup of a pathname of length N into N individual lookups, each of which takes roughly the same amount of time assuming there is no client-side caching. As shown in Figure 4.21(a) and (b), the average per-version discovery time of a *version search* is smaller than the average per-incarnation discovery time of an *incarnation search* because an *incarnation search* of a file involves a *version search* of the file's parent directory. However, the per-version discovery time of a *version search* is not necessarily longer or shorter than the per-incarnation discovery time of an *incarnation search* against the same target file system object. For example, if a file `/a/b` is created at $T1$ and deleted at $T2$ with no intermediate version, a version search and an *incarnation search* of `/a/b`

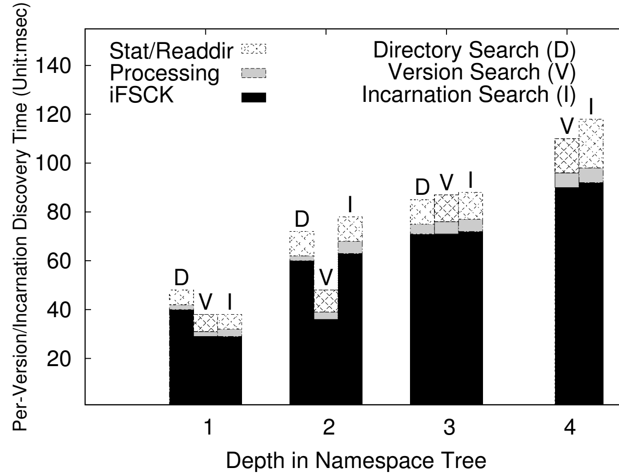


Figure 4.22: The average per-version or per-incarnation discovery time and its breakdown into the Stat/Readdir, internal processing and *iFSCK* components for an *incarnation search*, *version search* and *directory search* against file system snapshots generated by SPECsfs, with the target object's depth in the file system namespace varied from 1 to 4.Because no directory with a namespace depth of 4 exists, there is no measurement for *directory search* with $D = 4$.

between $T1$ and $T2$ take the same amount of time. However, if multiple versions of `/a/b` are created between $T1$ and $T2$ and the cost of discovering a new version by invoking *stat* is larger than that to discover an incarnation, the per-version discovery time of a *version search* is larger than the per-incarnation discovery time of an *incarnation search* in this case. On the other hand, if the number of files under `/a` is very large (e.g. 8,000), the cost to discover an incarnation is larger than the cost to discover a new version, and the per-version discovery time of a *version search* is going to be smaller than the per-incarnation discovery time of an *incarnation search*
.

In the calculation of the theoretical curve, we set $AvgRPCTime$ to a constant, 16 msec. However, in practice $AvgRPCTime$ varies with D. Larger D tends to increase the RPC cost due to less locality in data accesses during RPC processing. Thereafter, the theoretical curve does not always fit perfectly with the corresponding empirical results.

Being able to reuse NFS and iSCSI connections makes a big difference on the response time of the file version search queries, with the impact ranging from two orders of magnitude to a factor of 2 across all D values. If iSCSI/NFS connections need to be set up on demand, a large fixed overhead due to iSCSI/NFS connection set-up is added to the per-version or per-incarnation discovery time. That's why the slope of the curve for the no-connection-reuse case tends to be smaller or flatter than that for the connection-reuse case. Finally, iSCSI connection set-up takes much longer than NFS connection set-up, and one iSCSI connection set-up can support up to 255 snapshot accesses. Therefore, whenever servicing a file version search query needs more than 255 snapshots, an additional iSCSI connection set-up overhead will show up, for example, between D=10 and D=11 in Figure 4.21(a), between D=12

Configuration	Number of Distinct File and Directories	Versions Discovered	Elapsed Time (Unit: msec)
Ext3+CDP+UVFS	28,693	29,674	849,378
Ext3cow - 60s	28,693	29,456	13,892
Ext3cow - 5s	28,693	29,667	14,415

Table 4.5: Results of finding all file versions of a file system aged by the Lair trace using $Ext3 + CDP + UVFS$, and using $Ext3cow$ with two snapshot frequencies, 5 and 60 seconds.

and D=13 in Figure 4.21(b), and between D=7 and D=8 in Figure 4.21(c).

When the CDP server turns on disk block caching, the per-version discovery time of a file version search query is further reduced by a factor of 2 to 9, beyond what can be achieved with the network connection reuse optimization when $D = 16$. Specifically, this caching drastically cuts down the number of disk accesses associated with *readdir* and *lookup* operations in file version query processing. Accordingly, the effectiveness of this caching technique increases with D because the relative weight of file system traversal cost in a file version search query increases with D. In contrast, the effectiveness of the network connection reuse optimization decreases with D because the relative weight of network connection set-up cost in a file version search query decreases with D.

Figure 4.21(d) shows the average per-version discovery time of a *directory search* query and the portion of it related to detection of incarnation deletion and creation with $D = 0, C = 1, K = 2$ and N varied from 10 to 20,000. We modified the synthetic workload so that consecutive creations or deletions of files occur one immediately after another. When servicing a *directory search* query, *UVFS* needs to set up snapshot images and compare contents of the adjacent versions of a directory to detect creation or deletion of a file incarnation inside that directory. Because the current *UVFS* implementation organizes each directory's content as an ordered list of file pathnames, it takes $O(NlogN)$ pathname comparisons to detect creation and deletion of file incarnations in a directory, where N is the number of files in the directory. The lower curve in figure 4.21(d), which corresponds to the average time required to detect changes in the directory's content indeed increases in a logarithmic fashion. When N is greater than 100, the directory content comparison time is a significant component and the average per-version discovery time of a *directory search* query increases with N. However, when N is less than 100, the directory content comparison time is negligible and the time needed to set up snapshots dominates, and the average per-version discovery time decreases when N is increased from 10 to 100 because the snapshot set-up cost is amortized over a larger number of versions as N increases.

SPECsfs

We use the default setting of the SPECsfs benchmark: 12% write requests with the rest as read requests. A SPECsfs run creates a directory CL_i for the ith client, and another directory named *validatedir* for validation purpose. CL_i has in total N (the

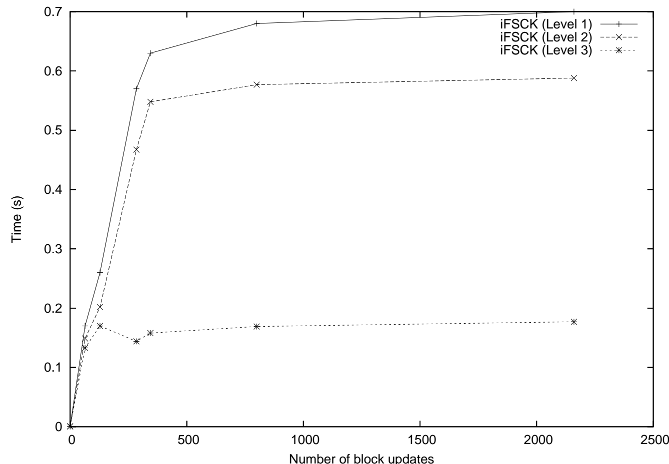


Figure 4.23: Elapsed time comparison of *iFSCK* operating under three different consistency levels when the number of disk block updates in the CDP node's log between $[T - LB, T + UB]$ is varied, where T is the user-specified target timestamp and both LB and UB are set to 40 seconds.

number of runs) $testdir_j$ directories, which hold the generated directories and files. For each test run, the corresponding $testdir_j$ contains up to 700 files/subdirectories. Processing a *directory search* query against the $testdir_0$ directory requires setting up 365 snapshots and only 10 of them need to invoke *iFSCK*.

We measured the per-version and per-incarnation discovery time for *version search*, *incarnation search* and *directory search* by issuing these queries against all files or directories at a particular file system name-space tree depth and computed the average of them. Figure 4.22 shows the per-version and per-incarnation discovery times of the three queries and their breakdowns under the SPECsfs benchmark. They are similar to those under the synthetic workload because SPECsfs creates only a small number of files or subdirectories and the name space hierarchy it creates is quite regular.

Comparison with Ext3cow

Ext3cow [135] is a file system that provides its users with file versioning, snapshotting and a time-shifting interface to navigate through the file versions. In this section, we compare Ext3cow with a vanilla Ext3 file system coupled with *UVFS* on top of a block-level CDP, in terms of the accuracy and performance of file version query processing. We used the Lair trace to age the test file system and measured the elapsed time required to locate all file versions under the root directory for each of the three configurations: Ext3cow on a local file system with the snapshotting frequency set to 5 seconds, Ext3cow on a local file system with the snapshotting frequency set to 60 seconds, and Ext3 on a local file system backed by *UVFS*, an NFS server and a block-level CDP server, with both NFS/iSCSI connection reuse and disk

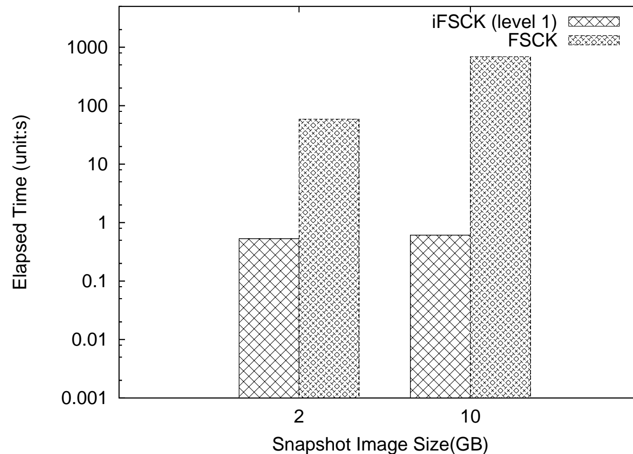


Figure 4.24: Elapsed time comparison between *iFSCK* and vanilla FSCK for an ext2 file system with different snapshot image sizes. When the snapshot image size is 10GB, *iFSCK*(level=1) is about three orders of magnitude faster than vanilla FSCK.

block caching turned on. The Lair trace used in this experiment is a one-hour trace starting from 10am on Oct 21, 2001. For Ext3cow, we use the time-shifting interface to retrieve all file versions under the root directory, starting from a cold file system.

Table 4.5 shows that $Ext3 + CDP + UVFS$ finds 0.7% more file versions than $Ext3cow$ with a 60-second snapshotting frequency, and only 7 more file versions than $Ext3cow$ with a 5-second snapshotting frequency. This result shows that CDP plus $UVFS$ indeed can capture some file versions that are missed by periodic snapshotting systems such as Ext3cow, although the marginal value of these missed versions depends on user and application requirements. As for performance overhead, $Ext3 + CDP + UVFS$ needs to set up 372 snapshots to find all file versions, and as a result is 60 times slower than $Ext3cow$. This performance difference is attributed to two factors. First, $Ext3 + CDP + UVFS$ involves three parties over the network whereas $Ext3cow$ only requires local processing. Second and more importantly, $Ext3 + CDP + UVFS$ does not require any modification to the host file system, whereas $Ext3cow$ builds file versioning directly into the file system itself.

4.6.10 Performance Evaluation of iFSCK

Evaluation Methodology

In this section, we evaluate the effectiveness and performance of *iFSCK*'s incremental file system check. For *iFSCK*, we use the number of blocks in a point-in-time snapshot that need to be mended to make it file system-consistent as the evaluation metric for its efficiency. In addition to performance overhead, we also verify the correctness of *iFSCK* by comparing their results with the ground truth.

The testbed used in this study consists of an NFS client node, an NFS server

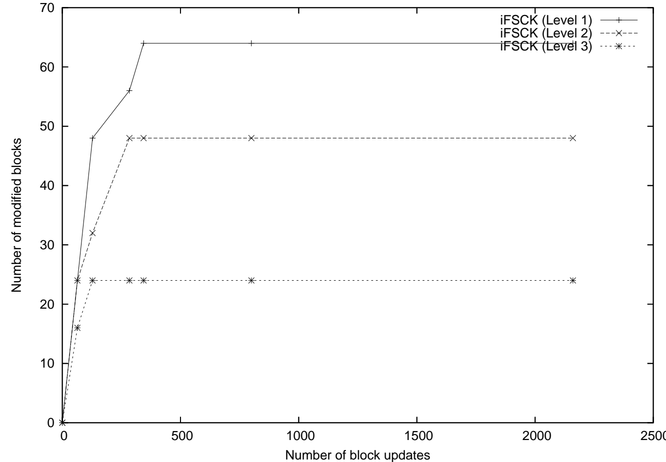


Figure 4.25: Number of file system metadata blocks *iFSCK* modifies to support different file system consistency levels when the number of disk block updates in the CDP node's log between $[T - LB, T + UB]$ is varied.

node, and a file update logging node based on a block-level CDP implementation called Mariner, all of which are connected by a Netgear GS508T 8-port Gigabit Ethernet switch. All these nodes are a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a 400 MHz front-side bus, an embedded Gigabit Ethernet Card, and up to five ATA/IDE hard disks, each of which is a 80-GB IBM Deskstar DTLA-307030 disk. The operating system is Fedora Core 3 with Linux kernel 2.6.11. The file system is an ext2 file system unless specified otherwise. We run the following four sets of workloads to create historical images on the CDP node, and use these images to evaluate *iFSCK*. All experiment runs are conducted on machines with cold cache.

- Synthetic Workload: The same synthetic workload as that is used in subsection 4.6.9.
- Postmark: Postmark [177] is a file system benchmark that emulates a heavy small file workload. The benchmark creates a specified number of files/subdirectories, performs various file update operations on them and eventually deletes all of them. The read/write ratio is always set to 1, but the number of transactions and other parameters are different from run to run.
- Lair trace: The same Lair trace as that is used in subsection 4.6.9.
- SPECsfs: The same SPECsfs workload as that is used in subsection 4.6.9.

Correctness Evaluation of *iFSCK*

We evaluate the correctness of *iFSCK* by comparing the restored result from *iFSCK*(level=1) and from ext3-FSCK for a set of snapshot images with an ext3 file system. *iFSCK* supports three file system consistency levels, which are useful for

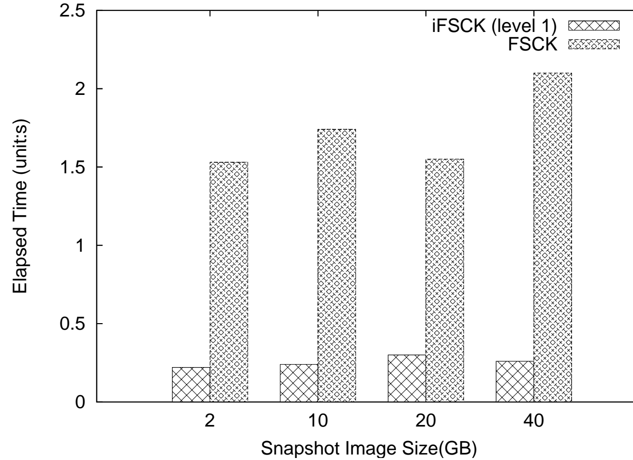


Figure 4.26: Elapsed time comparison between *iFSCK* and vanilla FSCK for an ext3 file system with different snapshot image size.

different applications. The first level is the strongest, and is designed to support read-write snapshots that span the entire file system. The second level is slightly weaker, and is designed to support read-only snapshots that span the entire file system. The third level is the weakest, and is designed to support read-only snapshots for a particular file or directory.

We ran the Postmark workload for 1,426 seconds with the following parameters: 1,000 files, 1,000 subdirectories, 1,000 transactions, and the inter-transaction interval set to 1 second. For the populated file system, we used UVFS [183] to discover all file/directory versions during the run. *UVFS* set up 650 snapshots in the process, 242 snapshot images are not file system-consistent and need to be "fixed". We invoked *iFSCK* (level = 1) against these 242 snapshot images to restore them to a consistent state. After that, we ran the vanilla ext3-FSCK against *iFSCK*'s restored images to determine if there is any residual inconsistency. None of these restored images require any additional fixes from ext3-FSCK. Moreover, all the files/directories within those images could be correctly read. This experiment demonstrates that *iFSCK*(level=1) indeed achieves the same file system consistency level as that of standard file system checkers.

Synthetic Workload Experiment

We evaluate both ext2 and ext3 file systems for the synthetic workload. Figure 4.23 shows the elapsed time of *iFSCK* when operating under these three consistency levels as the number of disk block updates appearing within $[T - LB, T + UB]$ is varied from 0 to 2500, where T is the user-specified target timestamp and both LB and UB are set to 40 seconds. As expected, the stronger the consistency level is, the more time-consuming the corresponding *iFSCK* version is.

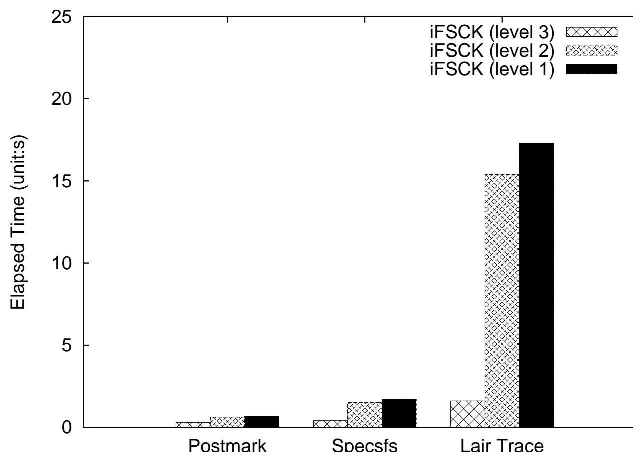


Figure 4.27: Elapsed time of *iFSCK* when operating at different file system consistency levels for the Postmark, SPECsfs and the Lair trace.

iFSCK(level=3) takes about 150 msec to check a point-in-time snapshot regardless of the number of disk block updates within $[T - LB, T + UB]$, because it only focuses on a particular directory and the number of disk block updates during the window that are related to that directory remains virtually independent of the file-level update rate of the synthetic workload. The synthetic workload always creates the same number of subdirectories inside a parent directory. On the other hand, the elapsed time of both *iFSCK*(level=1) and *iFSCK*(level=2) increases proportionally to the number of block updates within $[T - LB, T + UB]$ when it is smaller than 300. As the number of disk block updates grows beyond 300, their elapsed times levels off, because the same block gets updated multiple times and for each block *iFSCK* only needs to examine the latest update before T and the oldest update after T. The performance difference between *iFSCK*(level=2) and *iFSCK*(level=3) originates from the fact that their scopes of consistency maintenance are different, a directory vs. the entire file system. On the other hand, the performance difference between *iFSCK*(level=1) and *iFSCK*(level=2) is attributed to a difference in the number of file system metadata checks they perform, e.g. *iFSCK*(level=1) checks the consistency of Inode and Block bitmaps while *iFSCK*(level=2) does not.

Figure 4.25 shows the number of file system metadata blocks that *iFSCK* modifies to support different file system consistency levels as the number of disk block updates within $[T - LB, T + UB]$ is varied from 0 to 2500, and correlates very well with Figure 4.23. This list of metadata block updates correspond to the redo list described in Section 4.4.2.

Because *iFSCK* only needs to focus on a small number of file system metadata block updates around the snapshot timestamp, the number of disk reads and writes it incurs is much smaller than a vanilla file system check tool. Figure 4.24 shows the elapsed time comparison between *iFSCK*(level=1) and ext2's file system checker for two snapshot image sizes, 2GB and 10GB. The elapsed time of *iFSCK*(level=1)

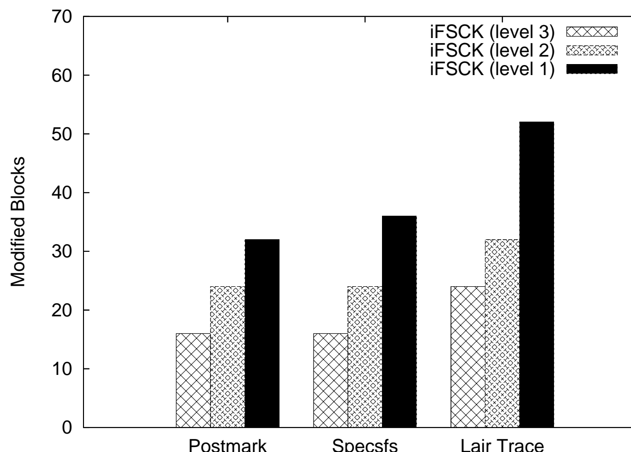


Figure 4.28: Number of file system metadata blocks *iFSCK* needs to modify to support different file system consistency levels for the Postmark, SPECsfs and the Lair trace.

remains virtually the same when the snapshot image size increases from 2GB to 10GB, because the number of metadata block updates around the snapshot timestamp is not affected by the snapshot image size. In contrast, the elapsed time of ext2's file system checker grows proportionally with the snapshot image size because it needs to scan the entire image. When the snapshot image size is 10GB, *iFSCK*(level=1) takes only 0.61 second, which is more than 1000 times faster than that of ext2's file system checker.

Figure 4.26 shows the elapsed time comparison between *iFSCK*(level=1) and ext3's file system checker for snapshot images of different sizes. By leveraging the metadata journal, which is similar in functionality to *iFSCK*'s redo list, ext3's file system checker can complete a file system check transaction using roughly the same amount of time regardless of the snapshot image size, as is the case of *iFSCK*. In all cases, *iFSCK* still outperforms ext3's native file system check, because the latter checks more global metadata than *iFSCK*, such as total Inode count, total block count, etc.

Standard Benchmark and Real Workload Experiments

Figure 4.27 shows the elapsed time of *iFSCK* when operating under different file system consistency levels for the Postmark and SPECsfs benchmarks and the Lair trace. *iFSCK* takes less time under the Postmark benchmark than under the SPECsfs benchmark because Postmark's disk write pattern is sparser than SPECsfs's. Unlike Postmark and SPECsfs, disk writes in the Lair trace are quite bursty in some short periods and become very sparse for the rest of the trace. Therefore, for snapshots created in the sparse-write periods, *iFSCK* returns almost immediately (less than 10 msec) because there are very few writes (in most cases, it is 0) in $[T - LB, T + UB]$.

However, for snapshots created in the bursty write periods, *iFSCK*(level=1) takes more than 17 seconds to complete, whereas vanilla FSCK takes more than an hour to do the same. In this test, *iFSCK* needs to examine 35,000 disk block updates, which appear in the 80-second interval of $[T - LB, T + UB]$, and 8878 directories.

Figure 4.28 shows the number of file system metadata blocks *iFSCK* needs to modify under these workloads. For most runs, *iFSCK* needs to modify fewer than 56 blocks to bring a snapshot image to the strongest file system consistency level. The fact that the numbers of blocks modified under these three workloads are roughly comparable suggests most of the performance overhead of *iFSCK* when running under the Lair trace comes from the need to extract a small number of file system metadata updates from a large number of disk block updates in $[T - LB, T + UB]$.

Chapter 5

Random Write Optimization for SSD

5.1 Design

5.1.1 Overview

LFSM is a storage manager that sits between a file system and a flash disk's native driver, and can be considered as an auxiliary driver specifically designed to optimize the random write performance for existing flash disks in a disk-independent way. A property shared by all commodity flash disks on the market is good sustained throughput for sequential writes, between 30-60 MB/sec. Accordingly, the key idea in LFSM is to convert random writes into sequential writes so as to eliminate random writes from the workload that a flash disk physically faces by construction. To perform such conversion, LFSM provides a separate logical disk address space to the file system and other higher-layer software, implements it using multiple logs of erasure units (EU), and turns every incoming write to the linear address space into a physical write to the end of one of these logs. Because writes to each log are sequential in nature, their performance is close to the flash disk's raw sequential write performance.

To reclaim unused space on the logs, LFSM performs garbage collection in the background, whose associated performance impact could be substantial if not carefully managed [79]. The performance cost of reclaiming an erasure unit mainly comes from copying out the *live* physical blocks in it and is thus proportional to the number of such blocks at the time of reclamation. A major design decision in LFSM is its use of multiple logs rather than one log to minimize the performance overhead associated with garbage collection. More concretely, LFSM estimates the life time of each logical block, which is the time between two consecutive writes to it, and maps logical blocks with a different life time range to a different log. Moreover, LFSM manages each log as a circular FIFO queue to simplify the implementation complexity of garbage

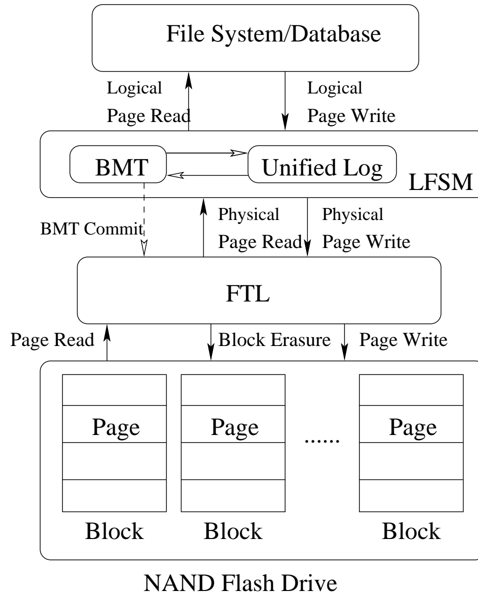


Figure 5.1: *LFSM resides on top of FTL, and issues two sequential write streams to FTL, one updating the BMT and the other updating the data logs. The dotted line indicates the corresponding write stream is asynchronous. The unified logging module queries the BMT to map a logical block into its corresponding physical block, and updates the BMT whenever a new write is logged.*

collection and recovery logic.

Because a logical block is re-assigned to a different physical block every time it is overwritten, LFSM needs to maintain a block map table (BMT) that associates each logical block with its current physical block, and changes a logical block's map entry whenever it is overwritten. That is, although LFSM converts every logical block write operation into a sequential physical block write operation, along the way it requires a random write to update the logical block's associated BMT entry. Moreover each random write to the BMT needs to be done synchronously to guarantee the consistency and integrity of the BMT. As a result, these random synchronous writes to the BMT becomes a new performance bottleneck. LFSM solves this problem by first synchronously logging these BMT writes so that they are recoverable, and then committing them to the BMT asynchronously using sequential writes. This scheme, known as BOSC, greatly decreases the performance cost associated with BMT writes because it turns them into asynchronous sequential writes without compromising the BMT's integrity and consistency.

5.1.2 Disk Write Logging

A flash disk exposes a linear sequence of disk blocks, e.g., 0–8M blocks for a 32-GB flash drive with a 4-KB block, which the file system and/or user-level applications use for allocation and read/write. Given such a flash disk, LFSM reserves a portion of the address space, about 0.6%, to store metadata (e.g. BMT), and exposes the

rest to the file system and/or user applications.

As shown in Figure 5.1, LFSM sits below the file system and above the flash disk's native driver. Accordingly, there are three address spaces in this design. The file system and/or user applications see a linear sequence of *logical* blocks exposed by LFSM. The native flash disk driver exposes a linear sequence of *physical blocks* to LFSM, and the actual flash disk hardware exposes a linear sequence of *real blocks* to the native flash disk driver and the on-disk firmware. Many classical optimizations for flash disks, such as *asynchronous pre-erasing*, which hides the performance overhead associated with erasure operations by performing them in the background, and *wear leveling*, which ensures that every real block on a flash disk is written approximately the same number of times during the disk's life time, are already built into the native flash disk driver and firmware and thus not the focus of LFSM. These optimizations require a translation between physical blocks and real blocks so that consecutive physical blocks are not necessarily mapped to adjacent real blocks.

Although the exact mapping between physical and real blocks is implemented in the proprietary driver/firmware and is thus unknown, we know all existing flash disks share one property: their sequential write performance is much better than their random write performance, sometimes by more than an order of magnitude. This motivates the entire design of LFSM: converting all writes in the input workload into sequential writes.

Whenever a logical block is written, LFSM allocates a new physical block from a data log to hold the new version of the logical block, writes to the allocated physical block, and updates the logical block's BMT entry to point to the physical block. With this approach, every random write from the file system is converted into two writes, one to the data log, which is sequential by construction, and the other to the BMT, which is random in general.

Logging BMT updates entails a space overhead problem. Because the minimum unit for reading and writing a flash disk is a 512-byte sector, each BMT update log record costs a 512-byte sector even though in actuality it requires no more than 20 bytes. This means the space overhead associated with BMT logging is about 12.5% (512 bytes for every 4-KB page), which is too high to be acceptable. Given a write to a logical block B, LFSM solves this problem by compressing the new version of B to squeeze out a 20-byte area that can be used to hold the log record for the logical block write's associated BMT update. This compression step increases the performance overhead of every logical block write operation, but only slightly as shown later, because empirically we found that it is possible to squeeze out 20 bytes by compressing the first 512 bytes of every 4KB page in most cases. If compressing the first 512 bytes of a 4-KB page cannot produce 20 bytes of spare space, LFSM uses a 512-byte sector to hold the log record of the associated BMT update log record.

5.1.3 Ensuring Metadata Consistency

To ensure the consistency and reduce the performance overhead of each BMT write, LFSM applies a BOSC scheme [21] to synchronously log the effect of each BMT update and asynchronously commits multiple updates to the BMT in a batched fashion. More specifically, LFSM organizes the BMT into a set of 4-KB pages and allocates an in-memory request queue for each BMT page. For each BMT update, LFSM logs its parameters and inserts it to the request queue associated with the page to which the update's target BMT entry belongs. In the background, LFSM fetches from disk each BMT page whose in-memory request queue is not empty, commits all pending updates associated with each fetched page, writes the BMT page back to disk, and moves on to the next BMT page, etc., until it traverses the entire BMT. LFSM repeats the same cycle constantly. In this process of committing BMT updates to disk, writes to the BMT are largely sequential.

Because of BMT update logging, even if the system crashes, the BMT updates that have not been flushed to the on-disk BMT can be correctly reconstructed at recovery time. Through BOSC's asynchronous batching, the performance overhead of the random writes to the BMT is reduced to the minimum, because these updates are aggregated and completed through asynchronous sequential writes. Using BOSC to update the BMT means each logical block write operation triggers three write operations, the first being writing a new version of the logical block to a data log, the second being logging the associated BMT update, and the third being actually updating the corresponding on-disk BMT entry. The first two writes are done synchronously and the third write is done in an asynchronously and batched fashion. LFSM combines the first two logging writes into a single physical write by merging the log records for a logical block write and its associated BMT update. Therefore, LFSM performs at most two physical writes for every logical block write operation, with the average cost of the second write being a fraction of that of a typical physical write because of the use of BOSC.

BMT update logging ensures that uncommitted BMT updates can be correctly recovered when the system crashes, and thus makes it possible to commit pending BMT updates in an efficient manner without compromising the BMT's integrity. When a system crashes, LFSM reads into memory the data log to identify the set of BMT updates that have been logged but have not yet been committed to the on-disk BMT. To facilitate the identification of not-yet-committed BMT updates, LFSM includes the following information in the BMT update log record associated with each logical block write operation: (1) the write's target *logical block number*, (2) the write's corresponding *sequence number*, (3) the *commit point*: the sequence number of the youngest logical block write operation all BMT updates before which have already been committed to disk at the time of logging. With these information, LFSM reconstructs pending BMT updates by first identifying the latest or youngest BMT log entry, whose sequence number is, say $N1$, then obtaining its associated commit point, whose sequence number is, say $N2$), and finally reading in all the BMT update log records between $N1$ and $N2$ to insert them into their corresponding

per-BMT-page request queues.

With the above design, LFSM successfully services each logical block write operation with a single synchronous physical write to the data log, and thus greatly improves the random write performance of commodity flash disks. However, LFSM's design entails two background activities, which can potentially slow down a flash disk's overall performance. First, pending BMT updates need to be committed to the on-disk BMT in the background. Second, because LFSM treats the entire flash disk as a log, it needs a background garbage collector to reclaim free blocks and make them available to LFSM for servicing subsequent logical block write operations.

5.1.4 Garbage Collection

If we see the processing of a WRITE request by LFSM, we would understand that we always log the WRITES sequentially. It is noteworthy to observe that this sequential nature is with respect to one erasure unit, i.e., WRITES are sequential inside one EU. Though many of the blocks get overwritten with time, we cannot immediately overwrite them since it would break the sequential write property of LFSM. Instead, we keep logging sequentially using the free blocks and mark the old over-written blocks as invalid. Thus, with time the number of invalid blocks increases and proportionally the number of free blocks decreases. Hence, to clean up the invalid blocks and make them re-usable (free), we do garbage collection (GC). The goal of GC is to maintain the balance between the invalid and free blocks. GC in LFSM is always done in the background.

Our garbage collection is EU-based. In other words, we collect valid blocks of one EU completely and move this EU to free pool and then proceed to do the same for another EU. We have a threshold (number of free EUs to the total number of EUs) to trigger the GC. Currently, it is 20%. When this threshold is hit, GC starts in the background. Due to various reasons like scheduling, heavy IO in main thread, etc., there might be a case where the background GC might not be able to pump up the free pool and hence the main thread cannot find any free EU to process its WRITE. In this scenario, the main thread yields to the background thread (to do the GC) and waits till it finds at least one free EU in the free pool. This we call critical garbage collection.

A good GC algorithm should have these features:

1. Minimize the number of valid pages copied. (Utilization)
2. The frequency of garbage collecting an EU should be proportional to the frequency of invalidation of blocks inside this EU (Temperature).

LFSM satisfies the above mentioned criteria in a novel way as explained below.

Utilization

Utilization represents how many valid pages are there in a EU. So, when we want to garbage collect an EU, we pick the EU which has the least utilization so that we have

to copy the minimum number of valid pages. After copying all the valid pages in the EU to new location, we move the old EU to the free pool. The data structure which is most efficient for this purpose is $Min-Heap$, which we call LVP_Heap, where the root has the EU with the least number of valid pages. In log N time, we can insert and delete a EU from this heap. One important thing here is that we do not want to GC a EU whose pages are still being over-written (invalidated) currently. Because, this would result in copying those pages which would immediately get invalidated. Hence, we implemented something called a HList. Once a EU which is in heap gets invalidated (a page gets over-written), this EU is moved from heap to HList. It is kept in HList until HLists length limit is reached. This limit is represented by $HLIST_CAPACITY$. This would give substantial time for the EU in HList to get invalidated as much as possible and by the time it gets moved to heap, it would be relatively inactive with respect to getting invalidated. In other words, we want to keep waiting a EU in HList to move to Heap until its utilization gets stabilized. Now, it can be safely garbage collected. Note that, if a EU in HList gets invalidated, it is moved to the head of the HList and when HList limit is reached, the EU from the tail of the HList is removed and inserted to Heap.

Temperature

Temperature denotes the frequency of invalidation of the blocks. We assign the blocks which get frequently over-written as HOT, the ones which are relatively stable as WARM, and the ones which are almost never over-written as COLD. For example, DLL files could be termed COLD, while TEMP files are treated as HOT. The idea is to group the blocks having the same temperature in the same EU. The assumption is that blocks having the same temperature generally die (invalidated) together. Hence, it makes sense not to garbage collect those EUs which have cold blocks as frequently as those EUs having warm blocks. Similarly, the EUs having warm blocks are garbage collected less number of times when compared to those having hot blocks. This will avoid the number of EUs that are actually garbage collected and also garbage collect those EUs which would give us more free blocks. Hence this improves the efficiency of the process. When a block is written for the first time, by default it goes to a cold EU. Once it gets over-written, it is moved to warm EU. If its again over-written, it is moved to hot EU and stays there for any further invalidation. Similarly, if a hot block survives (remains valid in the EU) a GC once, it is moved to warm EU. If it survives the 2nd GC, it is moved to cold EU and it stays there after any further GCs.

We keep the information regarding the utilization & temperature of every EU in its respective structure. Since garbage collection and main thread WRITES run concurrently, there might be a possibility of conflicts, i.e., both targeting the same LBN. For example, a GC WRITE finds out that the LBN it's moving is already in the Active List. This means that particular EU having this LBN is being invalidated and hence would be moved to HList and should not be garbage collected. Hence, we should abort the garbage collection of this EU.

Because the garbage collection is based on both the utilization and temporal

locality of EUs, we call it Locality-Preserving-Utilization-based Garbage Collection (LPU-GC). It is noteworthy that when the length limit of HList is 0, LPU-GC is the greedy garbage collection based purely on the utilization of EUs.

Procedure

As explained earlier, the main goal of GC is to pump up the pool of free EUs. We would try and garbage collect enough EUs so that effectively we would generate at least one EU worth of free space. So we try to pick EUs from the heap one after another until we find out that garbage collecting these EUs would give us one EU worth free space. If we find that EUs in heap are not enough to satisfy our constraint, we pick from HList.

The information regarding the LBN of all the blocks in the EU is kept in a sector called metadata sector. This sector resides in the last block (8 sectors) of the EU. So, we will read this sector now and get the information of which LBNs are present and also how many of those are still valid using the EU bitmap. If we find any conflict with main thread WRITE, we would stop the GC of that EU and proceed to the next. Next, we will read the entire EU. After having the content of the EU in memory, we would move the EU to Free List from Heap/HList depending on the present location of the EU. Next step would be to assign new PBN to these blocks where they would end up eventually. Then do the actual I/O to copy valid blocks to their new destination EUs. Repeat this process for all the EUs in the pickup list.

5.2 Prototype Implementation

To read or write a logical block, the LFSM needs to perform a BMT look-up to identify the corresponding physical block. Because the BMT is typically too large to fit into main memory, it is only available from the flash disk. This means that every logical block read or write may incur an additional read disk access to the BMT. One way to mitigate this performance overhead is to cache recently used BMT entries. In some sense, the per-BMT-page request queues holding pending BMT updates serve as such a cache, because LFSM needs to consult these queues before accessing the on-disk BMT. However, they are unlike a conventional cache in that they hold only recently modified rather than all recently accessed BMT entries.

The current LFSM prototype uses only two data logs, the *hot* log, which holds logical blocks that are sufficiently frequently written, and the *cold* log, which holds everything else. It constantly computes a moving average of each written logical block's past update counter, and classifies a written logical block as hot if its average update counter is above T_{update} within a moving window T, which is empirically chosen to be an hour in the current prototype. When the system starts up, LFSM lets these two logs compete for a fixed-sized memory pool, and when the utilization of the memory pool reaches 95%, the amount of memory occupied by each log at that instant becomes its allocation.

LFSM consists of two threads. The first thread is responsible for synchronous flash disk logging whereas the second thread is responsible for asynchronous BMT updates and garbage collection. The second thread always yields to the first thread whenever possible. The asynchronous background thread constantly scans through the BMT page by page to commit pending BMT updates, and starts reclaiming erasure units from a data log as soon as the log's percentage of free blocks falls below a threshold, T_{free}. Note that a log's free block percentage is computed with respect to its total memory allocation. To reclaim erasure units from a data log, LFSM reads in one or multiple erasure units at the log's head, copies their live blocks to a free erasure unit at the log's tail, frees these erasure units at the head and continues this until the free block percentage is above T_{free}.

Each BMT update log record is 20 bytes, and contains the following information: 4-byte length field for the size of the associated logical block write operation, 8-byte logical block number, 4-byte sequence number and 4-byte commit point. The LFSM prototype tries to compress the first sector of the physical block sequence allocated to a logical block write operation. If the extra space squeezed out by the compression step exceeds 20 bytes, it is used to hold the logical block write operation's BMT update log entry. Otherwise, an extra 512-byte sector is allocated to hold the BMT update log entry.

LFSM reads in BMT entries one erasure unit (256 KB) at a time, and commits pending updates associated with a BMT page to disk only if there are enough of them. The rationale is that it is better to defer the commit of pending BMT updates as much as possible in order to maximize the effectiveness of each update commit operation. The decision of using the erasure unit as the basic disk I/O unit is consistent with the fact that the flash disk's FTL needs to read an entire erasure unit even if only a small portion of it is overwritten.

To access a logical block, LFSM needs to perform a BMT look-up to identify the corresponding physical block. LFSM first consults with the per-BMT-page queues. If there is a hit, the corresponding BMT entry is returned. Otherwise, the corresponding 512-byte sector in the BMT is read into memory and the corresponding BMT entry is returned. However, when a 512-byte BMT sector is read into memory, the pending BMT updates associated with the sector will not be committed to disk. Rather, BMT updates are always committed to disk using sequential writes in the background.

Every time an erasure unit is reclaimed, the BMT updates corresponding to the dead physical blocks in the erasure unit can be removed from their per-BMT-page queues if they have not been committed to disk yet. Because BMT updates are committed to disk more rapidly than blocks are reclaimed, it is rarely necessary to remove pending entries from BMT update queues.

In addition to the BMT, LFSM also maintains two in-memory data structures, one keeping track of the update frequency of each logical block, and the other maintaining the utilization or the number of live physical blocks in every erasure unit. Both are updated after every logical block write operation. However, they don't need to be disk-resident because after a crash, the per-logical-block update frequency

can be recomputed from scratch, and the per-EU utilization data structure can be reconstructed from the BMT.

5.3 Performance Evaluation

In this section, we first demonstrate the effectiveness of LFSM's random write optimization in reducing the average write latency, and its impact on the average read latency. Then we employ block-level traces associated with three benchmarks to perform a sensitivity study on the impacts of three parameters of LFSM: the threshold of logical block update frequency known as T_{update}, the percentage of free blocks in each log that triggers a reclamation operation, and caching of BMT. Finally, we examine the end-to-end performance gain of LFSM under three standard benchmarks.

5.3.1 Methodology

The latency of a logical block write operation under LFSM is affected by the compression step for its first sector, the BMT look-up, its associated physical write to the chosen log, and the competing background operations of committing pending BMT updates to disk and free block reclamation.

To assess the contribution of the background BMT updates, we developed a version of LFSM that keeps the entire BMT in memory. This version not only completely removes the background BMT updates, but also does away with the BMT look-up overhead, which in turn consists of a scanning of a per-BMT-page link-list queue and an optional access to the on-disk BMT.

To gauge the performance impact of LFSM's background garbage collection activity, we use a bursty workload that consists of a burst of writes, followed by an idle period, another burst of writes, followed by another idle period, etc. Because the background thread that performs garbage collection always yields to the main logging thread, most of the garbage collection activity is expected to occur in the idle periods. By gradually increasing the length of the idle period, we can gradually decrease the average write latency and determine the minimal idle period length at which the average write latency remains unchanged. This minimal idle period length corresponds to the performance impact of background garbage collection.

The test machine used in the following experiments is a Lenovo Thinkpad T43 with 2.66GHz CPU and 1-GB memory and the SSD disk is Samsung's 32 GB SATA SSD. Because Levono T43 only supports a PATA interface, we install a SATA hard drive connector to convert its native PATA interface to a SATA interface and work with the SATA SSD disk.

We developed a synthetic workload generator to generate a sequential workload, where the target block addresses of the disk I/O requests are sequential, and a random workload, where the target block addresses of the disk I/O requests are random. This workload generator is implemented as a kernel-level thread that feeds the created disk I/O requests to the LFSM driver with an inter-request interval M and the number of

Delay Component	Random Workload (Unit: msec)		Sequential Workload (Unit: msec)	
	4 KB	8 KB	4 KB	8 KB
Metadata Lookup	0.2	0.2	0.01	0.02
Data Compression	0.11	0.12	0.11	0.11
Sequential Flash Disk Write	0.33	0.70	0.26	0.44
Garbage Collection and BMT Update Commit	0.99	0.91	0.01	0.01
Total Latency	1.63	1.93	0.38	0.58

Table 5.1: Breakdown of the average write latency under a random write workload and a sequential write workload when the write request size is 4 KB and 8 KB.

outstanding requests below a pre-defined value, K, which is set to 1 if not otherwise specified.

We also collected a set of block-level disk access traces from the hypervisor of a machine that runs XEN HVM and the following three applications on one or multiple virtual machines: the TPC-E workload generator running against a PostgreSQL DBMS instance on Linux of Fedora Core 8, the SPECSfs [182] workload generator against a Windows CIFS file server, and the LoadGen benchmark [184] against a Windows 2003 Exchange Server. As the current LFSM prototype is implemented as a device driver under Linux, we use TPC-E and SPECSfs NFS workload for the end-to-end performance evaluation.

- TPC-E Trace
  TPC-E [185] is a newly-introduced OLTP benchmark that simulates the OLTP workload of a brokerage firm. DBT-5 [186] is an open-source TPC-E implementation using PostgreSQL as the backend DBMS. DBT-5 initializes the brokerage database with 5,000 customers, the scale factor to 500 and the number of initial trade days to 200, and runs the TPC-E transactions for 1 hour. The collected TPC-E trace is a largely random workload with very poor data locality. The average disk read/write size is 8 KB, with 43% of the disk I/O requests being writes and the rest being reads.

- CIFS Trace
  The CIFS benchmark in SPECsfs 2008 is a synthetic workload simulating the typical load on production-mode Windows file servers. In this workload, there are 100 concurrent client processes and 1 server, and the number of sustained CIFS operation ranges from 10 to 100 with 10 as the increment. The average disk I/O request size for reads and writes in the resulting trace is 4 KB. 25% of the disk I/O requests are writes and the remaining are reads.

- Exchange Trace
  The Exchange trace is collected from a Windows Exchange 2003 server with

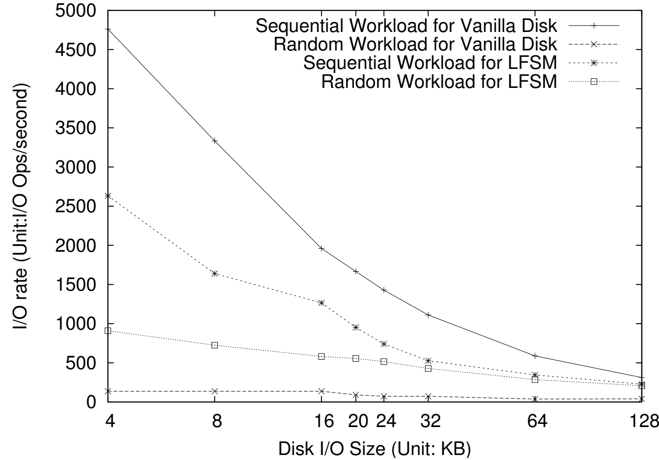


Figure 5.2: *Effectiveness of LFSM's random write performance optimization under a sequential write and a random write workload when the disk write request size is varied. The BMT is stored on disk.*

a load generator called LoadGen [184] developed by Microsoft Corporation. LoadGen simulates the workload of a medium-sized corporation's email server. The load generator runs for 1 hour with 1,000 email accounts and 1 user group. The average number of sustained tasks in each email is 132. The average disk I/O request size for reads and writes is 16 KB and 4 KB, respectively. 99% of the disk I/O requests are write requests and the remaining are read requests.

- TPC-E Workload
  The TPC-E benchmark is the same as in TPC-E Trace but runs directly on top of LFSM device. The scale factor is 100, the number of initial trade days is 200, and the number of customers are varied in the evaluation.

- NFS Workload
  The NFS benchmark in SPECsfs 2008 is similar to CIFS but targets Linux file servers. The NFS benchmark has 10 concurrent client processes and 1 server, and the number of sustained NFS operation ranges from 10 to 100 with 10 as the increment.

5.3.2 Performance Results

To evaluate the effectiveness of LFSM's random write performance optimization, we feed the LFSM driver with a sequential write and a random write workload, and measure the latency of each disk write request. In this experiment, the workload-generating kernel thread issues a disk write request to LFSM, waits until the preceding write is completed, then issues the next write, etc. The initial disk image is populated with 90% utilization so that there are still free erasure units to accommodate incoming disk write requests. The total size of each synthetic write workload set to 10% of

the disk capacity so that LFSM's garbage collection mechanism is triggered in the background. We perform several experiment runs, each corresponding to a different disk write request size.

Under LFSM, the latency of a logical block write operation is affected by (1) a BMT look-up, which is translated to a flash disk read of a 512-byte sector, (2) a compression step to create space for a BMT update log entry, (3) a sequential flash disk write, and (4) possibly interference from LFSM's background disk read/write activity. Table 5.1 shows the detailed breakdown of the average write latency under a random and sequential write workload. The performance cost of the BMT lookup under the sequential workload is 20 times lower than that under the random workload because each BMT sector fetched can be used to service multiple BMT look-ups under the sequential workload but only one BMT look-up under the random workload. The performance impact of LFSM's background activity, i.e., garbage collection and commit of pending BMT updates, is much higher under the random workload than under the sequential workload for the following two reasons. First, the random workload triggers more flash disk read accesses because it requires more on-disk BMT look-ups. Second, lower locality in the random workload creates more demands for free erasure units, and thus triggers more intensive garbage collection activities. Finally, the more intensive competition for the disk I/O channel under the random workload causes the flash disk write takes more time under the random workload than that under the sequential workload.

Figure 5.2 shows the I/O rate of the test flash disk with and without LFSM under a sequential write and a random write workload. The effectiveness of LFSM's random write performance optimization is conclusively demonstrated by the fact that the I/O rate improvement over the vanilla flash disk is a factor of 6.6 and 5.3 for 4-KB and 8-KB disk write requests, respectively. Under the sequential write workload, the I/O rate of the test flash disk with LFSM is lower than that of the same flash disk without LFSM, because of the additional steps of (1), (2) and (4). The I/O rate of LFSM under the sequential workload is higher than that under the random workload because the average BMT look-up delay is much smaller under the sequential workload.

Figure 5.3 shows the I/O rates of LFSM with on-disk BMT and LFSM with in-memory BMT running under a sequential write and a random write workload when the disk write request size is varied. Having BMT completely in memory eliminates the need to look up the on-disk BMT and committing pending BMT updates to the disk. As a result, the I/O rate of LFSM with in-memory BMT under the sequential workload is the same as that under the random workload. The I/O rate of LFSM with in-memory BMT under the random workload is even better than that of LFSM with on-disk BMT under the sequential workload because the former eliminates all BMT-related overheads. The I/O rate of LFSM with on-disk BMT under the sequential workload is still higher than that under the random workload, again because of the better disk access locality in BMT look-up under the sequential workload.

LFSM needs to compress the first sector of every data unit when it is written, and decompress the first sector of every data unit when it is read. The compression

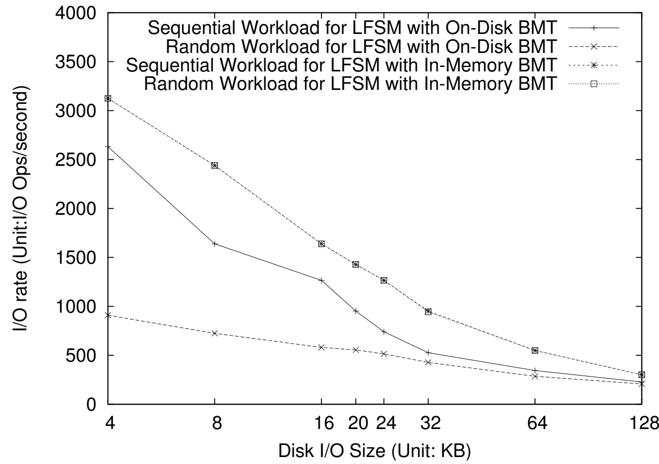


Figure 5.3: *I/O rate comparison between LFSM with on-disk BMT and LFSM with in-memory BMT under a sequential write and a random write workload when the disk write request is varied.*

Read Size (Unit: KB)	**Random Workload (Unit: msec)**		**Sequential Workload (Unit: msec)**	
	In-Memory BMT	On-Disk BMT	In-Memory BMT	On-Disk BMT
0.5	0.243	0.436	0.19	0.21
1	0.243	0.436	0.19	0.21
4	0.243	0.436	0.19	0.21
8	0.32	0.503	0.289	0.295
16	0.48	0.676	0.461	0.47
32	0.68	0.927	0.676	0.68
64	1.132	1.52	1.13	1.13
128	1.812	2.24	1.806	1.813

Table 5.2: The average read latency of LFSM with in-memory BMT and LFSM with on- disk BMT under a random read and sequential read workload when the disk read request size is varied.

Workload	**Distribution of Compression Region(%)**			
	1	2-5	6-8	Others
TPC-E	95.9	2	1	1.1
CIFS	95.8	1.8	0.9	1.5
Exchange	93.7	5.2	0.7	0.4

Table 5.3: The distribution of compression region for all three workloads.

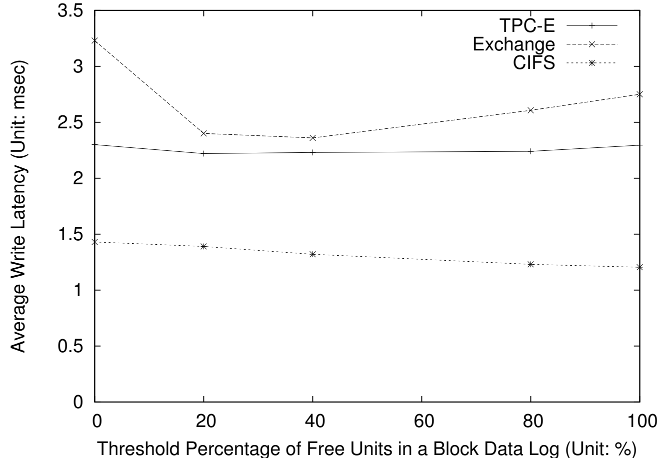


Figure 5.4: *The average write latency of LFSM under three block-level traces as T_{free} is varied.*

and decompression steps typically take about 0.1 msec and 0.01 msec, respectively. Because LFSM introduces an additional overhead during a read in the form of BMT look-up and decompression, the average read latency of LFSM with on-disk BMT is higher than that of a vanilla flash disk without LFSM. This performance difference becomes even more significant when the disk read request size is less than 4 KB, because the current LFSM prototype uses 4 KB as the minimum data unit for write, and thus needs to fetch the entire 4-KB block even if a disk read request's size is smaller than 4KB.

For TPC-E, CIFS and Exchange workload, Table 5.3 shows the distribution of the compression region. A compression region is defined as the minimal number of first several sectors to squeeze out enough space to hold the metadata, or ∞ if the payload is not compressible. For all three workloads, more than 93% of the write payload can squeeze out enough space from the first sector, which shows that partial compression is applicable for these three typical workloads.

Although the performance of flash disk read is largely unaffected by the locality in the input workload, the read latency of LFSM with on-disk BMT is worse under a random read workload than under a sequential read workload, because locality affects the BMT look-up delay. When the entire BMT is completely memory-resident, the difference in read latency under the sequential and random workloads shrinks substantially. However, even for LFSM with in-memory BMT, its read latency under the random workload is still slightly higher than that under the sequential workload, because of the page-mode operation of flash memories, similar to mainstream DRAM technologies.

5.3.3 Sensitivity Study

In this section we study the impact of three algorithm parameters in LFSM, the update frequency threshold for determining if a logical block is hot or cold, T_{update},

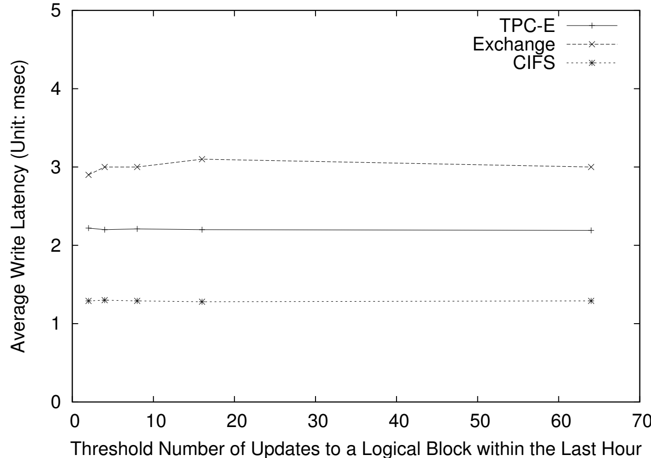


Figure 5.5: *The average write latency of LFSM under the three block-level traces as T_{update} is varied.*

the threshold percentage of a log's blocks being free that triggers garbage collection for both hot and cold logs, T_{free}, and the total number of pending BMT updates allowed in per-BMT-page queues, L_{queue}. In these experiments, BMT is stored on disk, and a kernel-level I/O thread issues both read and write requests in the gathered block-level traces to LFSM one after another. The default values for T_{update}, T_{free} and L_{queue} are 2 updates per hour, 2% and 1000000, respectively.

Figure 5.4 shows the average write latency of LFSM under the three block-level traces when T_{free} is varied. Larger T_{free} triggers asynchronous garbage collection more frequently so as to avoid synchronous garbage collection in the presence of a burst of demands for free blocks. On the other hand, smaller T_{free} reduces the amount of live block copying when reclaiming an erasure unit and thus the garbage collector's load on the disk I/O channel, making it less likely to interfere with LFSM's main write thread. If an input workload is not update-intensive, contention with the background garbage collector may be less an issue. To determine the optimal T_{free} for each trace, we first populated most of the disk image and then issued a series of reads and writes that are guaranteed to trigger the background garbage collector.

The Exchange trace is write-intensive with 99% of its disk access being writes. Because LFSM experiences a higher write request rate, the average write latency of LFSM under the Exchange trace is higher than those under the other two traces. As a result, the optimal T_{free} turns out to be 40%. A similar reasoning holds for the TPC-E trace. In contrast, because the CIFS trace has a small percentage of its requests being writes and thus only exerts a less intensive write workload, the drawback of larger T_{free} never comes into play. Consequently, the average write latency of LFSM keeps decreasing as T_{free} is increased to reclaim erasure units more frequently.

Figure 5.5 shows the average write latency of LFSM under the three block-level traces as T_{update} is varied. The original idea behind LFSM's garbage collection algorithm is to use multiple cyclic FIFO logs, each reclaimed at a different frequency, to approximate a single log that bases the reclamation order on the per-EU utilization.

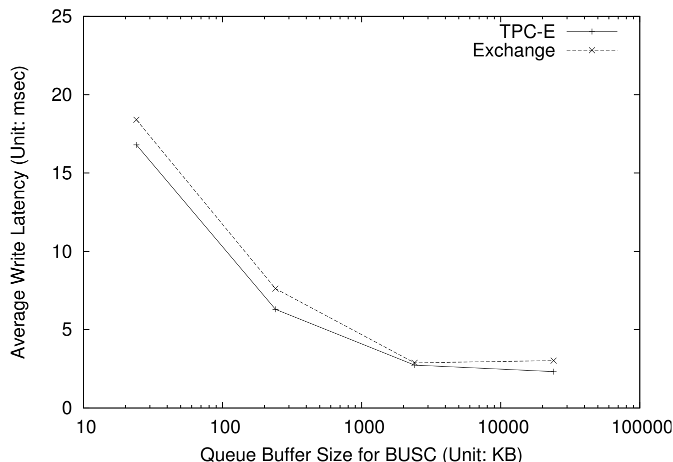


Figure 5.6: *The average write latency of LFSM under the TPC-E and Exchange traces when L_{queue} is varied. The X axis is in log scale.*

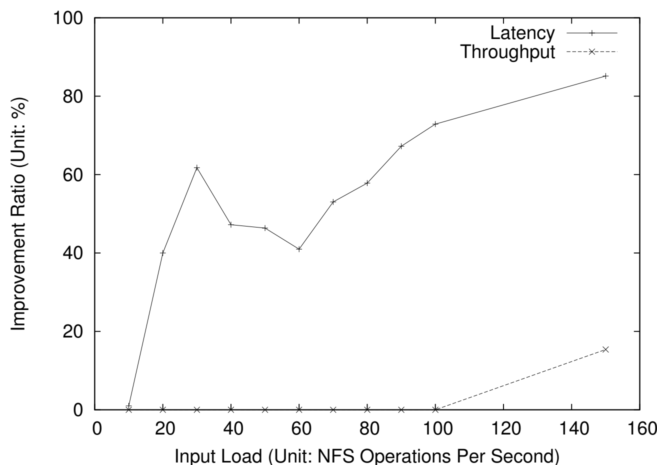


Figure 5.7: *The improvement ratio for both the latency and the throughput under the SPECsfs NFS workload.*

However, it turns out that a logical block's update frequency history is not a reliable indicator of whether it is going to be hot in the near future or not because a logical block could be used to hold parts of files with very different temperature in its life time. Consequently, the average write latency of LFSM is largely unaffected by T_{update} for all three block-level traces. This result suggests that more research on how to accurately determine a logical block's temperature is needed.

We chose the TPC-E and Exchange traces as input workloads to evaluate the impact of the parameter L_{queue}. We grouped disk access requests in this trace into bursts of 32, introduced a 100-msec interval to ensure the pending BMT updates have the required disk bandwidth resource to be committed to disk, and measured LFSM's average write latency while varying L_{queue}. Larger L_{queue} increases LFSM's ability to absorb bursts of write requests and aggregate more pending updates in each commit

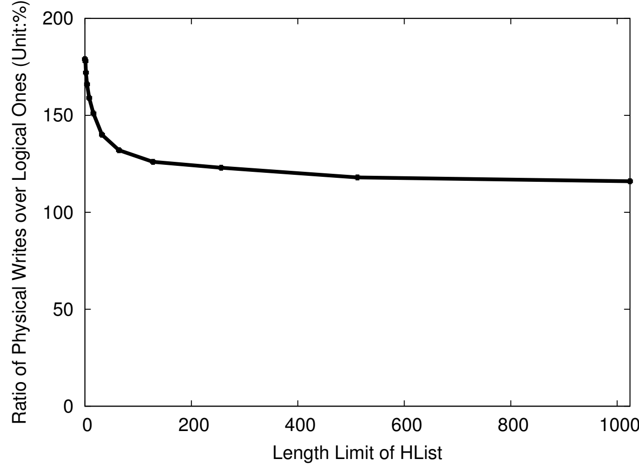


Figure 5.8: *The ratio of physical writes over logical ones with varied HList length limit for the TPC-E workload.* $T_{update} = 10$, *and* $T_{free} = 20\%$.

operation, but also increases the average BMT look-up time associated with every disk read and write under LFSM.

Figure 5.6 shows that for the transformed TPC-E trace, larger L_{queue} always decreases LFSM's average write latency. A closer examination reveals that the reason behind this is that the write requests in the TPC-E trace exhibit poor locality and as a result the average number of pending entries in each per-BMT-page queue rarely exceeds 10 in the whole experiment run. Consequently, increasing L_{queue} has no effect on the BMT look-up time but does successfully reduce the amount of disk I/O resource needed in committing pending BMT updates to disk. However, for the Exchange trace, the BMT look-up time is indeed slowed down when L_{queue} is increased from 1 MB to 10 MB, because the average length of per-BMT-page queue is increased from 50 when L_{queue} is 1 MB to 150 when L_{queue} is 10 MB.

Figure 5.8 shows that the ratio of physical writes over logical ones drops when the length limit of HList increases, demonstrating the effectiveness of LPU-GC. This is because larger length limit of HList allows more EUs to stabilize in HList and reduce the overall number of I/Os to copy valid pages in GC. When the length limit of HList is larger than 512, the ratio of physical writes over logical ones does not change much. This phenomenon shows that increasing the length limit does not always yields benefits because for a particular workload, the number of frequently invalidated pages, N_{FI}, is limited. Ideally, the length limit of HList should be equal to N_{FI}.

5.3.4 End-to-End Performance Evaluation

To gauge the end-to-end performance benefits of LFSM over vanilla flash disks, we implemented LFSM as a Linux device driver. We chose the TPC-E benchmark and the SPECsfs NFS benchmark because they can run under Linux. For the TPC-E benchmark, we chose the number of customers to vary the input load. For the SPECsfs

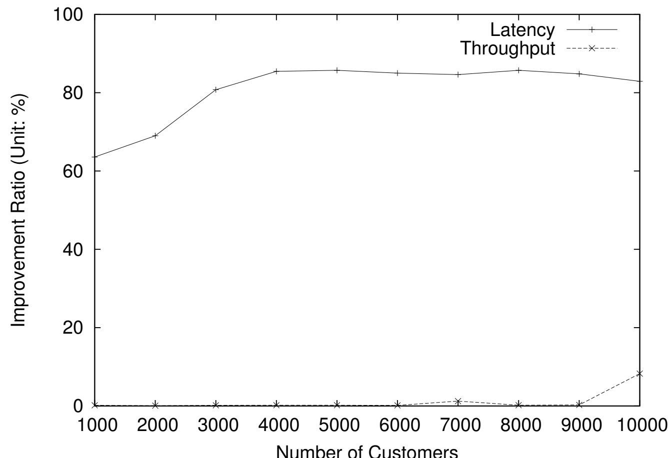


Figure 5.9: *The improvement ratio for both the latency and the throughput under the TPC-E workload.*

NFS benchmark, the target throughput can be set directly. For all experiment runs, $T_{update} = 10$, $L_{queue} = 1MB$ and $T_{free} = 20\%$. As a default, the proactive garbage collection is enabled.

Figure 5.9 shows that the average transaction latency improvement of *Trade-Update* transactions in the TPC-E benchmark generally increases with the input load, because, as the input load is increased, longer disk write latency of vanilla flash disk increases the lock waiting time and thus aggravates the transaction latency. The improvement in transaction latency ranges from a factor of 2.7 to a factor of 7.1. When the number of customers is 2000, LFSM completes a physical 4KB write in 0.9 msec on average. In contrast, it takes the vanilla flash disk 11 msec, or more than 10 times as long, to complete a physical 4KB write. However, as also shown in Figure 5.9, reduction in transaction latency does not translate into transaction throughput improvement when the input load is not latency-bound. When the number of customers in the TPC-E benchmark is no more than 10000, the input load is not latency-bound. Unfortunately our current testbed does not have enough flash disk capacity to allow us to perform experiments for when the number of customer is greater than 10000. Figure 5.7 shows a similar trend of latency improvement for NFS operations in the SPECsfs NFS benchmark to the transaction latency improvement in the TPC-E benchmark, and the underlying cause is the same: shorter disk write latency in LFSM than in vanilla flash disk. For the input load of 150 NFS operations per second, the average disk write latency of vanilla flash disk is 6.8 times larger than that of LFSM. Disk reads have little impact on the NFS operation latency, because the difference in disk read latency between LFSM and vanilla flash disk is within 10%. For the same reason as in the TPC-E benchmark, there is also no apparent throughput improvement for the NFS benchmark despite the significant reduction in average operation latency.

Table 5.1 suggests that under the truly random write workload, LFSM's background activities including garbage collection and BMT updates commit, can have

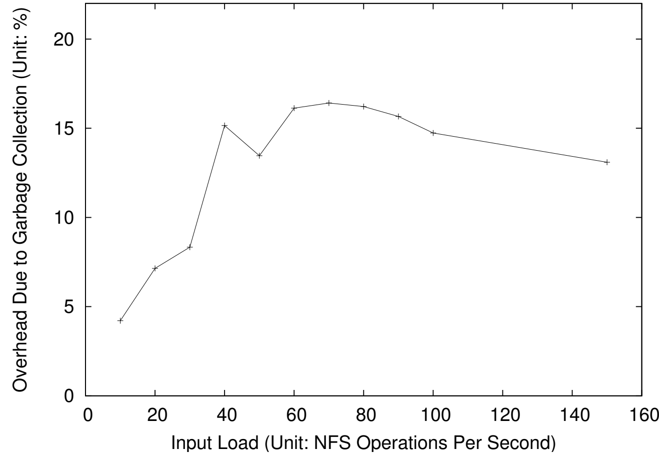


Figure 5.10: *The performance overhead due to proactive garbage collection for the SPECsfs NFS workload.*

a significant impact on the average write latency. To gauge the performance impact of garbage collection under a more realistic workload, we ran the NFS benchmark twice, once with garbage collection turned on and the other turned off. Figure 5.10 and Figure 5.11 show that the percentage differences in NFS operation latency and in TPC-E transaction latency due to background garbage collection increase with the input load, because larger input load exerts higher storage consumption pressure, which requires more frequent garbage collection. The latency impact of background garbage collection on the NFS operations and TPC-E transactions is no more than 16% and 15%, respectively, which are noticeable but not dominant.

5.3.5 Effectiveness of BOSC

In this subsection, we show the evaluation results of LFSM with BOSC and other alternatives using the random write workload. We employ three other alternatives to compare with BOSC. The first alternative, *In-Mem*, replaces the on-disk BMT in BOSC with a memory-resident BMT while the metadata logging and data logging are the same. The second alternative, *LRU-On-Disk*, is to employ an on-disk array with the traditional LRU memory cache. The third alternative, *One-Queue*, is similar to BOSC with a global queue instead of per-page queues. The *One-Queue-Trail* alternative employs a global queue to hold metadata updates. At the commit time, the first N queue items are ordered based on their target EU address, and the metadata commit follows this order. For the 32 GB raw flash capacity, we use 31 GB to log data payload, the BMT takes 62 MB because each mapping entry in BMT takes 8 bytes, the remaining space is used to hold metadata logging. The buffer memory is fixed as 512 KB. For BOSC and the *One-Queue* alternative, the 512-KB memory holds the metadata queuing items. For the *LRU-On-Disk* alternative, the 512-KB memory holds 512-byte sectors containing the metadata updates. The *In-Mem* alternative does not use buffer memory as it does not have on-disk counterpart. Instead, the

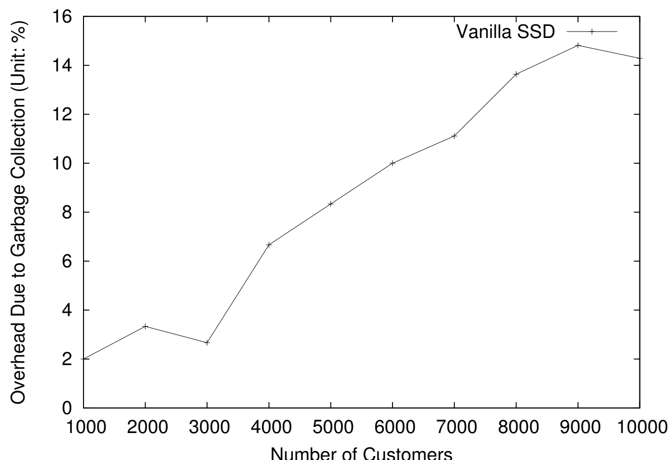


Figure 5.11: *The performance overhead due to proactive garbage collection for the TPC-E workload.*

In-Mem alternative consumes 62 MB to store the BMT in RAM. For the *One-Queue* alternative, the first 1000 metadata updates are ordered to commit metadata updates to the on-disk BMT.

To measure the latency of LFSM, we use a kernel thread to constantly generate new 4KB page writes, each of which follows the three-step procedure for writes in LFSM. The LBN of each new page write is randomly generated and is uniformly distributed in $[1, 7750000]$. In the run, there are totally 16 GB worth of new page writes logged and 32 MB ($16GB * \frac{8}{4K}$) worth of index records inserted. To demonstrate the tradeoff between the space overhead and the latency, we varied the metadata log size across experiments.

Figure 5.12 shows the average end-to-end latency of random writes with varied metadata log size for BOSC, the *In-Mem* alternative, the *One-Queue* alternative and the *LRU-On-Disk* alternative. Each bar contain 4 components, the striped component is the time spent on data logging, the gray component is the time to log metadata, the white component is the time spent on *critical commit*, and the black component is the time delay due to other background house-keeping tasks (e.g., metadata update and garbage collection).

The *In-Mem* alternative is not used in practice because BMT is persistently stored. The *In-Mem* alternative is only used to show the minimal latency when both metadata logging and data logging are turned on. For all metadata log sizes, the *In-Mem* alternative has the same latency, 1.0 msec. In this 1.0 msec latency, 0.45 msec latency is spent on the metadata logging as the grayed portion of the bar showed, and 0.55 msec latency is spent on the data payload logging as the white portion of the bar showed.

Different from the *In-Mem* alternative, the latency for BOSC increases as the metadata log size decreases because the critical commit takes longer for smaller metadata log sizes. The dark portion of the bar shows time spent on the *critical commit.* When the metadata log size reaches 512 MB, the latency of BOSC reaches 1.05 msec,

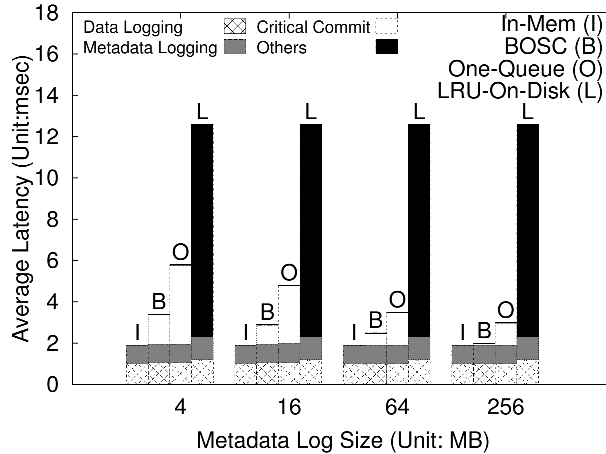


Figure 5.12: *The end-to-end latency breakdown of random 4 KB writes with varied metadata log size under the* In-Mem *metadata update alternative, BOSC, the* One-Queue *metadata update alternative, and the* LRU-On-Disk *metadata update alternative. The queuing memory is 512 KB, each metadata is worth of 8 bytes, 31 GB out of 32 GB is reserved for logging data payload from the Flash disk.*

which is 5% larger than that of the *In-Mem* alternative, which indicates that BOSC can eliminate the performance bottleneck due to metadata updating when the metadata log size is 1.6% (512 MB/32 GB) of the overall capacity.

In comparison, the latency for the *One-Queue* alternative is at least 50% (512 MB metadata log) larger than that of the *In-Mem* alternative for all metadata sizes because the *One-Queue* alternative does not commit metadata updates in the same EU in one batch and has a larger probability of being blocked by the *critical commit*. In particular, when the metadata log size, is 256 MB, the latency for the *One-Queue* alternative is 50% larger than that for BOSC.

The latency for the *LRU-On-Disk* alternative does not change with the metadata log size because updates to the on-disk BMT is random. On average, 1100 updates can be cached in the sector-based memory cache. When the cache is full, the oldest section is evicted from the cache. From the perspective of the on-disk BMT, the metadata updates are random. As a result, regardless of the metadata log size, the latency is 10.5 msec, which is the latency for random sector writes for the Samsung SATA SSD.

Chapter 6

Scalable and Parallel Segment-based De-duplication for Incremental Backups

This chapter is organized as below. Section 6.1 describes the motivation of scalable de-duplication based on the incremental backups. In section 6.2, we propose our scalable and parallel segment-based de-duplication techniques for incremental backups. Section 6.4 verifies design decisions based on the analysis of a collected trace.

6.1 Background and Motivation

As data redundancy grows in storage systems [187], data de-duplication functionality becomes more and more prevailing in storage systems. Depending on whether the storage system serves live storage requests, data de-duplication can be either a feature in primary storage systems [153–155], also known as online storage systems, or a feature in secondary storage system, also known as nearline storage systems. However, de-duplication in these two storage systems have different focuses. For the primary storage systems, de-duplication focuses more on the data consistency and performance overhead. In contrast, for the secondary storage systems, de-duplication focuses more on the de-duplication throughput and de-duplication quality due to the huge amount of input data for data de-duplication. In this technical report, we focus on de-duplication for the secondary storage systems because metadata management in data de-duplication of secondary storage systems plays a critical role in achieving good de-duplication throughput and quality.

Depending on the de-duplication object types, there exist two categories of data de-duplication techniques. The first type of data de-duplication takes a file as the basic object to de-duplicate, also known as *file-level* de-duplication. The second type

of data de-duplication takes a logical volume as the basic object to de-duplicate, also known as *block-level* de-duplication. The *file-level* de-duplication can leverage the file-level information to optimize the de-duplication algorithm [159]. However, *file-level* de-duplication either de-duplicate files directly the file system [153, 154] or de-duplicate files at the block level [159]. In contrast, *block-level* de-duplication takes blocks from a logical volume as the de-duplication input without the knowledge of files and file system. Therefore, *block-level* de-duplication can be transparent to storage systems employing the de-duplication functionality, which is critical for the successful deployment of the de-duplication functionality. The Data Domain de-duplication appliance [27] is an excellent example of the *block-level* de-duplication.

De-duplication techniques can be categorized into full backups and incremental backups based on the characteristics of the de-duplication input. For full backup systems, at the backup time, on one hand, the backup client prepares the fingerprint values of *all* data blocks of the backup snapshot, transmits all fingerprint values, waits for the backup server to figure out which data blocks are newly created, and only transmits all newly data payload. On the other hand, the backup server receives all fingerprint values, compares fingerprint values to the fingerprint index, and finally returns the new fingerprint values to the backup client. Because all data block fingerprint values are compared in the fingerprint index and most of blocks are not changed across snapshots, the de-duplication ratio is touted to be up to 100 for these data de-duplication systems.

Block change tracking (BCT) [188, 189] at the backup client side is the basis for incremental backups, and it changes the interaction between the backup client and the backup server in full backup systems [27]. Instead of blindly transmitting all fingerprint values of all data blocks from the backup client to the backup server, we can reduce the number of processed fingerprint values and achieve a more realistic de-duplication ratio by transmitting fingerprint values of changed blocks. More concretely, a de-duplication agent resides at the backup client side to collect changed blocks during the run time by employing block change tracking (BCT) mechanism. At the backup time, the agent prepares the fingerprint values of only changed blocks, and asks the backup server to figure out the new data blocks by checking the fingerprint index. We base our research on the block change tracking at the backup client for data de-duplication. Design details of block change tracking are in section 6.2.9.

The increase of the fingerprint index size poses a performance challenge for data de-duplication systems. If the full fingerprint index can not fit into the main memory, a query of the fingerprint index is very likely to trigger a disk I/O, which is prohibitive in data de-duplication. There exist two alternatives in mitigating the problem. The first approach [27], the *data-domain* approach, employs the locality-preserved caching to amortize the I/O cost of fetching a fingerprint by loading into the cache fingerprints of proximal data blocks. The fingerprints of proximal data blocks are organized into a *fingerprint container*. The assumption is that if a block is queried for de-duplication, the neighbouring blocks are very likely to be checked for de-duplication, which is true for de-duplication based on the full backup. However, the assumption does not hold

for de-duplication based on the block change tracking because fingerprints of proximal data blocks may not be used by the input changed blocks. If many proximal data blocks' fingerprints are not used in the data de-duplication, the caching efficiency decreases and the amortized cost of querying a fingerprint index increases. For BCT-based de-duplication, we propose to use file instead of proximal data blocks as the basic unit of de-duplication for two reasons. First, the file-based fingerprint container is resistant to the intra-file fragmentation. That is, a continuous physical block range may contain blocks for two different files, because changed blocks belong to one file, it is a waste of caching memory to load fingerprints of all physical blocks in the range. Second, we can selectively load fingerprints of related file portions into the cache based on the file-level information. For example, if the input changed blocks of a file has a fixed length, we can load fingerprints of the corresponding file portion with the same length into the cache instead of blindly loading the whole file's fingerprints into the cache. Regardless of the caching strategy, a de-duplication query can trigger two disk I/Os, the first one to fetch the fingerprint from the disk, the second one to load the corresponding fingerprint container into the cache.

The second approach [28], denoted as the *sparse-index* approach, reduces the I/O overhead by sampling the fingerprint index until it fits into main memory so that a query of the fingerprint index does not involve disk I/Os. Similar to the *data-domain* approach, the *sparse-index* approach relies heavily on the caching efficiency of the fingerprint containers to amortize the I/O cost in comparing the input fingerprints with the stored fingerprints. Fingerprints are organized based on the sequence of physical data blocks, not a file. Each sequence of physical data blocks is sampled with the same sampling rate regardless the de-duplication history of each sequence. When the input consists merely changed blocks, those sequence of physical data blocks that are not source of duplicates will consume too much memory, and potentially reduce the de-duplication quality. We propose the file-based varying-frequency sampling method to vary the sampling rate based on the file's de-duplication history. The hypothesis is that a file portion or a file is more likely to be the source of duplicate after it is first detected as a duplicate. For those file portions or files that are not detected as duplicates over time, we can reduce their sampling rate because they are not likely to be the source of duplicate.

State-of-Art data de-duplication techniques focus on the full backups, where all logical blocks of a logical volume are de-duplicated with existing stored blocks even if only a small portion of all logical blocks have changed since last backup. These data de-duplication techniques [27, 28] may work with incremental backups, but do not optimize for incremental backups.

We explore new opportunities, challenges and solutions of data de-duplication for incremental backups when the primary storage system can incrementally track written blocks and only backup these written blocks. In concrete, we assume that the primary storage system can track written blocks in a snapshot image since last snapshot. These written blocks are referred as incremental changed blocks. At run time, incremental written blocks are tracked by block change tracking (BCT) mechanism. At backup

time, these incremental written blocks are taken as input to the de-duplication system.

We propose four novel techniques to improve the de-duplication throughput with minimal impact on Data De-duplication Ratio (DDR) when incremental changed blocks are the input:

1. Containers are constructed based on the history of de-duplication instead of pure logical address proximity. For incremental changes as the input, the semantics of data locality is more subtle because it is hard to predict what could appear in the incremental change stream. We propose to put all incremental changes that share content with each other into the same container to capture the data locality due to changes by the same set of applications.

2. Sampling rates of segments are differentiated by having a fixed sampling rate for stable segments and by assigning a variable sampling rate for unstable targets of de-duplication when fitting fingerprints into RAM. A stable segment is defined as a segment that does not change over de-duplication history. The intuition behind the stable segment is that hosts tend to share an unchanged object larger than a block. By observing the history of data de-duplication, we can pin-point the stable segment and use 1 fingerprint out of the stable segment to represent the segment in the fingerprint index. We adopt a LRU policy to vary the sampling rate of all other "unpopular segments in the sampled fingerprint index.

3. Per-segment summary structure can be leveraged to avoid unnecessary I/Os involved in de-duplication. For those segments with large sizes that rarely change, it is not necessary to compare individual fingerprints in a segment one by one, and therefore reduces the disk I/Os to fetch individual fingerprints from disks.

4. For distributed storage systems, we can distribute the de-duplication functionality to multiple participating storage nodes. Fingerprint index and containers are distributed and split to participating storage nodes based on the consistent hashing of the fingerprint. Also, the summary structure is also distributed and split to participating storage nodes based on the consistent hashing of the root summary.

The resultant full-fledged de-duplication system is denoted as distributed deduplication system (D-Dedup). In this technical report, we focus on the design decisions employed in D-Dedup. The full-fledged implementation and evaluation will not be covered in this technical report.

6.2 Design

In this section, we first present the high-level data flow for the proposed de-duplication architecture in subsection 6.2.1, then elaborate on three novel de-duplication techniques for the standalone version of D-Dedup in subsection 6.2.2, 6.2.4, 6.2.3, 6.2.5 and 6.2.6. In subsection 6.2.7, we propose a scalable garbage collection methods in a secondary storage system with the de-duplication functionality. In subsection 6.2.8, we describe ways to distribute de-duplication functionality to multiple slave nodes, and present the detailed architecture and design of the full-fledged D-Dedup.

6.2.1 Data Flow for De-duplication

Before we delve into the details of de-duplication, figure 6.1 shows the high-level data flow in D-Dedup. This figure omits the parallel portion of D-Dedup, which can be found in subsection 6.2.8.

Incremental changes of a backup logical volume are first partitioned as segments, and related details can be found in subsection 6.2.2. Basically a segment is a group of blocks that are adjacent with each other. The memory-resident Sampled Fingerprint Index (SFI) contains pivot (sampled) fingerprints that point to the corresponding containers that the pivot fingerprints are from. The memory-resident Segment-based Summary consists of summary fingerprints of segments that are larger than a threshold, $T_{summary}$. The memory-resident Container Cache Store is a cache of containers organized in an LRU order. The on-disk container store holds all containers, and the on-disk fingerprint index holds fingerprints that are evicted from the memory-resident SFI, but not contains fingerprints that are not sampled in the memory-resident SFI.

Each input segment queries SFI by issuing fingerprint query for each individual fingerprint in the segment. The query of each individual fingerprint returns with a $\langle containerID, segmentID \rangle$ pair. Query results of fingerprints within the same segment are analysed to find out if there is only one store segment corresponding to the input segment. If yes, the input segment queries the In-Memory Segment-based Summary with $\langle containerID, segmentID \rangle$ to find if the container needs to be fetched in. Otherwise, containerID is used to query the In-Memory Container Store Cache to see if the container is cached. If the container is not cached, the container is loaded from the on-disk container store. At the same time, some containers cached in the Container Store Cache are evicted from the cache to leave space for the newly fetched container.

After the container is loaded, each input segment queries the per-container index to find the offset of a fingerprint routed to this container. The offset can be used to retrieve physical locations of the block because fingerprints are stored as an array in the container. By querying the segment index using the offset, we can retrieve the segment information of that particular fingerprint.

The output of the data flow is physical locations of logical block addresses in the input segment if there exist duplicates. In concrete, a Change List of the Logical-To-Physical (L2P-CL) map of the backup logical volume is generated as the output.

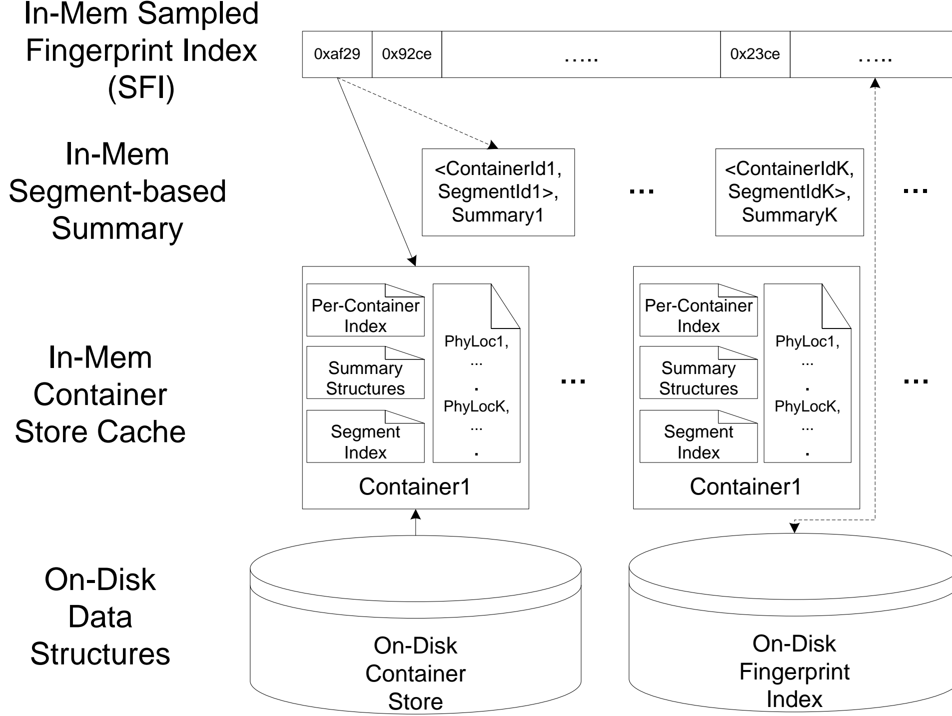


Figure 6.1: *Data flow of the standalone data de-duplication.*

6.2.2 Segment Identification

Segment is the basic unit in D-Dedup, and is determined by both the current input and the input history. In concrete, if several blocks are contiguous in incremental changes, these blocks belong to the same *input segment*. The *input segment* is then compared with *stored segments* to form the *store segment*. When the input segment and stored segment conflicts with each other, each fingerprint can belong to up to K segments. If there are more than K segments for a fingerprint, the oldest segment is removed from the fingerprint.

In D-Dedup, we allow up to K segments for each block. That is, we keep track of the last K segments that ever associated with a block. We employ a segment index (SI) to keep track of the mapping between the logical block address range and the segment in the following discussion. An interval tree can be employed to map offset intervals to corresponding segments. Because intervals in the interval tree do not interleave with each other, the interval tree can be a B^+ tree keyed by the start offset of each interval.

When a new store segment is created, a globally unique segment identifier is created. Each segment identifier is 64-bit, which means that there can be 2^{64} segments. Empirically each segment is more than one 4K bytes, the segment identifier range is large enough to ensure uniqueness.

As the segment evolves over time, it is likely a segment can container disjoint offset intervals. For example, if 4 leading fingerprints in the input segment (1, 6)

matches the store segment (8,11), but the remaining 2 fingerprints are new. The remaining fingerprints are appended to the end of the container. If the original container size is 16, the remaining 2 fingerprints correspond to the offset interval (16, 17). As a result, the input segment contains two offset pairs, (8, 11) and (16, 17). To keeps track of the mapping from the identifier to multiple offset intervals, each identifier maps to a list of offset intervals. The mapping from the segment identifier to the offset intervals is part of SI.

When the input segment has a length larger than a threshold, $U_{segment}$, the input segment is chopped into multiple sub input segments so that each sub input segment has a size smaller than or equal to $U_{segment}$. In another word, we enforce an upper bound on the size of the input segment.

Figure 6.2 shows an example of how store segments evolve when K = 2. Numbers in the rectangle represents fingerprint values. Logical block numbers of the input segment are marked below the rectangle. Numbers under the store segment represent the fingerprint offset in the container. As Input-1, there are two input segments, segment (a) and segment (b). After de-duplication, segment (a) is found to be duplicate of stored segment (1), but segment (b) is found to span two stored segments (segment (2) and segment (3)). The stored segment (1) and input segment (a) are the same so there is no change to the stored segment. In contrast, fingerprints 0x3f and 0x6d map to two segments: segment (2) and segment (b), while fingerprints 0x5c and 0x86 map to two other segments: segment (3) and segment (b). Input segment (c) matches fingerprints with offset from 51 to 55, but two remaining fingerprints (0xf7 and 0x8d) do not match any stored fingerprints. Because only 2 segments per fingerprint is allowed, segment (2) is removed for fingerprints 0x3f and 0x6d, and segment (3) is removed from fingerprints 0x5c and 0x86. After the removal, fingerprints with offset from 52 to 55 map to two segments, segment (b) and segment (c). Two new fingerprints, 0xf7 and 0x8d, are appended to the container for segment (c). Segment (c) maps to two offset intervals, 52 to 55, and 58 to 59, respectively.

A stable segment is defined as a *store segment* that does not get modified after its creation. For a stable segment, because all fingerprints in it appear all at once, 1 fingerprint is enough to represent the corresponding stable segment. However, in practice, it is hard to ensure a segment is stable because the segment can change any time in the future. To avoid confusion, we define a basic sharing unit (BSU) as a store segment that is referred to by more than T_{BSU}. The set of BSUs is a superset of stable segments. In our design, BSU is an approximation of the stable segment. We will use BSU instead of the stable segment onwards.

6.2.3 Variable-Frequency Sampled Fingerprint Index

In this subsection, we describe segment-based Variable-Frequency Sampled Fingerprint Index(VFSFI). Because fingerprint index cannot fit into RAM, we advocate the idea of sampling fingerprints to fit the fingerprint index into a fingerprint index cache. The sampling is based on characteristics of individual segments.

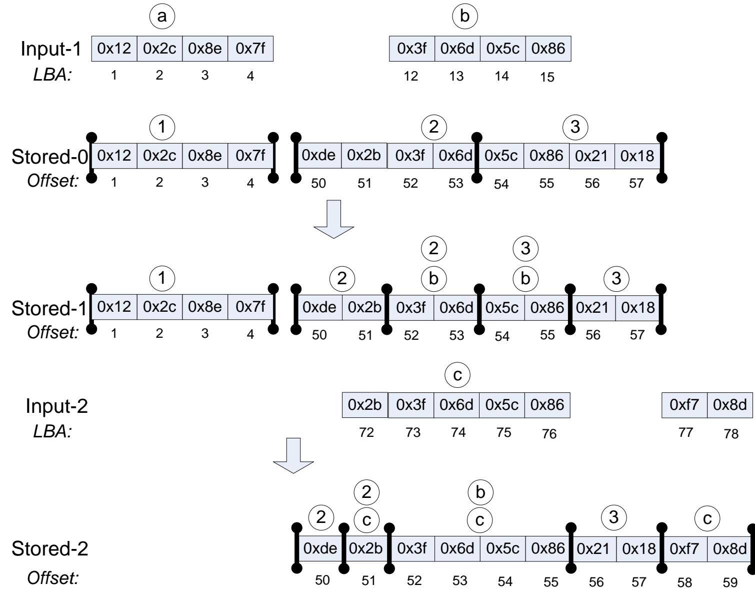


Figure 6.2: *An example of segment identification.*

The fingerprint index cache is implemented as a hash table. Each entry in the fingerprint index cache is keyed by the fingerprint value, and the entry contains one container ID, and up to K (e.g., K = 2) segment IDs in the specified container if multiple segments share the same fingerprint. Note that this K is the same K as mentioned in subsection 6.2.2.

There are 2 ways in varying the sampling rate of segments:

1. For segments that are BSUs, the first out of all fingerprints in a BSU is sampled to represent the BSU. Note that the sampling does not hurt the de-duplication ratio because fingerprints in a BSU always appear in one shot. However, the de-duplication throughput can be improved due to the saving of the memory space.

2. When the memory is out of RAM space, instead of reducing the sampling rate of all segments, those segments that are not referenced for a long time reduce their sampling rates before being evicted when the sampling rate reaches the minimal value. An LRU list is used for this purpose. In particular, BSU segments have a virtual sampling rate, which is the same as the sampling rate for a regular segment. The virtual sampling rate is reduced when BSU reaches the end of the LRU list.

For the reduction of sampling rates, each store segment is initially assigned an *initial sampling rate*. For example, if the initial sampling rate is 1/16 (N = 16 in Step 38), the first fingerprint of each 16 fingerprints is sampled into the global SFI. When the store segment reaches the tail of SFI's LRU list, the sampling rate is reduced to 1/32 (half) is there are more than 1 sampled fingerprint for the segment. If the

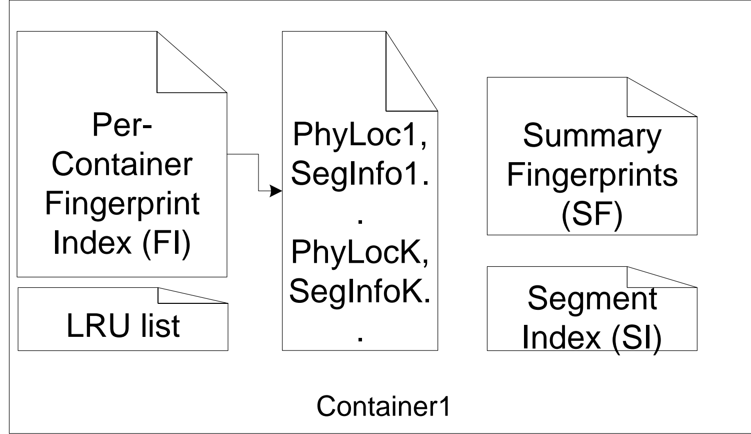


Figure 6.3: *The structure of the segment-based container.*

number of sampled fingerprints for the store segment is larger than 0, the segment is re-inserted to the LRU list to be at the end of segments of that date. For example, if initially the LRU list is $((S_4, D_2), (S_3, D_1), (S_2, D_1), (S_1, D_1))$, where S_i is the ith segment and D_j is the jth access date. Note that three segments at the LRU tail has the same access date. After the sampling rate of S_1 is reduced to half, the updated LRU list is $((S_4, D_2), (S_1, D_1)(S_3, D_1), (S_2, D_1))$, where S_1 moves to the head of the three segments (S_3, S_2, S_1) with the same access date D_1. Otherwise, the segment is removed from the LRU list.

6.2.4 Segment-based Container

Container is meant to group fingerprints with data locality to amortize the I/O cost in loading a container. For incremental backups, data locality is different from that in full backups. D-Dedup correlated segments based on de-duplication history. Initially, logical block addresses from different logical volumes comprise a unified pseudo logical block address by combining logical volume id with the logical block address. For example, for logical block address 300 in logical volume 1, the pseudo logical block address is (1, 300). To accommodate future updates for the container, each container is initially half full. As the de-duplication history data accumulates over time, segments sharing at least a single fingerprint with a store segment are put in to the same container as the store segment.

Figure 6.3 shows the content of each container. The per-container fingerprint index (FI) organizes the fingerprints of the same container into a hash table to enable the lookup of fingerprint values in the container. The result of a fingerprint lookup is the fingerprint's offset in container's fingerprint array. SI maps in-container offset intervals of fingerprints to segments, and vice versa. Each fingerprint can associate with up to K (e.g., K = 2) segments. The intuition is that (1) the latest K segments are the ones most likely to be the basic sharing unit in the future if there is any basic sharing, and (2) it is very unlikely that two segments accidentally share a few

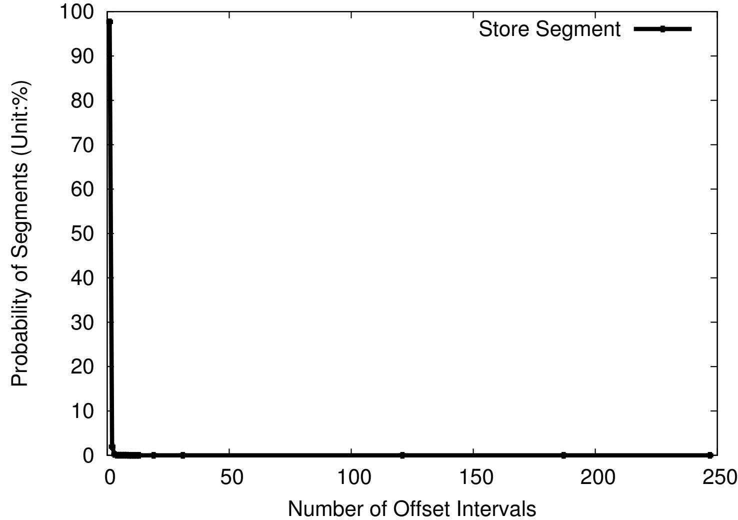


Figure 6.4: *The distribution of offset interval numbers for store segments when* $U_{segment}$ *= 512.*

intermittent fingerprints but not a whole segment. Note that this K is the same as the limit mentioned in subsection 6.2.2.

Each segment can have multiple offset intervals, in theory, the number of offset intervals can reach as large as the number of fingerprints in the segment (i.e., $U_{segment}$). However, more than 90% segments have only one offset interval. Figure 6.4 shows the distribution of the number of offset intervals of segments extracted from the collected trace which will be discussed in details in section 6.4.

As described above, the relationship of segments in a container can be categorized into two categories: (1) two segments are within the same pseudo logical block address range; (2) two segments share fingerprints. If we does not store segment information, fingerprints of these two segments could be placed in the container based on the LRU order. However, because we have the notion of segments, we need to distinguish between these two types.

For segments of type (1), each segment is not further processed and is directly kept in the container. For segments of type (2), segments may need to be split based on the history because each fingerprint can only have up to K (e.g., K = 2) segments. If a fingerprint points to more than K segments as a result of the inter-segment fingerprint sharing, the oldest segment is removed before the new segment is added.

A segment with a larger size than $T_{summary}$ has a summary fingerprint so that all summary structures can fit into RAM. All per-segment structures are put into the summary structures. Each entry in the summary structure is keyed by the segment ID, and its content consists of the summary fingerprint of the corresponding segment.

Each backup logical volume is a de-duplication *stream*. Each de-duplication stream has an *open container*, which accommodates new fingerprints until full. Other than the per-stream open container, any container can accommodate new fingerprints

if one segment in the container shares fingerprints with the input segment. For example, if a input segment $segment_1$ consists of three fingerprints (0xab, 0xdc, 0x12), and a container C_1 has the first two fingerprints but not the third one. $segment_1$ is added to C_1 due to the match of the first two fingerprints, and the third fingerprint 0x12 is added to C_1.

The container's size has an upper bound, denoted as $U_{container}$. A container splits when the container size exceeds $U_{container}$. A LRU list of segments is maintained for each container so that the container splits based on the access history. Each LRU entry represents a segment, and the LRU list is updated when the segment is referenced for de-duplication.

In summary, a new container is created when (1) the per-stream open container is full, or (2) when a container needs to be split. When a new container is created, a 64-bit container identifier is created for the new container. Similar to the segment identifier, a container identifier is 64-bit to ensure its global uniqueness.

6.2.5 Segment-based Summary

A segment that is larger than a threshold, $T_{summary}$, has a summary fingerprint. The summary fingerprint is computed by taking $T_{summary}$ fingerprints as the data content and computing a fingerprint for the content. The property of the summary fingerprint is that a change in one of $T_{summary}$ fingerprints can change the summary fingerprint dramatically. As a result, if the summary fingerprints of two segments agree with each other, the probability that these two segments are different is diminishingly small. In another word, if the summary fingerprints match, two segments are identical.

The segment-based summary index can fit into memory with a proper selection of $T_{summary}$. Each index entry is keyed by the $\langle containerID, segmentID \rangle$ pair. The content is the summary fingerprint, the segment length. As mentioned in subsection 6.2.1, after querying the fingerprint index cache, a containerID and up to K segmentIDs are retrieved. The resulting $\langle containerID, segmentID \rangle$ pair (s) is used to query the segment-based summary index to find out the corresponding segment, and the summary fingerprint of the resulting segment is compared with the summary fingerprint of the stored segment. To reduce the memory consumption due to the segment-based summary index, not all physical blocks in the segment are stored individually. Instead, the physical block ranges are stored to reduce the memory consumption.

For segments that are hit in the segment-based summary index, the information of physical blocks comprising the segment is cached. The physical blocks need to be stored because if the summary fingerprint matches, there is no need to fetch the container and these physical blocks serve as the target physical blocks for data deduplication output.

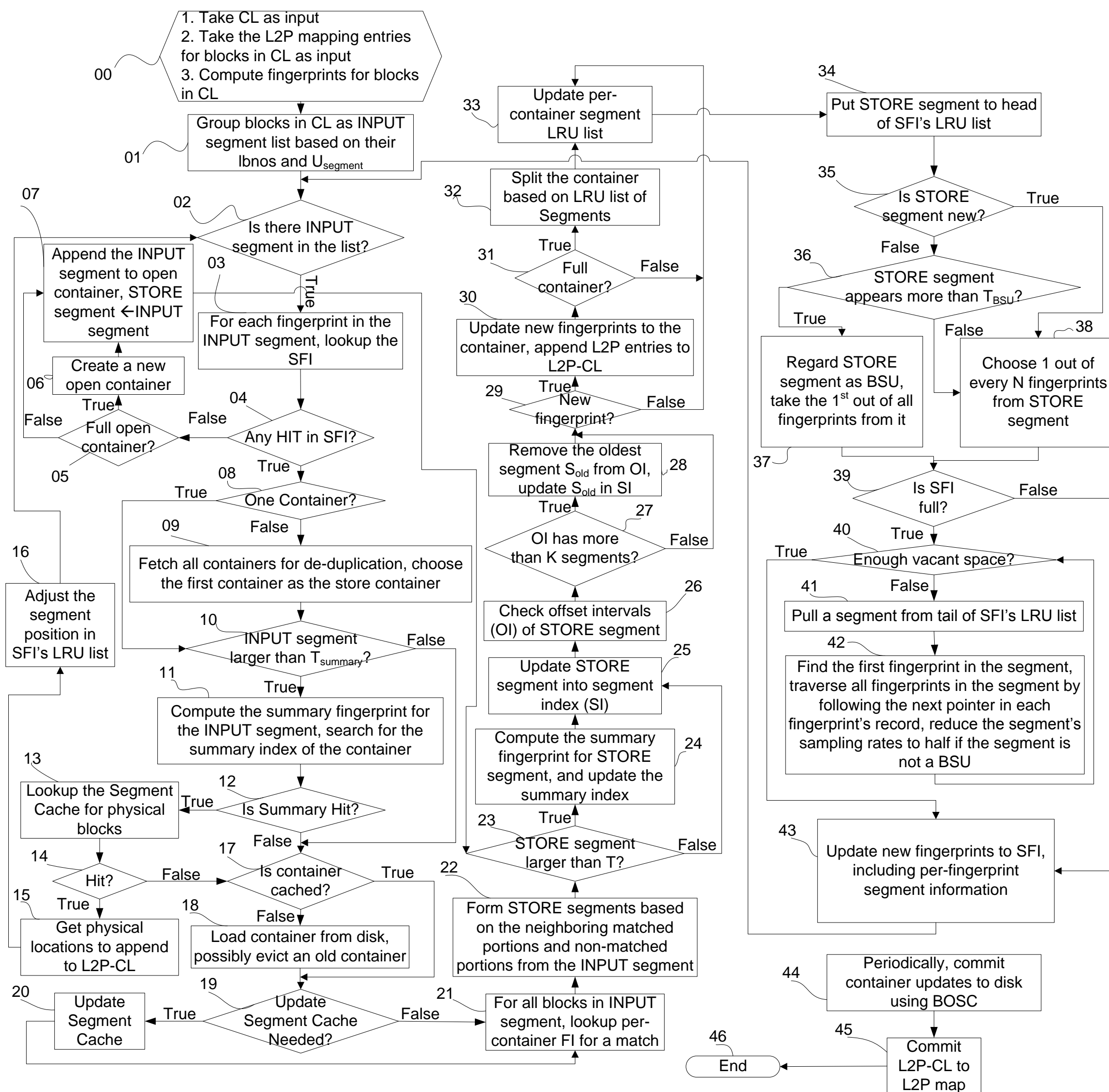


Figure 6.5: *The algorithm of a standalone de-duplication system.*

6.2.6 Put Them Together for Stand-alone De-duplication

Detailed algorithm of a stand-alone de-duplication can be found in Fig 6.5, which combines the segment identification, variable sampling rates and the use of segment summary structures.

At backup time, there are three pieces of input as shown in Step 00. Namely, the incremental changed blocks, the fingerprints of incremental changed blocks.

The first piece of input consists of incremental changed blocks. In concrete, the change list (CL) of the current backup image contains blocks changed in the current backup image since last backup. The before-image list (BIL) includes the previous version of the first overwrite in the current backup image for each overwritten block. BIL is used for garbage collection as described in details in 6.2.7.

Fingerprints of incremental changed block compose the second piece of input. A fingerprint must have low collision rate because two blocks are regarded as identical if their fingerprints are the same in data de-duplication systems [27, 28]. It can be a cryptographic hash value (e.g., MD5 hash or SHA-family hash), or a hash value computed using Rabin's algorithm [190]. For Rabin's algorithm, the probability of two strings r and s yielding the same w-bit fingerprint does not exceed $\frac{\max(|r|,|s|)}{2^{w-1}}$, where $|r|$ denotes the length of r in bits.

The mapping entries of each block from logical block address to physical block address of each logical volume in the form of (L_i, P_i) are the third piece of input. After de-duplication, if a block (L_a, P_a) is a duplicate of previous stored block with a physical block number P_b , P_a can be safely disposed for garbage collection, and an updated entry (L_a, P_b) is inserted into the L2P-CL. The output of the de-duplication is a collection of L2P-CLs for all logical volumes that are de-duplicated.

Initially (Step 01), adjacent blocks are grouped into the same segment. When the segment size exceeds a threshold, $U_{segment}$, a new segment is created to accommodate remaining blocks. These segments are regarded as *input segments*. In contrast, the to-be-stored segment is regarded as *store segment*.

For each input segment, all fingerprints in the input segment are used to query the global SFI (Step 03). If there is no hit in the global SFI, the input segment is regarded as having no stored duplicate. A new store segment is created, where the segment identifier is a globally unique 64-bit value.

The root cause of the miss can vary. It can be that the input segment has no stored duplicate fingerprint, it can also be that SFI does not choose duplicate fingerprints in the input segment as pivot fingerprints. The input segment has to been appended to the per-stream open container. The *store segment* is the *input segment* (Step 07). If the per-stream open container is full, a new per-stream open container is created to accommodate new fingerprints from the input segment (Step 06). A new container identifier is created for the new container. The store segment is further processed in step 23.

If there is any hit in SFI (Step 04), the number of containers the pivot fingerprints point to is checked (Step 08). If there are multiple containers, which means the input segment spans multiple containers, all containers are loaded into the container

cache for de-duplication purpose, but updates are only applied to the first container, including added fingerprints, updated SI and updated per-container LRU list (Step 09). The intuition to store the input segment into one single container is that the input segment reflects the latest status of data locality among data blocks.

In Step 10, the length of the input segment is checked to find out if a summary fingerprint needs to be computed. If so, the summary fingerprint is computed, and the summary fingerprint is is used to query the summary fingerprint index keyed by the summary fingerprint (Step 11). If the summary fingerprint query has a hit, the physical address information of the hit segment is retrieved from the summary segment cache (Step 13). If the physical address information is cached in the summary segment cache, the physical block information is added to L2P-CL (Step 15). And the matched segment is updated to SFI's segment-based LRU list for replacement purpose (Step 16). At this time, the processing of the input segment is done.

If the summary fingerprint index or the summary segment cache is not hit, the corresponding containers need to be accessed (Step 17). If the containers are not cached in the container cache, containers are loaded into the container cache, which can lead to eviction of old containers in the container cache following the LRU order (Step 18). When all involved containers are in the cache, if in step 14 there is no hit in the summary segment cache, the physical address information is extracted from the cached containers and updated to the summary segment cache (Step 19 & 20).

In Step 21, each fingerprint in the input segment is used to query the per-container FI to determined if there are any further match except those pivot fingerprints in SFI. The matched fingerprints may scatter in multiple offset intervals (OI), denoted as *old-OI(s)*, and all non-matched fingerprints compose a new offset interval, denoted as *new-OI*. Old-OI(s) and new-OI are the offset intervals for the store segment, which is the same as the input segment (Step 22). If the store segment has a size larger than $T_{summary}$, the summary segment index is updated with the new store segment's summary fingerprint (Step 24). The store segment and its OIs are updated to the per-container SI as shown in Step 25.

To enforce the limit of segments a fingerprint can have, in Step 26, the algorithm checks old-OI(s) of the store segment. In Step 27&28, for each OI in old-OI(s), if the number of segments exceed K, the oldest segment S_{old} is removed from the OI, and OI is removed from S_{old}, respectively. If S_{old} is empty after the removal of OI, S_{old} is removed from per-container SI. Otherwise, the updated S_{old} is updated to the per-container SI.

If there is new OI, new fingerprints in new OI are appended to the container, and mapping entries of those matched fingerprints are inserted to L2P-CL (Step 30). If the container size in terms of fingerprints exceeds a threshold, $U_{container}$, the container splits based on the per-container LRU list (Step 32). In Step 33, the per-container segment-based LRU list is updated with the store segment. In Step 34, the store segment is also updated to the LRU list of the global SFI.

From Step 35 to 38, the reference count of the store segment is checked to detect a BSU and adjust the sampling rate of the store segment accordingly. From Step 39

and 42, memory space in SFI is spared by reducing the sampling rates of segments at the tail of SFI's segment-based LRU list if SFI reaches its memory limit.

The adjustment of the sampling rates executes until enough memory is spared to accommodate sampled fingerprints from the new store segment. Step 42 describes the way to traverse sampled fingerprints of a segment in the global SFI. Due the memory size limit, there is no separate global data structure to map a segment to its sampled fingerprints. Instead, each sampled fingerprint can point to up to K segments in its content. By traversing from the first sampled fingerprint, which is stored in SFI's LRU list, all sampled fingerprints of the same segment can be visited. Note that a sampled fingerprint A is removed from the global SFI if and only if all segments containing A do not include A in their sampling set.

After enough space is reserved for sampled fingerprints for store segments, sampled fingerprints of the store segment is inserted to SFI (Step 43). At this pointer, the processing of the input segment is finished.

During the processing of the input segments, in Step 44, container updates are committed to disk using BOSC. In Step 45, the L2P-CLs are committed to on-disk L2P maps using BOSC. Details of updates through BOSC can be found in 6.3.

6.2.7 Garbage Collection (GC)

In this subsection, we discuss several traditional garbage collection techniques used in the backup server and proposes a new technique to speed up the garbage collection. Garbage collection is an indispensable component for backup server because some data blocks need to be reclaimed based on the backup retention policies. A data block belonging to expired backup images should be freed if the data block is not shared by other backup images to accommodate new data payload for the future backup image. Finding out such data blocks is the core issue of the garbage collection, and we discuss several approaches below.

Stand-alone Garbage Collection

In the following discussion, each backup image is associated with a logical-to-physical (L2P) map which maps all logical block numbers to physical block numbers holding data for that backup image. The garbage collection tracks the information (i.e., counter or expiration time) of each physical block with a *physical block array(P-array)*.

We denote the newly created backup image as the *current* backup image, and the previous backup image as the *old* backup image in the following discussion. To continue with the discussion, we use an example configuration of the backup server to compare different garbage collection techniques as below. There exist 128 32-TB backup images, 1 PB physical storage scattered over 64 nodes, 32-byte entry for each logical-to-physical mapping, 100 MB/s sequential bandwidth on each node, 5% incremental change from backup to backup, 4 logical volumes, and 8-KB block size.

Each 8-KB block access takes 5 msec on average, and a cached 8-KB block on average serves 16 access in cache.

The first approach, *mark-and-sweep*, first scans all per-snapshot L2P mappings, marks those used physical blocks, and then scans all physical blocks to reclaim those unmarked blocks. The expensive *marking* during the scanning of L2P mappings makes the *mark-and-sweep* approach infeasible for the backup server. For the example configuration, one need to scan 128*4 G = 512 G per-snapshot mapping entries to mark used physical blocks, and to scan 128 GB physical block entries to find out those unused physical blocks. If each per-snapshot mapping entry is 32 byte and each physical block entry is 1 bit, it needs to scan in total 20.48 TB of data, and marks 16 GB physical block entries in a random fashion. The scanning of all L2P mappings can take less than 0.9 hours if all 64 nodes participate fully in scanning the L2P mappings striped over all 64 nodes. If the 16 GB physical block entries reside on the disk, the marking alone can take thousands of hours. We do not assume the physical block entries can fit into memory because the physical raw storage can grow arbitrarily larger than 1 PB. Assuming we use storage area network (SAN) with sustainable sequential throughput of 100 MB/s on each node, it takes at least 0.9 hours to scan all L2P maps.

The *counter-based* approach employs the per-physical block counter to avoid the time-consuming L2P scanning in the *mark-and-sweep* approach. Each time a per-snapshot L2P mapping entry is created, the reference count (RC) of the corresponding physical block is incremented. At the time the snapshot is reclaimed, for each physical block referred by the per-snapshot L2P mapping, the counter is decremented. If the reference count drops to 0, the block is added to a *recycle list (RL)* of physical blocks.

Updating time taken at both the backup time and expiration time of the snapshot is $2 * 4 * \frac{32TB}{8KB} * \frac{1}{16} * 0.005s * \frac{1}{64} = 44$ hours, where 2 means both backup time and expiration time, 4 means 4 logical volumes, $\frac{1}{16}$ means the cache efficiency, and $\frac{1}{64}$ shows the parallelism due to 64 nodes. 44 hours are prohibitive because 44 hours are more than a day, which is the interval of daily backup.

At the garbage collection time, blocks in RL are reclaimed. Because RL consists of physical blocks pertaining to reclaimed backup images, it is much smaller than the whole physical block space, and takes much less to scan. For example, as the example configuration shows, the whole physical block list can contain 128 G entries and takes 224 GB disk space (14 bits for counter), it takes 35 seconds to scan the whole physical block list if the list is striped over 64 nodes. But it takes 2 seconds if 5% of physical blocks have a zero reference count.

An *ET-based* approach can further reduce the number of updates at the backup time by replacing the counter with an expiration time, but the incremental change list methodology can not be used for *expiration-time-based* scheme because the expiration time of overwritten blocks in the *old* backup image does not change. In concrete, at the creation time of a backup image, the expiration time of each physical block referred by the *current* backup image is updated.

Updating time taken at both the backup time and expiration time of the snapshot

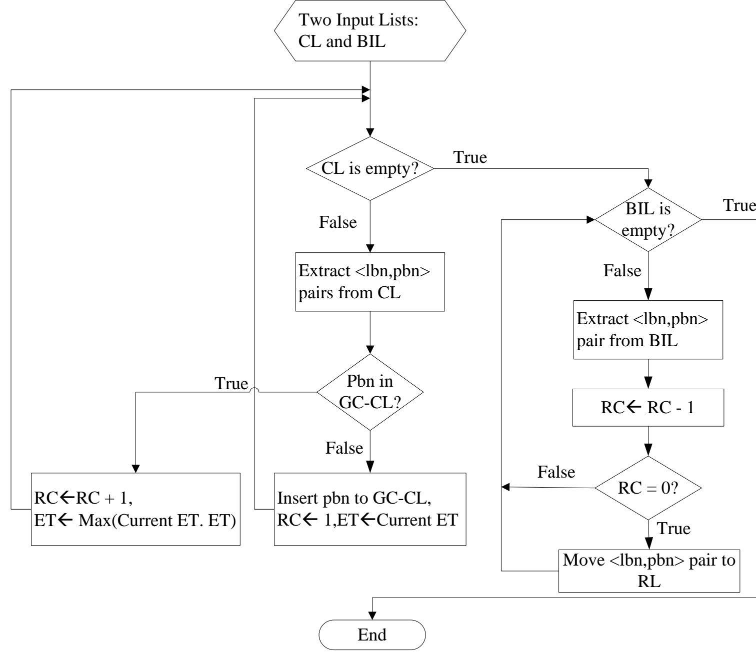


Figure 6.6: *The flowchart of metadata updating for garbage collection at backup time.*

is $4 * \frac{32TB}{8KB} * \frac{1}{16} * 0.005s * \frac{1}{64} = 22$ hours, where 4 means 4 logical volumes, $\frac{1}{16}$ means the cache efficiency, and $\frac{1}{64}$ shows the parallelism due to 64 nodes. 22 hours are prohibitive for daily backup.

At the garbage collection time, all physical block entries are scanned to find out those expired blocks based on their ETs. Take the example configuration shown above, if each ET takes 10 bits, it takes 25 seconds to finish the garbage collection.

The *counter-based* and *expiration-time-based* approaches both shift the burden from the garbage collection time to the backup time. The time spent during garbage collection is greatly reduced, but the overhead at the backup image creation time can be overwhelmingly large. Take the example configuration, for each 32 TB snapshot creation, on average 4 G physical blocks mapped by the *current* backup image are updated regardless whether the block has changed or not. If each physical block update takes 5 msec as the physical blocks can scatter over the physical space due to de-duplication, it can take up to 86.8 hours. Although the BOSC scheme can aggregate updates and mitigates the update overhead, it is still prohibitive to have such a large volume of updates at the backup time.

To reduce the volume of updates at the backup time, we propose a brand-new *mixture* GC scheme by lazily updating the metadata of physical blocks with both counter and expiration time information for each physical block. Instead of updating all physical blocks pointed by a snapshot image, only those physical blocks pointed to by changed logical blocks are updated.

At the backup time, there are two lists as the input: CL and BIL as mentioned in 6.2.1. Physical blocks referred in CL increment their RC, and update ET accordingly. Those physical blocks referred in BIL decrements their RC. All these changed physical blocks are added to a Garbage Collection related Change List (GC-CL), which is an incremental list sorted by the physical block number to speed up the updates to GC-CL.

GC Alternatives	**Backup Time Operations**		**Garbage Collection Time Operations**		**Elapsed Time**
	Lookup	**Update**	**Lookup**	**Update**	
Mark-and-Sweep	None	None	Scanning of per-backup logical-to-physical mapping	Bit-array updates	0.9 hour GC
Counter-Based	None	RC changes of physical blocks of L2P mappings	Scanning of RL	None	44-hour backup, 2-second GC
ET-Based	None	ET changes of physical blocks of L2P mappings	Scanning of ET of all blocks	None	22-hour backup, 25-seconds GC
Mixture	Scanning of all per-block queues for all physical blocks	RC and ET changes of overwritten physical blocks	Scanning of RL	None	18-second backup, 3-second GC

Table 6.1: *Comparison of 4 different GC techniques. The example configuration is as below: 128 32-TB backup images, 1 PB physical storage scattered over 64 nodes, 32-byte entry for each logical-to-physical mapping, 100 MB/s sequential bandwidth on each node, 5% incremental change from backup to backup, 4 logical volumes, 8-KB block size. For regular cache, a block can be accessed 16 times before eviction.*

Physical blocks referred in BIL decrement their RC. If RC drops to zero, the physical block is moved to the recycle list (RL). At the garbage collection time, physical blocks in RL are checked for their ET. Those blocks that have expired are garbage collected.

Because the size of GC-CL is proportional to the size of incremental changes and incremental changes are smaller compared to the full block set, the proposed garbage collection mechanism is scalable to daily changes. Furthermore, if GC-CL cannot fit into RAM on one node, BOSC can be employed to improve the update performance.

The expiration time is updated when a physical block is referenced due to deduplication, the expiration time is updated as the latest expiration time between

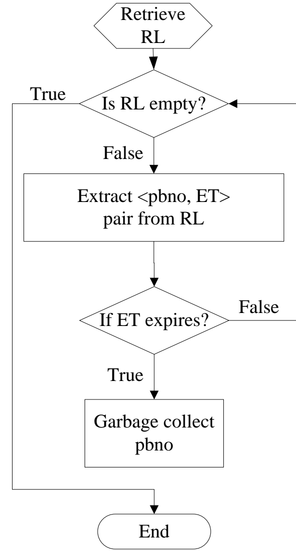


Figure 6.7: *The algorithm of stand-alone garbage collection.*

the stored expiration time and the expiration time of the snapshot containing the de-duplicated block.

The counter is updated in two cases: (1) When a physical block P is referenced due to data de-duplication, the counter is incremented for P, and (2) when a physical block P belongs to BIL of a snapshot, the counter is decremented for P.

RL is an incremental list, it is initialized as NIL because initially there is no de-duplication among main-storage volumes. This incremental list can be used to find out data blocks to garbage collect.

The detailed flowchart of the markup for garbage collection is shown in Fig 6.6. Fig 6.7 shows how garbage collection proceeds based on the RL. Basically, all physical blocks in RL are checked to recycle those blocks that have expired.

Figure 6.8 shows an example of how garbage collection works. At the backup time, CL and BIL of each snapshot are used to update GC-CL: Backup A is an initial backup and there does not exist L2P mapping yet for all logical block addresses (in total 12 logical blocks). Only logical block 12 has its corresponding physical block address, 700. Backup image A has the expiration time of 700. For backup B, logical block address 1, 2, and 7 are written. CL records the written physical blocks (320, 321, and 440). Note that the expiration time of all three physical blocks is updated as 600, the expiration time of backup B. For backup C, logical block address 1, 2, and 9 are written. Note that logical block 9 shares the same physical block (physical block 321) as the old version of logical block 2. The expiration time of physical block 321 is updated to 750, the expiration time of backup C. For backup D, logical block address 4, 5, and 9 are overwritten. Note that RC of physical block 321 drops to 0.

At the garbage collection time (after backup image D is created), those entries in GC-CL with zero RC are extracted to form RL. In this particular example, physical block 320 and 321 are included in RL. Physical blocks 320 and 321 can be recycled

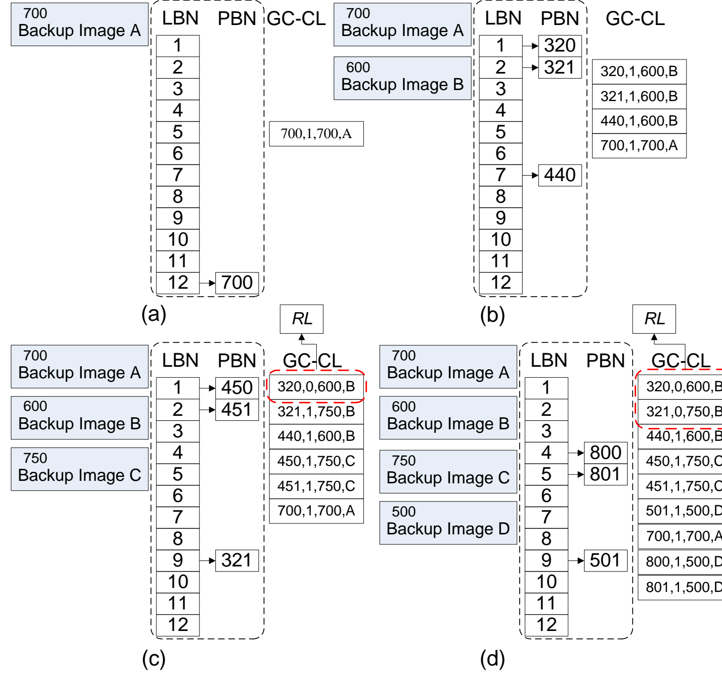


Figure 6.8: *An example of updating GC-CL and RL.*

at time 600 and 750, respectively.

The BOSC scheme can be employed to improve the update throughput for GC-CL. Using the same example for the Mixture scheme, at backup time, if GC-CL can not fit into memory, BOSC can be employed to improve the update throughput with $64 * \frac{1}{4}$=16 GB BOSC buffer. Each P-array entry is 3-byte (10 bits for ET, 14 bits for counter) GC-CL can be $3 * \frac{1PB}{8KB}$=384 GB, block size for BOSC is 512 KB, average queue length $\frac{\frac{16GB}{3B}}{\frac{384GB}{512KB}}$=7K BOSC throughput is $64 * \frac{7K}{\frac{512KB}{(\frac{100}{2})MB/s}} = 44$ M Request/Second. Total time taken at the end of the day is $4 * \frac{\frac{32TB*0.05}{8KB}}{44M}$ =18 seconds. At GC time, blocks in RL accounts for around 5% of all physical blocks because each day at most 5% of all physical blocks get overwritten, the sequential scanning of the RL can take $\frac{3*\frac{1PB}{8KB}*0.05}{64*100MB/s} = 3$ seconds.

Table 6.1 shows 4 alternative schemes in garbage collection and compares the overhead at the backup time and the garbage collection time. For the given example configuration, it turns out that *Mixture* scheme has the minimal overhead at the backup time.

Parallel Garbage Collection

Because a particular hash value resides on one data node and a physical block is represented by its hash value, the $\langle RC, ET \rangle$ pair of a particular physical block is associated with a fingerprint. The physical block is distributed to a particular data

node based on the consistent hash of the fingerprint.

Fig 6.9(a) shows how GC-CL and RL are distributed to participating parallel nodes based on the consistent hashing of the fingerprint of the physical block.

Figure 6.9(b) shows an example of how to distribute GC-CL in Fig 6.6 to 4 participating nodes: All physical blocks in GC-CL have their fingerprints computed (a fingerprint is a hash value of the block content). Each fingerprint is long enough to have a very low collision rate. For example, the fingerprint can be 20-byte long. Each fingerprint is then mapped through consistent hashing to 1 of 4 participating nodes. In this example, Node 1 accommodates physical blocks 440 and 700. Node 2 accommodates physical blocks 320 and 800. Node 3 accommodates physical blocks 321, 501 and 801. Node 4 accommodates physical blocks 450 and 451. Note that after the distribution, each node can independently garbage collection physical blocks allocated to it. For example, Node 4 is responsible to garbage collection physical block 450 and 451.

6.2.8 Parallel De-duplication

If a single server does not have enough resource to de-duplicate input data blocks with good DDR and de-duplication throughput, the task of de-duplication can be distributed to multiple storage nodes. A distributed secondary storage system with multiple slave nodes fits the purpose well. For distributed de-duplication, the node coordinating the de-duplication procedure among multiple storage nodes is denoted as the *master node*, and the distributed storage node is denoted as *slave node*.

In this section, we will discuss (1) how to distribute related data structures from a master node to multiple slave nodes, and (2) how to accomplish data de-duplication tasks in a distributed fashion. Distributed data structures include the segment-base container and its related data structures, the segment-based summary, and garbage collection related data structures. Garbage collection related data structures will be described in section 6.2.7, and we will omit the garbage collection related data structures in this section. Data de-duplication tasks to be distributed include segment identification, variable fingerprint index management, and distributed container management.

The design presented in this section aims to de-duplicate backup images in an out-of-line fashion. In another word, the backup images are first stored in the secondary storage system without de-duplication. Periodically, the de-duplication functionality is invoked to detect duplicates and garbage collect physical blocks. However, the design principles can apply equally to inline de-duplication systems without significant changes.

Figure 6.10 shows the architecture of the parallel de-duplication. We will show how the de-duplication procedure works on the master node and the slave node, respectively.

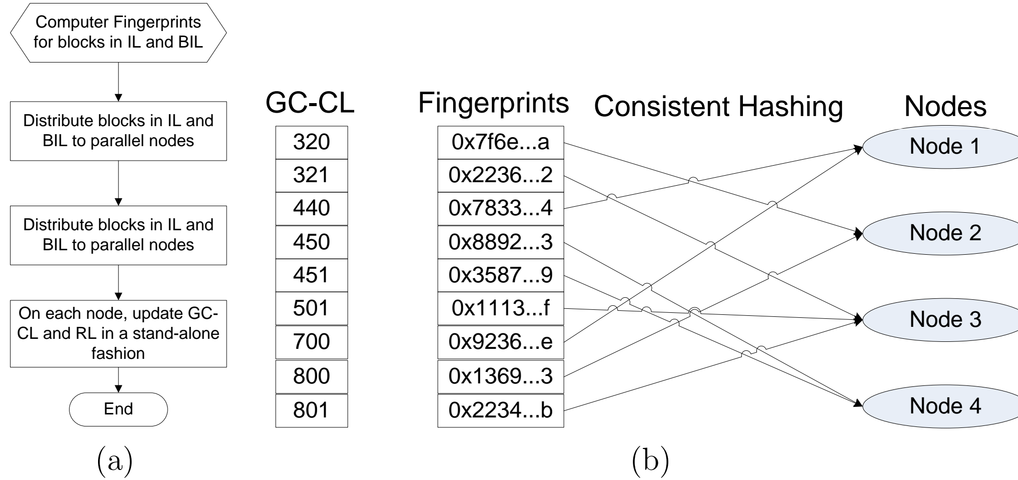


Figure 6.9: *(a) The flowchart of parallel metadata updating for garbage collection. (b) An example of distributing RL and GC-CL.*

Daemon at the Master Node (M-Daemon)

Container Distributor is responsible to distribute container content to slave nodes. Fingerprint entries in the container are distributed to related slave nodes using the same consistent hashing function as that of distributing fingerprints by the Container Distributor. These entries of the same container on a particular data node are organized as a sub-container file. Formally, the container distributor stripes a conceptual container to sub-containers using the well-known consistent hash so that sub-container files, per-container FI, per-container SI and the container cache of the same fingerprint reside on the same data node. The container distributor is also responsible to distribute the segment summary based on the same consistent hashing of summary fingerprints.

Note that the Container Distributor does not write the sub-container file directly. Instead, the Container Distributor gives each sub-container file on data node a well-known file name so that sub-container files belonging to the same container can be retrieved together. For example, if a container file has the identifier 0x01234, four sub-container files are named as 0x01234.ip_1, 0x01234.ip_2, 0x01234.ip_3, and 0x01234.ip_4, respectively. When Container Distributor retrieves container 0x01234, it asks each individual data node for corresponding sub-container files. Each data node in turn reads the sub-container file 0x01234.ip_i, where ip_i is the IP address of that particular data node. The benefit of writing sub-container files on data nodes is that at the retrieval time, each sub-container file has a local copy and the data node can retrieve the sub-container file from its local disk instead of from other data nodes through network. Fingerprint Distributor hashes each individual fingerprint to determine a destination slave node, S_i. In fingerprint distributor, M-Daemon waits for response from all involved slave nodes. The output is the target physical block pointed by a logical block address. There can be three possible cases: (1)the fingerprint has

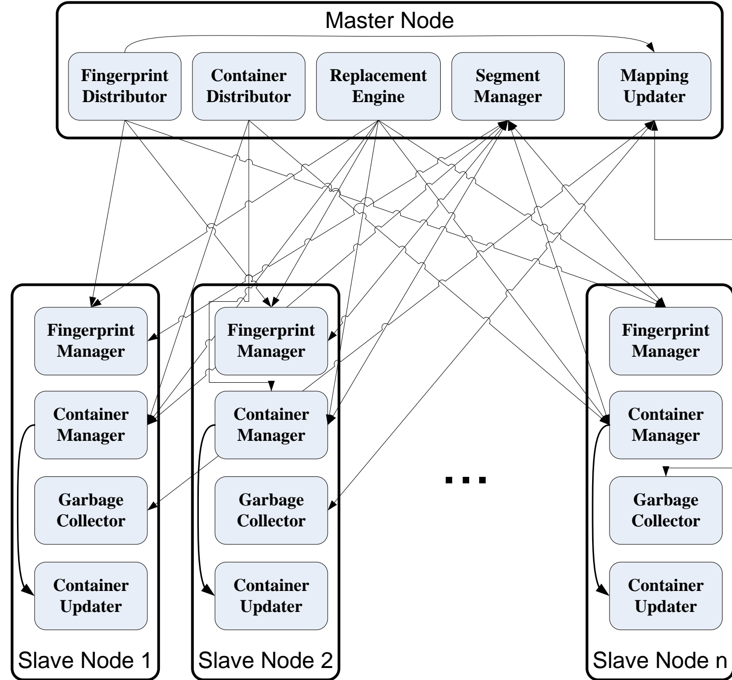


Figure 6.10: *The architecture of parallel data de-duplication.*

a hit in SFI on the slave node, and the container is cached in the container cache or the segment-based summary index is hit. In this case, the response includes the information of the target physical block, the store segment and related SI. The original physical block can be disposed. (2) The fingerprint has a hit in SFI, but the container is not cached in the container cache and the segment-based summary index does not match. The container is loaded into RAM of the slave node. The entry corresponding to the queried fingerprint is extracted to get the target physical block by querying the per-container FI, and store segments are formed and updated to the per-container SI. (3) The fingerprint does not have a hit in the fingerprint cache. Fingerprint are appended to the corresponding container, and store segments are formed and updated to the per-container SI. The original physical block is used as the target physical block.

Replacement Engine determines the replacement policy of containers and the global SFI. M-Daemon stores the LRU list for SFI and container cache to determine which fingerprints to evict from parallel SFIs and which container to evict from parallel container cache on M-Daemon. The LRU list for SFI is based on segments, not individual fingerprints. The LRU list for the container cache is based on containers.

Segment Manager has two functionalities: (1) Determining the destination container of the input fingerprints on the fly. For each input segment, initially there is no container corresponding to the input segment If the de-duplication response indicates that the input segment belongs to a container C, C is the corresponding container for the input segment. Otherwise, a per-stream new container is created to accommodate the input segment. A stream is defined as the backup data from an

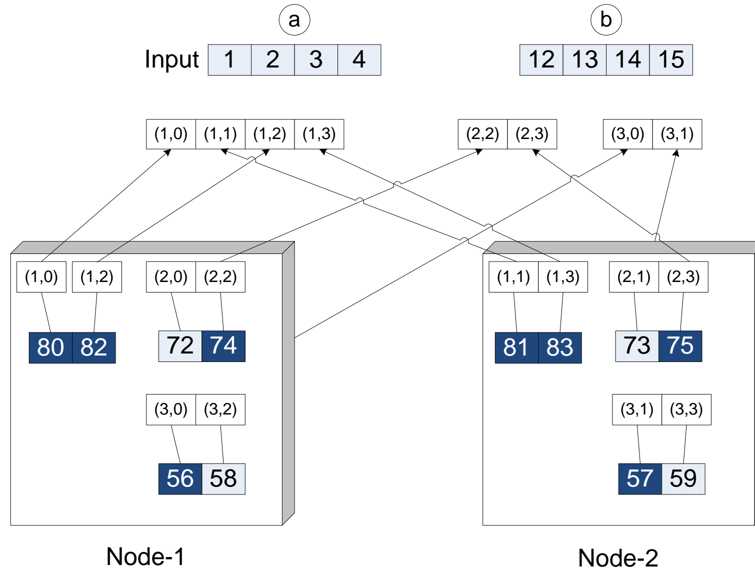


Figure 6.11: *An example of distributed segment identification.*

end user or a backup logical volume. (2) Identifying segments. Segment identification is the same as the stand-alone case except the following: per-container SI is split so that fingerprints in an offset interval of the same segment in a sub-container forms a smaller offset interval, and the method to reconstruct a segment. To deal with split of per-container SI, each sub-container holds the sub-SI consisting of offset intervals mapped that slave node.

Because containers are distributed on all participating slave nodes, store segment information is not stored on a single slave node. Instead, the offset within a segment is stored on participating nodes. After all offsets and segmentIDs of a segment are collected from participating slave nodes, the information of stored segment need to be re-constructed.

Figure 6.11 shows an example of reconstructing a segment from two participating nodes. The de-duplication scenario is the same as that in figure 6.2. Segment 1, 2, and 3 all belong to the same container. The parenthesized pair (n, k) represents the kth physical block in the nth segment. Segment 1, 2, and 3 are distributed to two slave nodes evenly in the example. After querying the per-container FI, offsets and segmentIDs are returned to S-Daemon. On M-Daemon, the matched portions of segment 1, 2 and 3 are reconstructed and the segment identification algorithm mentioned in subsection 6.2 can continue.

Mapping Updater updates the mapping metadata (from logical block address to physical block address) of the volume based on the de-duplication output (L2P-CL). The mapping updater works with the garbage collectors on parallel slave nodes to protect the consistency of the L2P from node crashing. Detailed algorithm can be found in the garbage collector on S-Daemons.

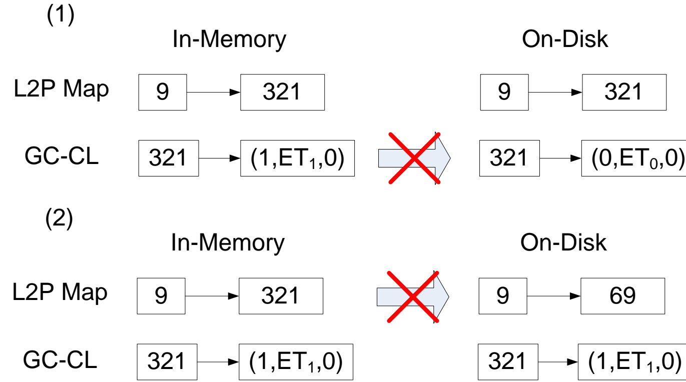


Figure 6.12: *An example of metadata data inconsistency between GC-CL of garbage collector and L2P of mapping updater.*

Daemon on the Slave Node (S-Daemon)

Fingerprint Manager manages the fingerprint cache except the replacement of fingerprints. Replacement of fingerprints is coordinated by the Replacement Engine on M-Daemon.

Fingerprints are organized as a hash table. Each fingerprint entry has a container identifier list with up to K entries, the offset of the fingerprint within the container, and a pointer to fingerprints of the same segment for replacement purpose. With the pointer, all fingerprints of the same segment are organized as a single linked list. The head of the linked list is the first sampled fingerprint of the segment.

In Replacement Engine, each LRU list entry for SFI consists of the fingerprint value of the first sampled fingerprint of the corresponding segment, and the container identifier. Each LRU list entry for the container cache has the container identifier as the key.

When fingerprints of a segment are evicted from the SFI, these entries are written out to disk as a persistent copy of the SFI.

Container Manager manages the container cache except the replacement of containers. Container Manager has three parts: (1) container cache, containers are organized as a hash table keyed with the unique container identifier the same as the container cache mentioned in subsection 6.2.4. (2) container format, it is the same as the container format mentioned in subsection 6.2.4. (3) container reader, it reads the sub-container file from underlying file system with the ContainerID + IP as the file name.

Garbage Collector works as described in 6.2.7, the garbage collector on each slave node recycles physical blocks assigned to the slave node. The output of the garbage collection is a incremental list, GC-CL. Because both GC-CL and the L2P map are written to disk, the consistency of the updates of these two data structures is critical to the success of garbage collection.

Figure 6.12 shows two examples of inconsistency between GC-CL used by garbage

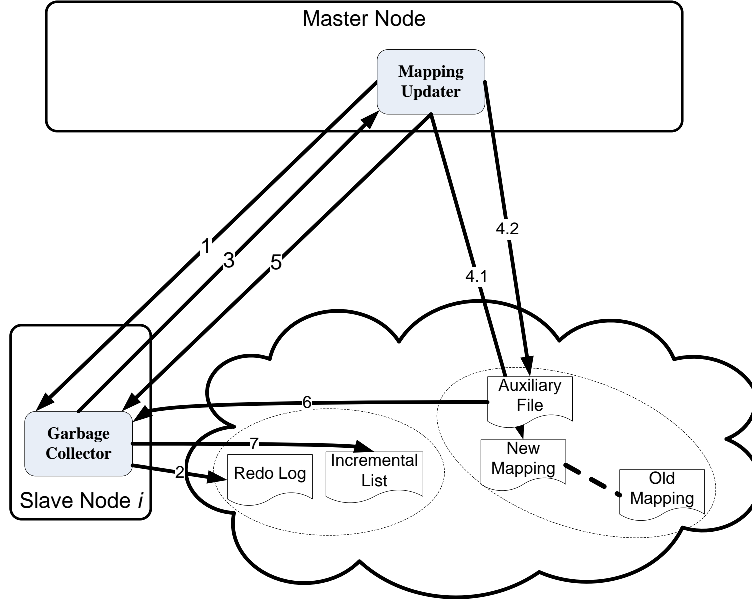


Figure 6.13: *The algorithm to ensure the consistency between garbage collection and L2P mapping updater.*

collector and the L2P map used by mapping updater. These two data structures have an in-memory version and an on-disk version. A failure of the data commit to the disk can lead to data inconsistency. In example (1), the update to L2P map, $9 \rightarrow 321$, is committed to disk successfully. However, the update to GC-CL, $321 \rightarrow (1, ET_1, 0)$, fails to be committed to disk. As a result, the physical block 321 will be recycled after ET_1, which can corrupt the in-use physical block 321.

In example (2), the update to GC-CL, $321 \rightarrow (1, ET_1, 0)$, is committed to disk successfully. However, the update to L2P map, $9 \rightarrow 321$, is not committed to disk due to crash. As a result, physical block 321 in GC-CL has no chance to decrement its reference count and will not be recycled, which defeat the very purpose of garbage collection.

The root cause of the metadata inconsistency is that the L2P map and the GC-CL are not updated atomically. A general solution is to group updates of the same physical block to the L2P map and the GC-CL in a transaction. However, the transaction can be expensive because each individual update has to be committed to disk before the transaction is completed. To reduce the performance overhead due to transaction, we relax the transaction definition by batching updates to GC-CL and L2P map and committing updates to disk when the batch exceeds a threshold. When a batch is committed to disk, a *consistency point* is reached. If a crash happens during the commit of the batch, written updates in the batch are rolled back to previous consistent point, and all pending updates are re-committed to the L2P map and the GC-CL.

Figure 6.13 shows the step-to-step procedure of the batch-based transaction. An

auxiliary file is created to mark the start of the commit of a batch, and it is assumed that the creation of the auxiliary file is atomic. Fortunately, the assumption holds for most distributed file system. In the following discussion, the mapping updater and the garbage collector communicates through network message, and the network message can get lost at any time. A time-out mechanism is employed to tell if a message gets through successfully or not.

In step 1, the mapping updater on the master node asks the garbage collector on the slave node to write updates of GC-CL to a redo log via a network message. In step 2, the garbage collector on the slave node writes updates of GC-CL to a redo log. In step 3, all updates of GC-CL are written successfully to the redo log, and the garbage collector replies to the mapping updater through a network message. If the network message does not go through, the mapping updater does nothing. Otherwise, the mapping updater goes to step 4.

In step 4, the mapping updater first writes the new L2P mapping, and then an auxiliary file to the distributed file system. The failure of either writes is regarded as a failure of the update. The mapping updater will find it out later on in step 6. If both writes succeed, the mapping updater goes to step 5.

In step 5, the mapping updater notifies the garbage collector on the slave node about the success of the writes of the new L2P mapping file and the auxiliary file through a network message. If the network message is delivered successfully, the garbage collector goes to step 7. Otherwise, the garbage collector will later find out the existence of the auxiliary file in step 6.

In step 6, the garbage collector checks the existence of the auxiliary file to decide if the garbage collector can merge the redo log with the GC-CL file. If the garbage collector crashes in step 6, after the garbage collector reboots, the first thing it does is to check the existence of the auxiliary file. Otherwise, the garbage collector goes to step 7.

In step 7, the redo log and the GC-CL file are merged by the garbage collector. Periodically, the old mapping file is replaced by the new mapping file. And the auxiliary file can be removed after a period large enough so that the corresponding GC-CL and redo log have been merged before the removal of the corresponding auxiliary file.

Container Updater is responsible for the writing of sub-container files on the local data node. To speed up the writing, updates are sequentially written to disk using BOSC. The buffer memory for BOSC is separately allocated from the container cache.

Figure 6.14 shows the detailed algorithm of parallel de-duplication. In the algorithm, the LRU lists of SFI and of container store is stored by M-Daemon, all other data structures are distributed to S-Daemons.

The input of parallel de-duplication is the same as that of standalone one as shown in Step 00. The way to compose input segments is also the same as shown in Step 01. In Step 04, each fingerprint in the input segment is distributed to S-Daemon via the consistent hashing. In Step 05, fingerprint is used to query parallel SFI on each slave node. If there is no hit in parallel SFI, a bit in the input segment's

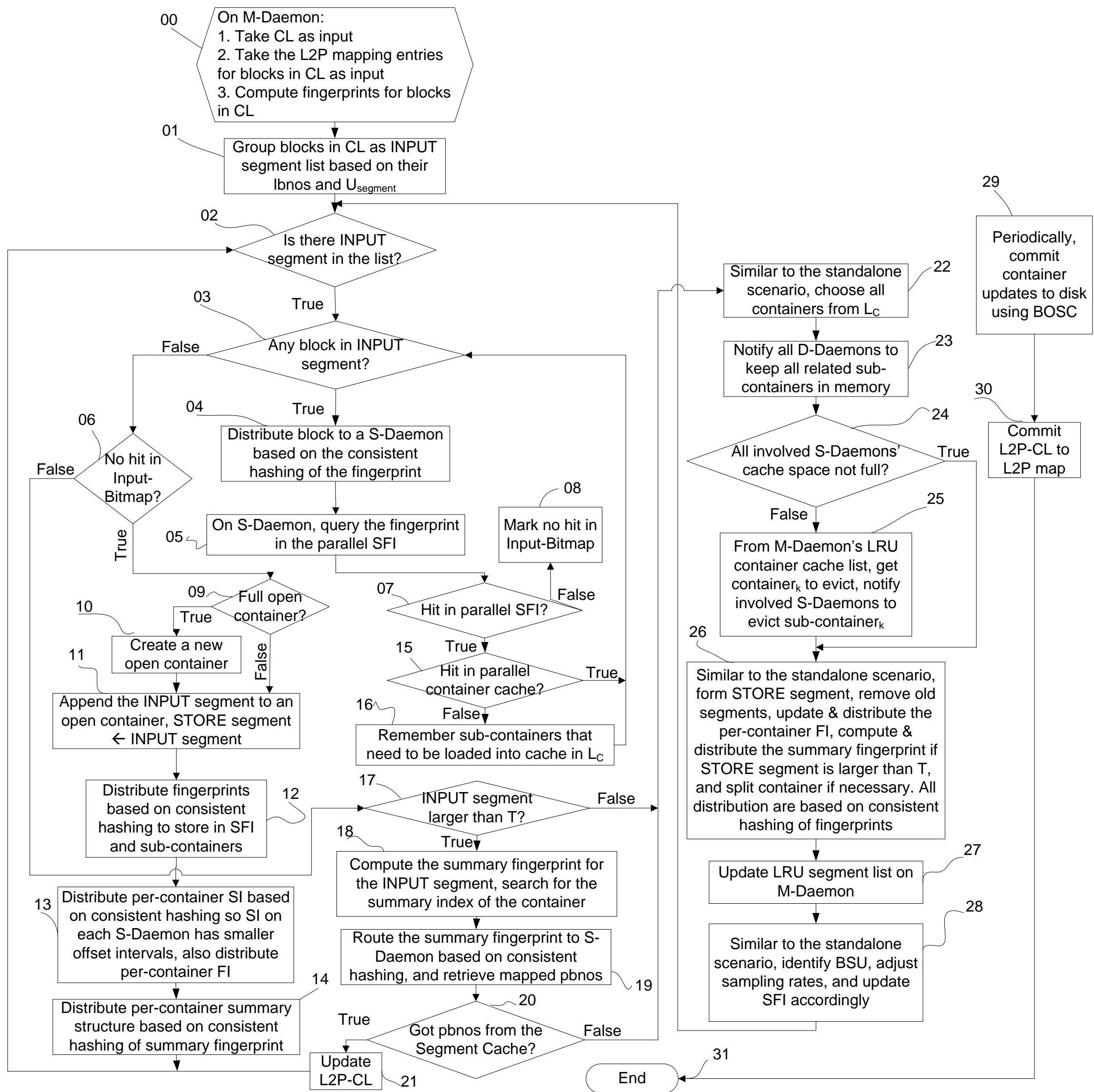


Figure 6.14: *The algorithm of distributed data de-duplication.*

bitmap of the input segment is reset to indicate that there is no hit for the particular fingerprint (Step 08).

In the destination container is not cached in the container cache on the slave node, the sub-container information is inserted to a list L_C.

After all fingerprints in an input segment are checked, the input bitmap is checked. If all bits are unset, the input segment is a new segment. The segment is appended to the per-stream open container (Step 09). Similar to the standalone algorithm, Step 10&11 create a new per-stream open container if the container is full and set the store segment as the input segment. In Step 12, fingerprints are distributed to slave nodes by the fingerprint distributor. In Step 13&14, the per-container SI, per-container FI, and summary fingerprints are distributed to slave nodes through container distributor.

If there is hit in parallel SFI according to the input bitmap, Step 17&18 computes the summary fingerprint if the input segment is larger than $T_{summary}$. In Step 19, the summary fingerprint is distributed to a slave node by the container distributor. In Step 20, the algorithm checks if there is hit in the summary fingerprint index and summary segment cache, the physical address information can be retrieved and the processing of the input segment is finished.

Otherwise, all containers in L_C needs to be loaded for all involved S-Daemons (Step 22&23). If the result of loading containers evicts a victim container $Container_k$, M-Daemon notifies all involved S-Daemons to evict $subContainer_k$. In step 26, the segment identification, updates of per-container SI and FI, updates of summary fingerprint, container split are similar to the standalone algorithm except all data structures are distributed to all involved slave nodes as mentioned in section 6.2.8.

In Step 27, M-Daemon maintains the global LRU segment-based list for the global SFI. In Step 28, the global SFI is updated and the BSU is identifier the same as that of the standalone algorithm.

Periodically, container updates and L2P-CLs are committed to disk using BOSC, the same as those in the standalone algorithm.

6.2.9 Optimizations

In *file-level* de-duplication system, if a file is modified, the file is added to the change list. At the incremental backup time, all files in the change list are transferred to the backup storage system. However, even changes to a small portion of a file makes the whole file to be backed-up. For example, for a large file of 10 MB in size, a change of a 4 KB page in the file will put the whole 10 MB file in the file change list, which causes more than 3 magnitudes of backup volume overhead.

BCT tracks changed blocks instead of changed files to remove the overhead of transferring the whole file at the backup time. Oracle's recovery manager (RMAN) [189] supports block change tracking, where a dedicated change tracking file is used to record the block changes since last backup operation. In contrast, Fanglu et al [188] proposed a transparent change block tracking mechanism for modern file systems (i.e.,

ext3) by regarding all written blocks as changed blocks. In this technical report, we advocates Fanglu's approach for its simplicity. The BCT can be implemented as a device driver intercepting all writes to the block device. If we can extract the metadata information about the file system above, we can further divide written blocks to data blocks and metadata blocks. Metadata blocks are those blocks that fall in the well-known range of file system metadata, including Inode table, Inode map, super blocks, data block map, free map, etc. All other non-metadata blocks are data blocks, including the directory blocks.

The BCT technique provides a promising start point to save resources used in data de-duplication systems, including storage space, network bandwidth, and power consumption. Our proposed de-duplication techniques employs the BCT as the corner stone to build the de-duplication capability. On one hand, because only changed blocks are used to de-duplicate with each other, it is not clear if current de-duplication techniques [27, 28] can still perform well in terms of de-duplication quality and de-duplication throughput. On the other hand, because block changes have strong spatial and temporal locality, it is worthwhile to craft new techniques to fully explore the spatial and temporal locality exposed by such incremental backup streams.

To better explore the data locality among changed blocks, it is desirable to have file-level information because the data locality is specific to the file containing the blocks, not the block itself. However, it is cumbersome to retrieve file-level information at the block layer without intrusive changes to the file systems. Metadata inference engine (MIE) [29, 68] closes the gap between the non-intrusive changes to file systems and the retrieval of detailed file metadata by snooping block-level traffic.

Instead of associating blocks with the corresponding file Inodes at run time with *inference*, we can associate blocks with Inodes in an offline fashion. In concrete, at the end of each backup, we take a snapshot of the file system metadata. For ext2/ext3, the debugfs tool can be used to extract the location of Inode table and Inode map. At the backup time, the current Inode table/map is compared with the previous Inode table/map byte by byte to detect Inode changes. The backup volume is remounted to ensure the cache is cold. Changed Inodes are examined by a user-level program to scan blocks belonging to them, which can associate the data blocks with corresponding Inodes. Note that in examining the Inode, the offset of each block can also be established.

The offline analyser can enumerate file offsets for all live blocks but not for those dead blocks which do not belong to any file. Dead blocks can not be accessed by any file Inode and these changed blocks are attributed to a special "dead" file. For the purpose of de-duplication, we anticipate the "dead" file is effective in de-duplicating dead blocks from all backup clients.

With the help of MIE, the de-duplication engine on the backup server side has more flexibility to de-duplicate the backup streams for two reasons. First, file-level information is more accurate to capture the data locality in segment identification. For example, instead of blindly de-duplicating against adjacent contiguous blocks, it

is more efficient to de-duplicate against blocks of the same file.

6.3 BOSC in De-duplication

Our de-duplication system employs BOSC to improve the update performance in 3 metadata update scenarios. First, at the backup time, in updating the fingerprint container when new fingerprints are inserted into the container or per-container segment index is updated, the proposed de-duplication system uses BOSC to commit these updates. In concrete, updates to fingerprint containers are batched in individual per-block queues and are committed to the underlying disk in an I/O efficient manner. The updates to the per-file fingerprint container and the corresponding cached fingerprint container can be combined together as the unified cache of the fingerprint container. At the commit time, only the updates are committed to the underlying disk. Note that the fingerprint container cache is organized based on the starting physical location of the fingerprint container,the fingerprint container cache does not disrupt the order of per-block update queue scanning in the BOSC scheme.

Second, at the backup time, in the *Mixture* GC scheme, backup-time updates to the L2P map employs BOSC to maximize the update performance. Although updates to the L2P map exhibits strong data locality, BOSC can improve the update throughput by decoupling the updates from I/O operations.

Third, at the backup time, in updating GC-CL, as the whole GC-CL is likely to be disk-resident, BOSC can be used to speed up the updates to GC-CL.

At the backup time, a unified logging in BOSC can be shared by these three BOSC tasks. That is, updating to the file-based fingerprint container, updating to the per-logical-volume L2P mapping entries, and updating to GC-CL can share the same logging device used by BOSC. These two types of updates are differentiated by a field in the logging record indicating which BOSC task the logging record is for.

6.4 Performance Evaluation

In this section, 4 design decisions of the proposed data de-duplication system for incremental backups are examined with a trace collected under a typical enterprise environment. The design decisions include (1) segment identification, (2) segment-based variable frequency fingerprint index, (3) segment-based container, and (4) segment-based summary. I will first introduce the evaluation methodology, then present results for all 4 design decisions. Finally, I will present the analytical result for the proposed scalable garbage collection technique and its comparison with other alternative garbage collection methods.

File Category	Conjectured Write Method	Contribution Ratio (Unit:%)
VM-Related Files	append	35.1
Multimedia Files	write	21.6
System Files	write	21.9
User Documents	write	11.5
Installation Media	write	5.1
Log Files	append	1.6
Mails	write	1.6
Database Files	write	1.6

Table 6.2: *File types, conjectured update methods, and their percentage in the trace.*

6.4.1 Evaluation Methodology

To evaluate various design decisions, it is critical to have real-world traces that keeps recording incremental changes of logical volumes from multiple users for a long period of time. For this purpose, we developed a user-level program to track changed files of individual end users, and deployed the program to 23 regular employees of a research institution to collect their daily changed files for 10 weeks. We use the trace to drive the evaluation of our design decisions. We will refer the collected trace as ***Workgroup*** in the subsequent discussion. We understand the collected trace does not fully represent incremental backups under all environments. However, we do believe the evaluation based on the ***Workgroup*** trace provide valuable insight for incremental backups.

In this section, we first present the details of collecting the trace, then show methods in analysing the ***Workgroup*** trace, and finally discuss the metrics and parameters in analysing the trace.

Trace Collection

Ideally, a trace for incremental backups should contain block-level writes to a secondary storage system in a daily fashion. Unfortunately, we do not have a block-level backup system to collect the trace. Instead, we used a user-level daemon program to track the changed *files* at the end of each day for all participating Windows desktop machines in the workgroup. We used a user-level program instead of a kernel-level interceptor of block-level writes for the ease of deployment because end-users are reluctant to deploy an "unknown" kernel module in their production desktop machines.

The user-level daemon program tracked daily file changes by comparing the modification time of the file with its old modification time during a file system traversal. Initially, each file has no modification time and all blocks of all files are traversed to compute their 64-byte long MD5 hash values (i.e., the fingerprints). If the modification time does not agree with each other, the file is changed and a fingerprint is computed for each 4K data block in the file. The fingerprints, the new modification

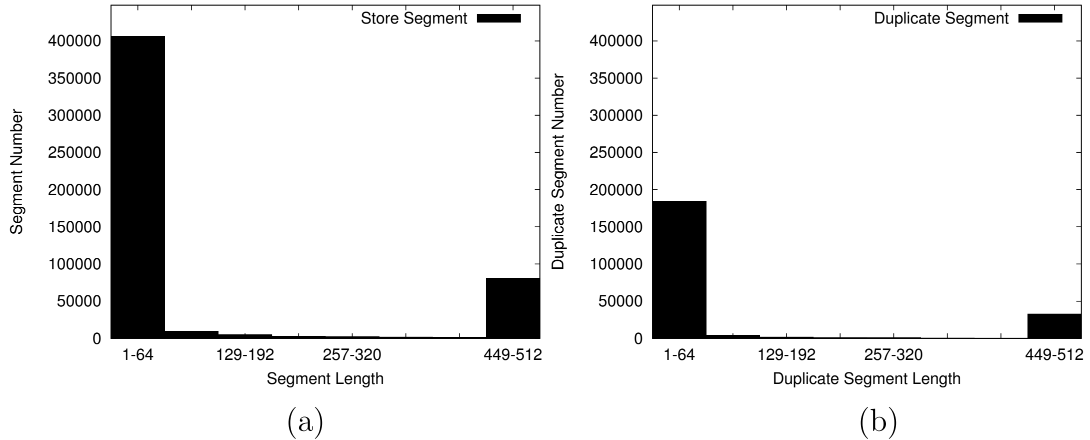


Figure 6.15: *(a) The length distribution of store segments for the 3-week trace. K is 1, and* $U_{segment}$ *= 512. (b) The length distribution of duplicate segments for the 3-week trace. K is 1, and* $U_{segment}$ *= 512.*

time of the file, and the size of the file are stored in a database file keyed by the file pathname. Next time when the user-level daemon program traverse the whole file system, the database file is queried. If the file is a new file or the file modification time mismatches, the modification time and hash values for all data blocks are added to the database file. Otherwise, the file is skipped in the current file system traversal.

Changed files do not necessarily translate to block writes for *all* blocks in the file. The behaviour of the translation from changed files to written blocks depend on the write method used to change the file. Unfortunately, we do not have the knowledge of file change methods for Windows operating systems. Instead, we rely on the file type to refer their modification behaviour.

Table 6.2 shows the file types, their conjectured write methods, and their ratio of contribution in the trace. 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 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's log files. Mails include files that are updated by the Microsoft outlook, including files with the suffix pst and ost. Database files cover all database files used by either application or the system.

For those files that append to existing files, we account the newly-appended blocks into written blocks but discard those unchanged leading portions by comparing the old file size with the current file size. If the current file size is smaller than the old file size, the file wraps around, and all blocks in the new file size range are regarded as written blocks. Otherwise,blocks that fall between the old file size and the new file size are regarded as written blocks. To make ***Workgroup*** more

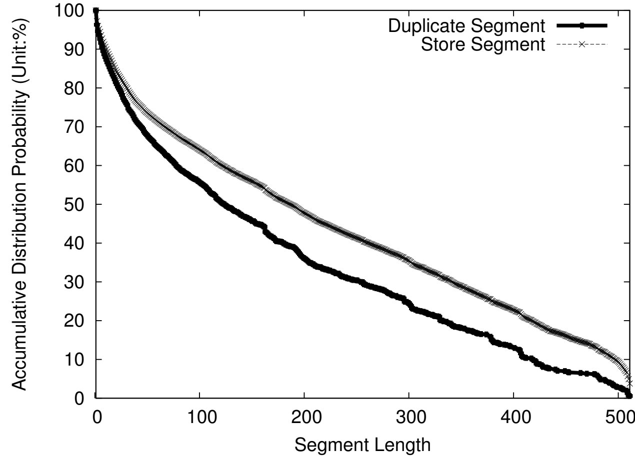


Figure 6.16: *The cumulative distribution probability of blocks in segments when the segment length is varied from 1 to 511. The upper curve is for store segments, and the lower curve is for duplicate blocks.* $U_{segment} = 512$, $K = 1$.

representative, we removed VM-related files because these files are not typical in a production workgroup.

After pre-processing of the trace, the trace is 43.7 GB in size, the file size of all initial files is 10.8GB. As we focus on the incremental changes, we only analyse those incremental changes, which is 32.9 GB in size. Because the analysis of the whole incremental changes takes more than 10 hours, we used a three-week portion of the trace to drive the analysis. The three week trace is 6.4 GB in size, and it takes less than 3 hours to finish.

Trace Analysis

To analyse the trace, the analysis program must implement all features of the proposed de-duplication system, including (1) Segment Identification, (2) VFSFI, (3) Segment-based Container, and (4) Segment Summary. To identify segments, the store segment is stored in the container. To speed up the segment query, a segment index keyed by the segment identifier, and a offset interval index keyed by the fingerprint offset are implemented as part of the container.

To evaluate the effectiveness of VFSFI, each entry in the global FI has an entry indicating if the entry is sampled or not. The LRU list of the global VFSFI is implemented as described in the design for VFSFI. We reduce the sampling rate of the segment at the end of the LRU list instead of removing the segment directly as in the basic LRU policy (denoted as *Basic-LRU*). Our LRU policy, denoted as *Rate-based-LRU*, is shown to be more effective than *Basic-LRU*.

A segment-based container clusters related segments in one container. As a result, any container can be split due to newly added segments. In contrast, traditional stream-based container have one open container for one backup stream. We

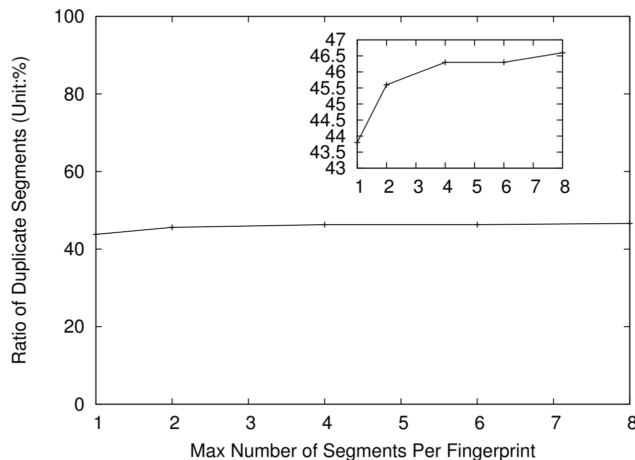


Figure 6.17: *The ratio of duplicate segments over store segments when K is varied from 1 to 8.* $U_{segment} = 512$.

implemented the container split to accommodate potential container splits. Also, a LRU-based cache is implemented to cache containers recently visited. Because containers are frequently updated, a separate update buffer is reserved to hold updates of containers. BOSC is employed in committing updates, and we can calculate the average queue length for the trace.

A summary fingerprint is computed from the fingerprints of a segment if the segment length is larger than a threshold, and the summary fingerprint is added to the summary FI. Next time, when an input segment has a size larger than a threshold, the summary fingerprint of the input segment is computed, and the summary fingerprint is used to query the summary FI. If hit, the stored physical block information is retrieved to finish the de-duplication procedure.

If the analysed trace can not fit into RAM of a single machine, the analysis of the trace can incur significant I/O overhead to read the FI and containers. Also, the trace analysis cannot sample the trace because the analysis should base on the characteristics of the whole trace, not sampled trace. To overcome this dilemma, we employed MemCached [191] to expedite the trace analysis over a cluster of machines.

Because the container is the basic unit of dealing with segments, the container is transferred over the network between the MemCached clients and MemCached servers, which can become the performance bottleneck for single-threaded analysis program. To improve the throughput, we have developed a multiple-threaded program.

The conversion from single-threaded analysis program to multiple-threaded one is not straightforward. To make threads as parallel as possible, each container has a lock. When a thread processes an input segment, it first queries the global in-RAM FI, holds the lock of the destination container, updates the destination container, releases the lock of the container, and finally updates the global FI if there are new fingerprints.

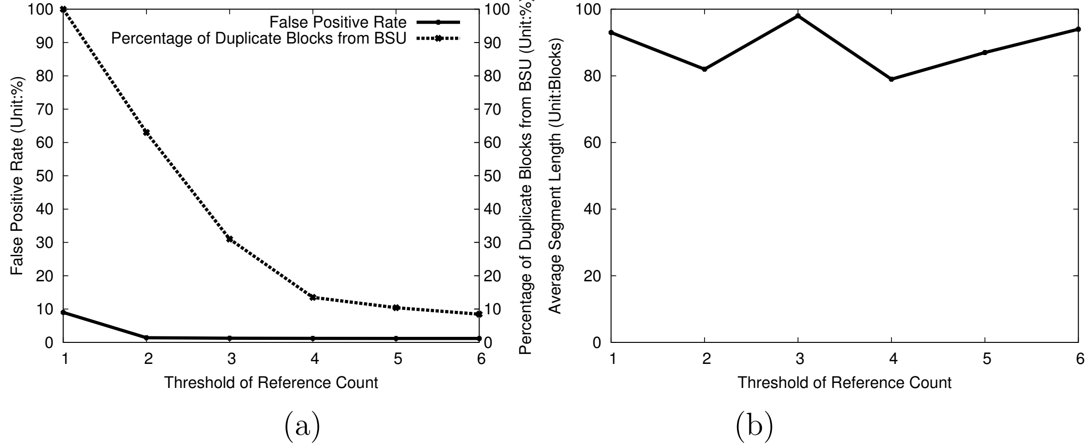


Figure 6.18: *(a) The identification of basic sharing unit (BSU) when T_{BSU} is varied from 1 to 6. The left Y axis shows the false positive (FP) rates of BSU identification. The right Y axis shows the ratio of blocks in BSU over all duplicate blocks. $U_{segment}$ = 512, K = 2. (b) The average segment length when the reference count threshold is varied from 1 to 6. All other parameters are the same as (a).*

The machines comprising the MemCacheD cluster are 6 PowerEdge SC1425 machines with up to 4 GB memory, and they are connected by a Netgear 1 Gbps Ethernet card. When multiple threads proceed with the trace analysis, the average throughput can reach 12K fingerprints per second when each container contain up to 2K 20-byte fingerprints.

Evaluation Parameters and Metrics

There are 9 parameters in the analysis program as mentioned above. The first parameter, $U_{segment}$, is the upper bound of the segment length. Segments larger than $U_{segment}$ are divided so that the divided segments are smaller than $U_{segment}$. The second parameter, K, is the upper bound of segments a fingerprint can be part of. The third parameter, T_{BSU}, is the threshold of the reference count, above which the segment is regarded as *basic sharing unit* (BSU). The fourth parameter is the amount of memory to hold SFI. If the specified amount is exceeded, the global LRU list is queried to find out victim segments to release memory from. S_{init}, the fifth parameter, is the initial sampling rate for each store segments. The sixth parameter, $S_{container}$, is the container cache size. If the container cache is full, a victim container is evicted from the container cache based on a LRU list of cached containers. The seventh parameter, Q_{BOSC}, is the amount of queue buffer to hold pending updates to containers. The eighth parameter, $T_{summary}$, is the lower bound of segments that should compute a summary fingerprint. The 9th parameter, $U_{container}$, is the upper bound of the container size.

In verifying various design decisions, we mainly used three metrics to gauge the effectiveness of each design decision. The first metric is the memory used to store

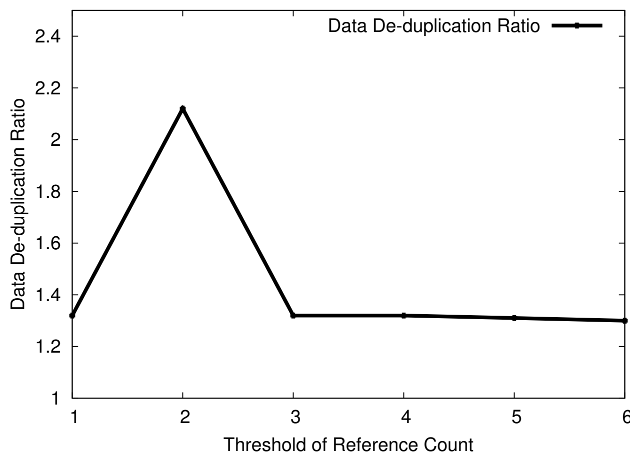


Figure 6.19: *The data de-duplication ratio when T_{BSU} is varied. For the VFSFI scheme, $U_{segment}$=4096. K =2, and the sampled fingerprint index is 1/128 of the whole fingerprint index. The initial sampling rate is 1/16.*

the sampled fingerprint index. In concrete, we derived the ratio of the sampled FI over the whole FI as the metric to gauge the memory saving. The second metric is the data de-duplication ratio (DDR), which is the ratio between the input data size over the stored data size. The third metric is the I/O cost to read containers, which is emulated by counting the number of I/Os to load containers from RAM. Other metrics are used to compare our approaches with other existing approaches, which can be found in the detailed evaluation.

6.4.2 Segment-based De-duplication

Segment is the core idea of our proposal that separates our de-duplication approach from previous approaches in the field of de-duplication for incremental backups. This section shows the distribution of the length of segments, and studies the sensitivity of the segment identification with the value of K.

Length Histogram of Store Segments for Incremental Backup

Figure 6.15 shows that the distribution of store segments and duplicate segments with regard to the length of the segment is bimodal. For input segments, segments smaller than 64 is 80% of all segments, and segments equal to the maximal segment length is 15.4% of all segments. For duplicate segments, segments smaller than 64 is 81.5% of all segments, and segments equal to the maximal segment length is 14.4% of all segments.

Further investigation of the trace shows that segments with the maximal segment length roots from multimedia files. Because multimedia files account for a large portion of the trace (21.6% of all blocks in the trace), it is not surprising that there

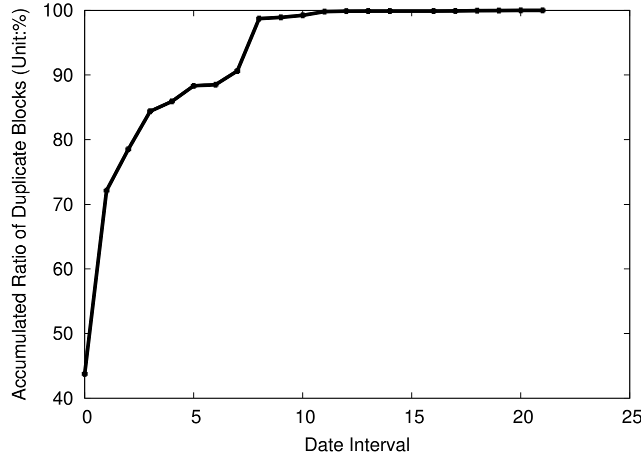


Figure 6.20: *The ratio of duplicate blocks with regard to the date interval between duplicates.* $U_{segment}$ *= 512. K =2.*

exists a significant amount of segments with maximal segment length.

Figure 6.16 shows the cumulative distribution function of blocks with regard to the segment length when segments with the maximal segment length (i.e., 512) are excluded for both store segments and duplicate blocks. For duplicate blocks, blocks in segments larger than 119 account for 50% of all duplicate blocks. The existence of duplicate segments with size larger than a trivial value (i.e., 1) provides a solid basis for our segment-based de-duplication.

Sensitivity Study of Segment History

Figure 6.17 shows that the ratio of duplicate segments increases when K increases from 1 to 8. The sole benefit of keeping up to K segments for a single fingerprint is to improve the chance of finding duplicate segments. Surprisingly, increasing K does not improve the ratio of duplicate segments over store segments significantly. The root cause is that there do not exist many shared fingerprints among segments. In the following experiment, we choose $K = 2$ to have a moderate ratio of duplicate segments.

6.4.3 Variable-Frequency Sampled Fingerprint Index

Basic Sharing Unit

Figure 6.18 (a) shows that the FP rate of BSU identification drops when T_{BSU} increases from 1 to 6. In particular, when T_{BSU} is larger than 2, the FP rate of BSU identification is under 1.4%. The trend of the FP rate of BSU identification shows that we can use T_{BSU} to identify a BSU with less than 1.4% FP rate when $T_{BSU} \geq 2$

The right Y axis of figure 6.18 (a) shows the ratio of duplicate blocks in BSU over all duplicate blocks decreases when T_{BSU} increases from 1 to 6. To balance the

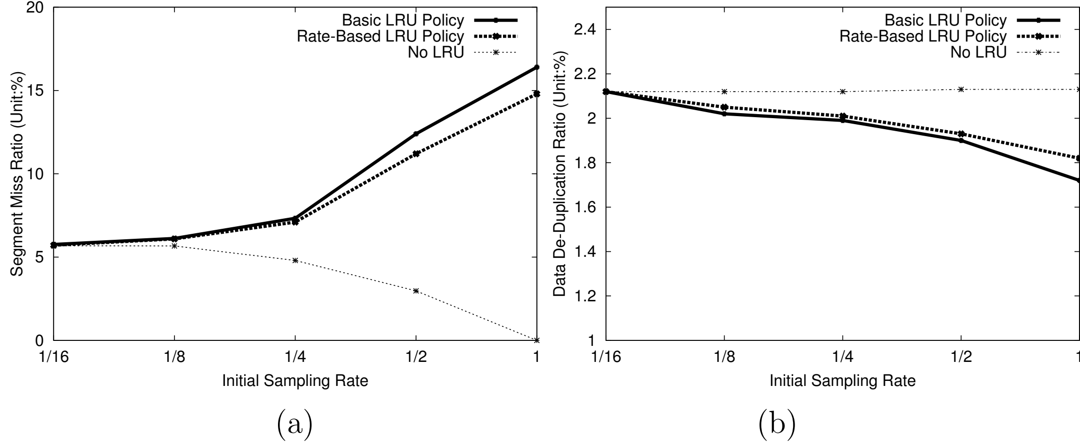


Figure 6.21: *(a) The segment miss ratio when the initial sampling rate is varied from 1/16 to 1/2 for three schemes. The basic LRU scheme removes the whole segment from SFI when the segment reaches the end of the LRU list. The rate-based LRU scheme reduces the sampling rate to half when the segment reaches the end of the LRU list. The no LRU scheme does not remove segments from SFI. The SFI memory is 1/8 of the total memory.* $U_{segment}$ *= 512, K = 2, and* T_{BSU} *= 2. (b) The data de-duplication ratio when the initial sampling rate is varied from 1/16 to 1/2. All parameters are the same as in (a).*

FP rate and the ratio of duplicate blocks, we recommend to choose $T_{BSU} = 2$ for all subsequent experiments.

One may hypothesis that larger segments are more likely to be duplicated, and therefore the average segment length can increase with the increase of T_{BSU}. However, figure 6.18 (b) shows that there is no strong correlation between the segment length and the reference count of these segments. Therefore, a smaller T_{BSU} can also result in great savings of memory in SFI because the average segment length does not change much with varied T_{BSU}.

Figure 6.19 shows that DDR reaches its maximum when $T_{BSU} = 2$. For T_{BSU} = 1, as shown in figure 6.18 (a), the FP rate is high so that many segments miss the opportunity of de-duplication due to the "1 out of N" sampling policy for those non-stable segments. When T_{BSU} is larger than 2, the percentage of BSU segments drops from 60% to less than 10% of all store segments. The drop in the number of BSU segments leads to more SFI memory consumption and results in smaller DDR for the same amount of SFI memory. When T_{BSU} continues to increase from 3, DDR drops slightly because the percentage of BSU segments drops slightly from 10% when T_{BSU} increases from 3 onwards.

LRU-based eviction policies

Figure 6.20 shows that duplicate blocks have strong temporal locality, the probability that a duplicate block and its immediate preceding instance are less than or equal to

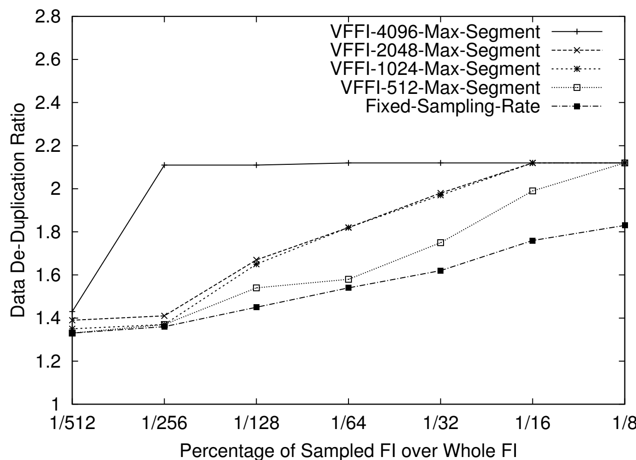


Figure 6.22: *The data de-duplication ratio when the memory for sampled fingerprint index is varied. The scheme with fixed sampling rate is compared with the VFSFI scheme. For the VFSFI scheme,* $U_{segment}$ *is varied from 512 to 4096.* T_{BSU} *= 2, K =2. The initial sampling rate is 1/16.*

9 days apart is more than 99%.. This strong temporal duplicate locality is the corner stone of our proposed LRU-based replacement policy for SFI.

Figure 6.21 compares the *Basic-LRU* scheme with our *Rate-based-LRU* scheme by varying the initial sampling rate. Initially, all segments have the same initial sampling rate. When SFI is out of memory, the segment at the end of a segment-based LRU list is chosen as the victim. For the *Basic-LRU* scheme, all fingerprints in the victim segment is evicted from SFI. For the *Rate-based-LRU* scheme, the victim segment reduces its sampling rate to half until the victim segment is eventually removed when there is one sampled fingerprint for the victim segment. The *no-LRU* scheme does not remove segments at all, and it provides an upper bound on either the de-duplication ratio or the segment miss ratio.

Figure 6.21 (a) shows that the segment miss ratio for the *no-LRU* scheme decreases when the initial sampling rate increases because more and more fingerprints are sampled into SFI and these sampled fingerprints are not evicted from SFI. In contrast, the segment miss ratio for both the *Basic-LRU* and *Rate-based-LRU* schemes increase when the initial sampling rate increases. This is because the initial sampled fingerprints are evicted from SFI, and the the larger the initial sampling rate, the larger the probability of fingerprints evicted from SFI, and the larger the segment miss ratio.

However, figure 6.21 (a) shows that the segment miss ratio of the *Basic-LRU* scheme is larger than that of the *Rate-based-LRU* scheme for all examined initial sampling rates. This is not surprising because the *Rate-based-LRU* does not remove the whole segment from SFI when the segment is chosen to be the victim segment. As a result, with the same amount of memory for SFI, the *Rate-based-LRU* scheme can cover more segments than the *Basic-LRU* scheme.

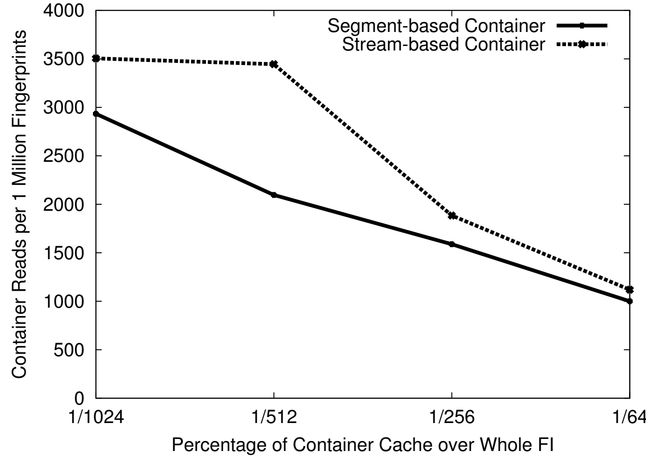


Figure 6.23: *The number of I/Os to load containers when the container cache varies its size from 1/1024 to 1/64.* $U_{segment}$ *= 512, K = 2, and* T_{BSU} *= 2. The initial sampling rate is 1/16, and the percentage of SFI over whole FI is 1/128.*

Figure 6.21 (b) shows the data de-duplication ratio (DDR) when the initial sampling rate is varied. For the *no-LRU* scheme, the DDR drops slightly from 2.13 to 2.12 when the initial sampling rate changes from 1 to 1/16. For both the *Basic-LRU* scheme and the *Rate-based-LRU* scheme, the DDR drops because more fingerprints are evicted from SFI for larger initial sampling rates. The *Rate-based-LRU* scheme out-performs the *Basic-LRU* scheme because SFI can cover more fingerprints with the same amount of memory.

Effectiveness of VFSFI

Without VFSFI, traditional SFI employs a *Fixed-Sampling-Rate* scheme for all de-duplicated streams. In the *Fixed-Sampling-Rate* scheme, after de-duplication of each input stream, the newly-added fingerprints are sampled into SFI with a fixed rate. When the maximal amount of SFI is reached, the oldest fingerprint is removed from SFI. In this section, we compare VFSFI with the *Fixed-Sampling-Rate* scheme by evaluating their DDR.

Figure 6.22 shows that VFSFI outperforms the *Fixed-Sampling-Rate* scheme in terms of DDR regardless the amount of SFI memory because VFSFI makes more efficient use of SFI memory with two methods: (1) 1 sampled fingerprint for BSU, and (2) *Rate-based-LRU* policy for the eviction of sampled fingerprints. The DDR increases for all three configurations when the amount of SFI memory increases because more segments can be covered in SFI. The DDR improves when $T_{segment}$ increases from 512 to 4096 for the following reason. Larger $T_{segment}$ can lead to larger average segment length for BSU, and more memory can be saved in composing SFI. As a result, more segments can be covered in SFI and the DDR can improve with an increased $T_{segment}$.

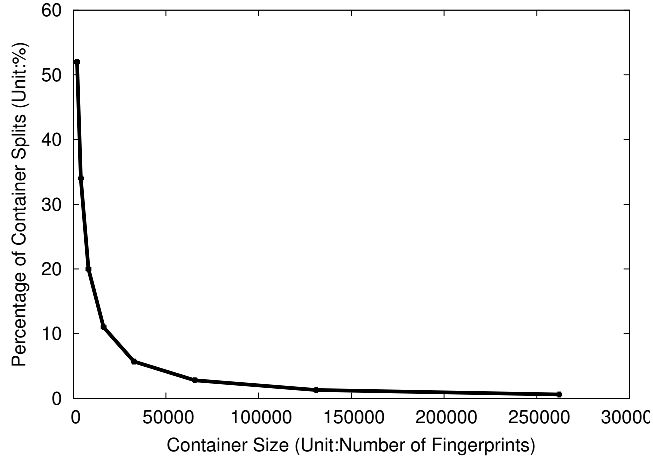


Figure 6.24: *The percentage of container splits when the container size is varied from 2K to 256K fingerprints in size.* $U_{segment} = 512$, $K = 2$, *and* $T_{BSU} = 2$.

6.4.4 Segment to Container Association

Segment-clustered Container Versus. Stream-based Container

In this section, we evaluate our segment-based container with stream-based containers. For our segment-based container, segments sharing fingerprints are put into the same container, and new segment is added to a per-stream container. In contrast, for traditional stream-based containers, any new fingerprint is added to the per-stream container. Intuitively, our segment-based container can save I/Os to read containers because an input segment can be retrieved from one container, while multiple containers need to be read into RAM if the segment dispersed over multiple containers.

Figure 6.23 shows that the number of container reads increases when the size of the container cache increases because larger container cache has larger container hit ratio.

Figure 6.23 shows that our segment-based container outperforms the traditional stream-based container scheme because the segment-based container preserves the data locality for segments in a better fashion. In particular, the saving of container I/O reads can be as high as 39% when the container cache is 1/512 of whole FI.

Overhead of Container Split

The segment-based container has the advantage of reducing the I/O cost of loading containers from disks, however, the benefit comes with a cost. That is, any container can be split when the container size exceeds a threshold. In contrast, for the traditional stream-based container, fingerprints keep being appended to the end of the per-stream open container and there is no split of existing containers.

Figure 6.24 demonstrates that the ratio of container splits over all containers drops rapidly when the container size increases because larger container size can

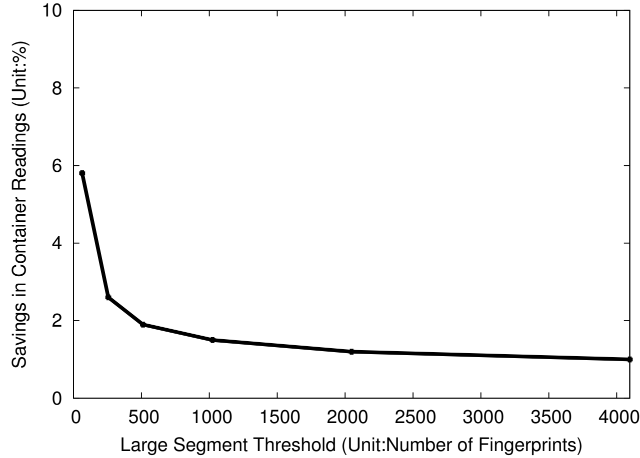


Figure 6.25: *The savings in container readings when the size threshold of segment to compute summary fingerprints is varied from 64 to 4096 fingerprints.* $U_{container} = 8192$, $U_{segment} = 4096$, $K = 2$, *and* $T_{BSU} = 2$, *the container cache size is 1/1024 of all memory. Note that all memory is computed as the memory of all fingerprints.*

accommodate more segments. In particular, when the container contains 128K fingerprints, the ratio of split containers is 1.3%. The split ratio is higher than expected.

One optimization to reduce the overhead of container split is to pre-split a container when the container's size would be exceeded if new segment is added. To achieve this optimization, each container should have an in-memory data field to indicate its current size.

6.4.5 Segment-based Summary

A segment-based summary can be employed to avoid the disk I/Os to load a container from disks if the segment summary matches with a stored segment summary. However, for de-duplication purpose, if the segment summary hits, the corresponding physical location information should be returned. If the physical location information can not fit into RAM, the loading of the physical location information itself is a disk I/O, which can defeat the very first purpose of using segment-based summary. To minimize the memory overhead due to storing physical locations, one can use the file-level information to store the collection information of stored physical locations. For example, for 1000 physical locations in a file, instead of storing 1000 individual physical locations, one can store only the file identifier and offsets of these physical blocks in the file, and rely on the file system to retrieve the physical block information if these blocks are read in the future.

In this section, we evaluate the hit ratio of the segment summary and related memory overhead to store physical location information.

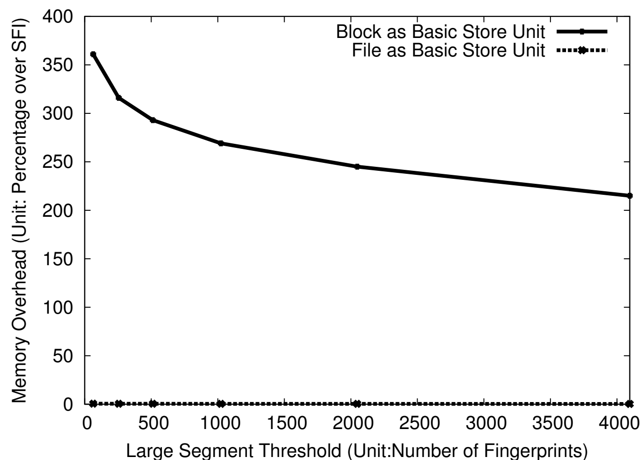


Figure 6.26: *The memory overhead of storing duplicate block information when the size threshold of segment to compute summary fingerprints is varied from 64 to 4096 fingerprints.* $U_{segment} = 4096$, $K = 2$, *and* $T_{BSU} = 2$, *the container cache size is 1/1024 of all memory. The SFI is 1/128 of the whole FI in size.*

Savings on Container Readings

Figure 6.25 shows that the ratio of saved container readings decreases as $T_{summary}$ increases because larger $T_{summary}$ has a smaller hit ratio of segment summary fingerprints. For all $T_{summary}$, the savings on the number of container readings is less than 6%, which does not translate to a significant improvement of the de-duplication throughput.

Memory Overhead

Figure 6.26 shows the memory overhead of storing physical location information when $T_{summary}$ is varied. Larger $T_{summary}$ leads to smaller memory overhead. However, if individual physical locations are stored, the memory overhead is above 200% of the SFI size for all $T_{summary}$ values. In contrast, if file-level information is available, the memory overhead drops below 1% of the SFI size.

6.4.6 BOSC to Update Containers

Because all containers have the potential to be updated, it is necessary to employ BOSC technique to improve the efficiency of committing updates. Updates to containers include added fingerprints, updated per-container fingerprint index, updated per-container segment index, and updated per-container offset interval index. We count all updates in bytes in the trace analysis program.

Figure 6.27 shows the average queue length is above 80 KB for all buffer queue size, which indicates that BOSC can greatly improve the update efficiency. Generally speaking, larger queue buffer leads to larger average queue length, however, because

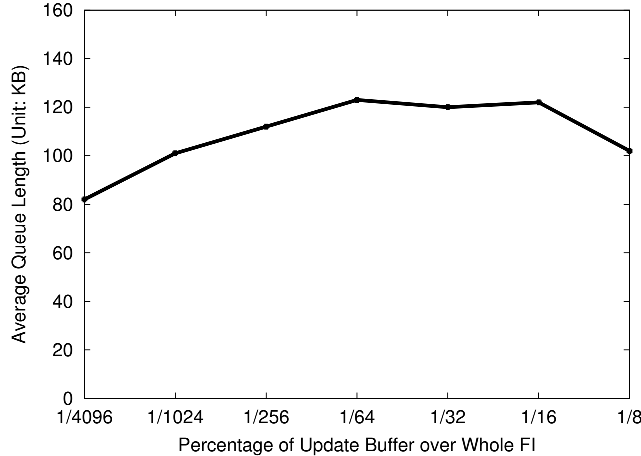


Figure 6.27: *The average queue length of each container when the queue memory is varied from 1/4096 to 1/8 of all memory.* $U_{segment} = 4096$, $U_{container} = 8192$, $K = 2$, *and* $T_{BSU} = 2$.

the update is not purely random, the average queue length does not consistently increase with the queue buffer size. In concrete, when the queue buffer size changes from 1/16 to 1/8 of all fingerprints, the queue size drops from 122 KB to 102 KB.

6.4.7 Conclusion of Design Decisions

From the analysis of collected trace, we can safely draw the following conclusions regarding the design decisions:

1. The idea of taking the segment as the basic de-duplication unit is successful because (1) the average segment length is non-trivially large (i.e., 119 for the analysed trace), (2) blocks in the basic sharing unit accounts for most of duplicate blocks.
2. K, the maximal number of segments a particular fingerprint can participate in, does not have great impact on the de-duplication procedure.
3. T_{BSU}, the threshold of the reference count to identify a basic sharing unit, does not need to be large. $T_{BSU} = 2$ can produce good de-duplication results.
4. The *Rate-based-LRU* scheme is better than the *Basic-LRU* scheme in evicting fingerprints in SFI.
5. VFSFI with BSU and the *Rate-based-LRU* scheme can have very good DDR even when the SFI is 1/256 of whole FI.
6. Segment-based containers can save I/Os in reading containers compared to stream-based containers.

7. Segment summary can save 5% of container readings.
8. For segment summary, if the physical location information is not stored per-file, the memory overhead is significant.
9. BOSC can be employed to improve the container update throughput.

Chapter 7

Conclusion and Future Work

In this chapter we first summarize the BOSC scheme. Then we highlight the applications of the BOSC scheme in three different storage systems. Finally we outline to-do tasks and future research directions.

7.1 Summary of BOSC

Modern database management systems suffer serious performance degradation when the input workload is update-intensive and has low access locality. Such workloads are not uncommon. For example, the back-end databases in typical Internet E-commerce services are routinely bombarded with workloads with intensive update requests (e.g. order processing) and random access (e.g. a large number of users). As of now, no commercial database management systems can effectively handle such workloads without resorting to special hardware such as battery-backed DRAM. This technical report describes a simple but effective solution to this problem, which consists of two key ideas: (1) an *update-aware* disk I/O interface that allows a storage application to explicitly issue disk update requests in addition to conventional disk read and write requests, and to specify an update function that the disk I/O system can invoke on behalf of the application when the target disk blocks are brought into memory, and (2) a highly efficient batched processing strategy for completing pending update requests that not only effectively amortizes the cost of each disk I/O over multiple update requests but also reduces the cost of each disk I/O to the minimum through sequential commit. We have successfully built a disk I/O system called BOSC that embodies these two ideas, and empirically demonstrated its efficiency by showing that the update request throughput of BOSC-based database index implementations is more than 50 times faster than the same database index implementations built on top of the conventional disk access interface. In addition, BOSC is able to deliver this performance improvement while providing the same durability guarantee as servicing update requests synchronously.

7.2 Summary of Applications of the BOSC Scheme

1. Continuous Data Protection
   Modern enterprise storage systems are increasingly geared towards the notion of *comprehensive data protection*, which aims to protect data from hardware/software failures, human errors, malicious attacks and environmental disasters. To achieve comprehensive data protection, existing storage systems or products tend to glue together a variety of data protection mechanisms that were developed separately in an ad hoc way. The result is that comprehensive data protection comes with excessive performance overhead. The goal of the *Mariner* project is to develop efficient implementation techniques that could reduce the performance penalty associated with comprehensive data protection to a negligible level. Along the way, we recognize that replication and logging are the two fundamental building blocks for comprehensive data protection, and organize *Mariner*'s system architecture around these two primitives to minimize the associated performance overhead.

   For a five-disk configuration, *Mariner* is able to deliver 1.8msec write latency and achieve 70% log disk space utilization under an input workload of 12500 writes/sec and 4KB per write request. With this performance result, we believe we have proved our thesis that it is possible to support comprehensive data protection without incurring significant performance overhead.

2. UVFS for Continuous Data Protection
   Commercial block-level CDP products [15, 16, 18] have emerged as a critical building block in the set of data backup tools used in modern enterprises, and have the potential to replace most of existing periodic data backup systems because of its flexible RTO and RPO and its ability to simplify storage administration. However, existing block-level CDP systems have two weaknesses. First, the point-in-time snapshots they create are not necessarily file system-consistent, or more generally do not guarantee any metadata consistency for the file servers whose data they protect. Second, they don't provide a high-level versioning view of the block versions they maintain that is more user-friendly and easier to use. Specifically, they do not provide the kind of file versioning view that a versioning file system can support. Both problems are related to the fact that block-level CDP systems are designed to be transparent to the application servers interacting with the storage that CDP systems monitor and protect. As a result, existing block-level CDP systems are limited to data backup and cannot be used as an extension of the on-line filing system.

   This technical report describes the design, implementation and evaluation of *UVFS*, which reconstructs a file versioning view similar to that provided by a versioning file system based on block versions maintained by block-level CDP systems, and for the first time demonstrates that it is possible to use a block-level CDP system as an on-line extension of the main filing system. Because

UVFS relies only on the last modify time field of files/directories, it is portable across all main-stream operating systems, including Linux, Solaris, Windows XP and Windows Vista. To discover file versions or incarnations, point-in-time snapshots need to be "fixed" so that they are file system-consistent. *UVFS* incorporates an incremental file system checker called *iFSCK* for this purpose. Although the design of *iFSCK* is targeted at legacy file systems such as ext2, it can also effectively leverage any metadata journal if the underlying file system is a journaling file system, such as ext3 or NTFS. Overall the performance costs of *UVFS* and *iFSCK* are pretty modest. It takes less than 3 seconds to discover a new version for a file that is 16 levels below the root, and on average 50 ms to find a old file version for a file that is 3 levels below the root.

3. iFSCK for Continuous Data Protection
   Existing block-level CDP systems have one limitation: the point-in-time snapshots they create are not necessarily file system-consistent, or more generally do not guarantee any metadata consistency for the file servers whose data they protect. As a result, existing block-level CDP systems are limited to data backup and cannot be used as an extension of the on-line filing system.

   This technical report describes the design, implementation and evaluation of *iFSCK*, an incremental file system checker to efficiently turn point-in-time snapshots to be file system consistent. Although the design of *iFSCK* is targeted at legacy file systems such as ext2, it can also effectively leverage any metadata journal if the underlying file system is a journaling file system, such as ext3 or NTFS. Overall the performance costs of *iFSCK* is pretty modest. It takes less than 1 second to turn a 10GB point-in-time block-level snapshot into file-system consistent.

4. Write Optimization for SSD
   LFSM is designed to solve the random write performance problem of Solid State Disks (SSD). It achieves this goal by successfully combining efficient logging and sequential writes. For a given logical write operation, LFSM first determines whether the logical block is hot or cold according to its past write pattern, allocates the tail physical block of the hot or cold log accordingly, compresses the logical block's new content to squeeze out space to hold a BMT update log entry, combines the BMT update log entry and the compressed logical block content and writes the result to the chosen log. LFSM's BMT is in a fixed location and therefore can be easily located. However, the random updates to the BMT are converted to sequential writes through a BOSC scheme, which aggregates pending updates to the BMT and asynchronously commits them to the BMT on a page by page basis. The erasure units in the hot and cold logs are used in a cyclic FIFO fashion. When reclaiming erasure units, LFSM strikes a delicate balance between minimizing the overhead of copying live blocks in each reclaimed erasure unit and ensuring enough free blocks are available to meet the demands. Combining these techniques into a software layer between the file

system and the flash disk's native driver, LFSM is able to reduce the average write latency for a 4-KB logical block from 7.8 msec to 1.6 msec, approximately a factor of 5 reduction. About 50% of the average write latency is due to the background garbage collection overhead.

Although the design of LFSM does not make any assumption about the underlying flash disk except that it has good sequential write performance, we have not tested the current LFSM prototype on a variety of commodity flash disks to claim that LFSM's performance benefits are indeed universal and independent of the internal implementation details of the flash disks. We plan to perform a comprehensive test of this sort to identify potential conflicts between LFSM and the Flash Translation Layer (FTL) and commercial flash disks. We also plan to implement LFSM directly in the FTL as a part of the firmware or native device driver. The main challenge of this port is to reduce the memory requirement of LFSM's in-memory data structures so that they can fit within the physical memory available on flash disks.

7.3 Conclusion of De-duplication for Incremental Backups

From the analysis of collected trace, we can safely draw the following conclusions regarding the design decisions:

1. The idea of taking the segment as the basic de-duplication unit is successful because (1) the average segment length is non-trivially large (i.e., 119 for the analyzed trace), (2) blocks in the basic sharing unit accounts for most of duplicate blocks.

2. K, the maximal number of segments a particular fingerprint can participate in, does not have great impact on the de-duplication procedure.

3. T_{BSU}, the threshold of the reference count to identify a basic sharing unit, does not need to be large. $T_{BSU} = 2$ can produce good de-duplication results.

4. The *Rate-based-LRU* scheme is better than the *Basic-LRU* scheme in evicting fingerprints in SFI.

5. VFSFI with BSU and the *Rate-based-LRU* scheme can have very good DDR even when the SFI is 1/256 of whole FI.

6. Segment-based containers can save I/Os in reading containers compared to stream-based containers.

7. Segment summary can save 5% of container readings.

8. For segment summary, if the physical location information is not stored per-file, the memory overhead is significant.

9. BOSC can be employed to improve the container update throughput.

7.4 Future Research Directions

More research areas can be explored to apply the BOSC scheme. One area is to apply the BOSC scheme to the layer between the CPU L2 cache and the main memory. One research challenge is that the current OS does not have direct control of the data placement in the CPU cache. More work needs to be done to port the BOSC scheme between the CPU L2 cache and the memory layer.

BOSC can be extended to operate on a cluster of machines instead of a local machine. Similar to PNUTS and Bigtable, BOSC can provide read/write services for key/value pairs on top of a cluster of machines. There are 3 research challenges in adapting BOSC to a cluster of machines. First, the software layer to position BOSC is not straightforward in a cluster environment. For a local disk, the block level is a natural place to manipulate the commit order of pending updates for BOSC. However, for a cluster connected by network, it is not clear where to manipulate the commit order of pending updates. Second, for a cluster of machines, each machine can independently commit their records and the way to distribute records to these clustered machines needs much more careful consideration. Third, durability of pending updates can be guaranteed either in a centralized fashion or a distributed fashion. For the centralized approach, all updates of all clustered machines are pushed to a single logging machine, which suffers from the single point of failures and performance bottleneck. For the distributed approach, each cluster machine logged its own updates. More research effort is required to study the trade-off among the performance, reliability and the ease of deployment in ensuring durability by the BOSC scheme atop a cluster of machines.

Bibliography

[1] 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.*, vol. 33, no. 1, pp. 145–156, 2005.

[2] 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*, Washington, DC, USA, 2004, p. 176, IEEE Computer Society.

[3] Asit Dan and Don Towsley, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 143–152, 1990.

[4] Norman P. Jouppi, "Cache Write Policies and Performance," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, New York, NY, USA, 1993, pp. 191–201, ACM Press.

[5] 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*, Washington, DC, USA, 2005, pp. 143–152, IEEE Computer Society.

[6] 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*, Berkeley, CA, USA, 2002, pp. 275–288, USENIX Association.

[7] 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*, Berkeley, CA, USA, 2002, p. 24, USENIX Association.

[8] 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)*, Washington, DC, USA, 2001, p. 87, IEEE Computer Society.

[9] Gordon F. Hughes and Joseph F. Murray, "Reliability and Security of RAID Storage Systems and D2D Archives using SATA Disk Drives," *Transactions on Storage*, vol. 1, no. 1, pp. 95–107, 2005.

[10] 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*, Washington, DC, USA, 2007, pp. 263–268, IEEE Computer Society.

[11] 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*, Washington, DC, USA, 2003, p. 1, IEEE Computer Society.

[12] Tzi-cker Chiueh and Lan Huang, "Track-Based Disk Logging," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 429–438, IEEE Computer Society.

[13] Cisco Corp., "A Closer Look at SANTap An Open Protocol for Appliance Based Storage Applications," http://www.snseurope.com/cisco_supplement/5.pdf.

[14] M. Lu, S. Lin, and T. Chiueh, "Efficient Logging and Replication Techniques for Comprehensive Data Protection," *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pp. 171–184, 2007.

[15] "InfiniView," http://www.mendocinosoft.com/technology.htm, 2007, Mendocino Software Inc.

[16] "RecoveryPoint," http://software.emc.com/products /software_az/recoverpoint.htm, 2007, EMC Inc.

[17] "Veritas NetBackup," http://www.symantec.com/business/products /overview.jsp?pcid=2244&pvid=2_1, 2007, Symantec Inc.

[18] "InMage," http://www.inmage.net/features-and-benefits.html, 2007, InMage Inc.

[19] Rick Bauer, "SSD and the SNIA: Collaborating on The Next Big Thing in Storage," http://insidesnia.blogspot.com/2008/07/ssd-and-snia-collaborating-on-next-big.html, 2008.

[20] ASUS Corporation, "Asustek Eee PC 900," http://displayblog.wordpress.com/2008/04/16/asustek-eee-pc-900/, 2008.

[21] Maohua Lu and Tzicker Chiueh, "An Update-Aware Disk I/O System for High-Performance Database Indexes," Technical Report, Stony Brook University, 2008.

[22] BTony Asaro and Heidi Biggar, "Data de-duplication and disk-to-disk backup systems: Technical and business considerations," *The Enterprise Strategy Group*, 2007.

[23] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, no. 11, pp. 190–201, 2000.

[24] Navendu Jain, Mike Dahlin, and Renu Tewari, "Taper: tiered approach for eliminating redundancy in replica synchronization," in *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2005, pp. 21–21, USENIX Association.

[25] Athicha Muthitacharoen, Benjie Chen, and David Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.

[26] Ehsan Pakbaznia and Massoud Pedram, "Minimizing data center cooling and server power costs," in *ISLPED '09: Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, New York, NY, USA, 2009, pp. 145–150, ACM.

[27] 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*, Berkeley, CA, USA, 2008, pp. 1–14, USENIX Association.

[28] 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: Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, 2009, pp. 111–123, USENIX Association.

[29] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu, "Semantically-smart disk systems: past, present, and future," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 4, pp. 29–35, 2006.

[30] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dussseau, and Remzi H. Arpaci-dusseau, "Semantically-smart disk systems," 2002, pp. 73–88.

[31] Sergey Brin, James Davis, and Hector Garcia-molina, "Copy detection mechanisms for digital documents," in *In Proceedings of the ACM SIGMOD Annual Conference*, 1995, pp. 398–409.

[32] Udi Manber, "Finding similar files in a large file system," in *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, Berkeley, CA, USA, 1994, pp. 2–2, USENIX Association.

[33] Mike Ko, Binny Gill, Biplob Debnath, and Wendy Belluomini, "STOW: A Spatially and Temporally Optimized Write Caching Algorithm," in *ATC'2009: 2009 USENIX Annual Technical Conference*, pp. 51–65.

[34] Yuanyuan Zhou, Zhifeng Chen, and Kai Li, "Second-level buffer cache management," Piscataway, NJ, USA, 2004, vol. 15, pp. 505–519, IEEE Press.

[35] Sorav Bansal and Dharmendra S. Modha, "Car: Clock with adaptive replacement," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2004, pp. 187–200, USENIX Association.

[36] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The multics virtual memory: concepts and design," *Commun. ACM*, vol. 15, no. 5, pp. 308–318, 1972.

[37] Theodore Robert Haining, *Non-volatile cache management for improving write response time with rotating magnetic media*, Ph.D. thesis, 2000, Chair-Long, Darrell D.

[38] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer, "Non-volatile memory for fast, reliable file systems," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 1992, pp. 10–22, ACM.

[39] P. Biswas, K.K. Ramakrishnan, D. Towsley, and C.M. Krishna, "Performance analysis of distributed file systems with non-volatilecaches," in *Proceedings the 2nd International Symposium on High Performance Distributed Computing in 1993*. 1993, pp. 252–262, ACM.

[40] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young, "I/o reference behavior of production database workloads and the tpc benchmarks—an analysis at the logical level," *ACM Trans. Database Syst.*, vol. 26, no. 1, pp. 96–143, 2001.

[41] Anujan Varma and Quinn Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," *IEEE Trans. Comput.*, vol. 47, no. 2, pp. 228–235, 1998.

[42] P. H. Seaman, R. A. Lind, and T. L. Wilson, "On teleprocessing system design, Part IV: An analysis of auxiliary-storage activity," *IBM System Journal*, vol. 5, no. 3, pp. 158–170, 1966.

[43] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul, "Cache-oblivious string b-trees," in *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, New York, NY, USA, 2006, pp. 233–242, ACM.

[44] Keith A. Smith and Margo I. Seltzer, "File system aging—increasing the relevance of file system benchmarks," in *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 1997, pp. 203–213, ACM.

[45] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul, "Concurrent cache-oblivious b-trees," in *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2005, pp. 228–237, ACM.

[46] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu, "A locality-preserving cache-oblivious dynamic dictionary," *J. Algorithms*, vol. 53, no. 2, pp. 115–136, 2004.

[47] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, "Cache-oblivious algorithms," in *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1999, IEEE Computer Society.

[48] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson, "Cache-oblivious streaming b-trees," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 2007, pp. 81–92, ACM.

[49] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook, "On external memory graph traversal," in *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 2000, pp. 859–860, Society for Industrial and Applied Mathematics.

[50] Bernard Chazelle and Leonidas J. Guibas, "Fractional cascading: A data structuring technique with geometric applications," in *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, London, UK, 1985, pp. 90–100, Springer-Verlag.

[51] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro, "Cache-oblivious priority queue and graph algorithm applications," in *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, New York, NY, USA, 2002, pp. 268–276, ACM.

[52] 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*, London, UK, 1995, pp. 334–345, Springer-Verlag.

[53] Gerth Stolting Brodal and Rolf Fagerberg, "Lower bounds for external memory dictionaries," in *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 2003, pp. 546–554, Society for Industrial and Applied Mathematics.

[54] Ben Martin, "Simple access berkeley db using stldb4," *Linux J.*, vol. 2007, no. 154, pp. 8, 2007.

[55] Goetz Graefe, "Write-optimized B-trees," in *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. 2004, pp. 672–683, VLDB Endowment.

[56] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, 2008.

[57] Saman Zarandioon and Alexander Thomasian, “Optimization of online disk scheduling algorithms,” *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 4, pp. 42–46, 2006.

[58] Peter M. Ridge and Gary Field, *The Book of SCSI 2/E, No Starch*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[59] Lars Reuther and Martin Pohlack, “”rotational-position-aware real-time disk scheduling using a dynamic active subset (das)”,” .

[60] Margo Seltzer, Peter Chen, and John Ousterhout, “Disk scheduling revisited,” in *USENIX Winter90: Proceedings of the USENIX Winter Technical Conference*, 1990, pp. 313–324.

[61] Lan Huang and Tzi cker Chiueh, “Implementation of a rotation-latency-sensitive disk scheduler,” Tech. Rep., Stony Brook University, 2000.

[62] Florentina I. Popovici, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau, “Robust, portable i/o scheduling with the disk mimic,” in *USENIX Annual Technical Conference (San Antonio, TX, 0914 June 2003*. 2003, pp. 297–310, IEEE.

[63] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt, “Scheduling algorithms for modern disk drives,” in *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, New York, NY, USA, 1994, pp. 241–251, ACM.

[64] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson, “Trading capacity for performance in a disk array,” in *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, Berkeley, CA, USA, 2000, pp. 17–17, USENIX Association.

[65] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger, “A Framework for Building Unobtrusive Disk Maintenance Applications,” in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2004, pp. 213–226, USENIX Association.

[66] 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*, Berkeley, CA, USA, 2002, p. 24, USENIX Association.

[67] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger, “On multidimensional data and modern disks,” in *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2005, pp. 17–17, USENIX Association.

[68] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, “Life or death at block-level,” in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004, pp. 26–26, USENIX Association.

[69] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony I. T. Rowstron, “Everest: Scaling Down Peak Loads Through I/O Off-Loading,” in *OSDI*, 2008, pp. 15–28.

[70] Dave Anderson, “OSD Drives,” http://www.snia.org/events/storage-developer2007/presentations/SIren_Object_Based_Storage_OSD.pdf, 2005.

[71] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka, “File server scaling with network-attached secure disks,” in *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 1997, pp. 272–284, ACM.

[72] Dave Hitz, James Lau, and Michael Malcolm, “File system design for an nfs file server appliance,” in *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, Berkeley, CA, USA, 1994, pp. 19–19, USENIX Association.

[73] Sun Microsystems, “ZFS: The last word in file systems,” http://www.sun.com/software/solaris/zfs_lc_preso.pdf, 2006.

[74] Himani Apte and Meenali Rungta, “Adding parity to the linux ext3 file system,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 1, pp. 56–65, 2007.

[75] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis, “BORG: block-reORGanization for self-optimizing storage systems,” in *FAST '09: Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, 2009, pp. 183–196, USENIX Association.

[76] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub, "Implementation of a linux log-structured file system with a garbage collector," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 1, pp. 24–32, 2007.

[77] Jun Wang and Yiming Hu, "A novel reordering write buffer to improve write performance of log-structured file systems," *IEEE Trans. Comput.*, vol. 52, no. 12, pp. 1559–1572, 2003.

[78] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina I. Popovici, Aditya Akella, Andrea C. Arpaci-Dusseau, Remzi H.Arpaci-Dusseau, and Suman Banerjee, "Avoiding file system micromanagement with range writes.," in *OSDI*, Richard Draves and Robbert van Renesse, Eds. 2008, pp. 161–176, USENIX Association.

[79] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[80] Tuukka Haapasalo, Ibrahim Jaluta, Bernhard Seeger, Seppo Sippu, and Eljas Soisalon-Soininen, "Transactions on the multiversion b+-tree," in *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, New York, NY, USA, 2009, pp. 1064–1075, ACM.

[81] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, pp. 19, 2007.

[82] Sang-Won Lee and Bongki Moon, "Design of flash-based dbms: an in-page logging approach," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2007, pp. 55–66, ACM.

[83] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu, "Lazy-update b+-tree for flash devices," in *MDM '09: Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, Washington, DC, USA, 2009, pp. 323–328, IEEE Computer Society.

[84] Goetz Graefe, "B-tree Indexes for High Update Rates," *SIGMOD Rec.*, vol. 35, no. 1, pp. 39–44, 2006.

[85] Goetz Graefe, "Sorting and Indexing with Partitioned B-Trees," in *Conference on Innovative Data Systems Research*, 2003.

[86] Laurynas Biveinis, Simonas Šaltenis, and Christian S. Jensen, "Main-memory operation buffering for efficient R-tree update," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. 2007, pp. 591–602, VLDB Endowment.

[87] "High Availability MySQL: InnoDB insert performance," 2003.

[88] "Oracle and TimesTen - enabling the Real-Time World," 2005.

[89] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar, "Datablitz storage manager: main-memory database performance for critical applications," in *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1999, pp. 519–520, ACM.

[90] "solidDB product family: In-memory, relational database software for extreme speed," 2005.

[91] Tokutek Inc., "TokuDB For MySQL," http://tokutek.com/technology.php, 2009.

[92] Vertica Inc., "Vertica Analytic Database," http://www.vertica.com/news/press-releases, 2005.

[93] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik, "C-store: a column-oriented dbms," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. 2005, pp. 553–564, VLDB Endowment.

[94] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. 2007, pp. 1150–1160, VLDB Endowment.

[95] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker, "Oltp through the looking glass, and what we found there," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2008, pp. 981–992, ACM.

[96] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker, "Oltp through the looking glass, and what we found there," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2008, pp. 981–992, ACM.

[97] Tiankai Tu and David R. O'Hallaron, "A computational database system for generatinn unstructured hexahedral meshes with billions of elements," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004, p. 25, IEEE Computer Society.

[98] 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*, Hongkong, China, 2002, pp. 382–393, VLDB Endowment.

[99] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[100] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[101] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[102] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner, "Monetdb/xquery: a fast xquery processor powered by a relational engine," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2006, pp. 479–490, ACM.

[103] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller, "Spyglass: fast, scalable metadata search for large-scale storage systems," in *FAST '09: Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, 2009, pp. 153–166, USENIX Association.

[104] "libkdtree++," 2005.

[105] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch, "A five-year study of file-system metadata," *Trans. Storage*, vol. 3, no. 3, pp. 9, 2007.

[106] P. A. Boncz, *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*, Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[107] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten, "Optimizing database architecture for the new bottleneck: memory access," *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, 2000.

[108] Kwan-Liu Ma and Thomas W. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *PRS '97: Proceedings of the IEEE symposium on Parallel rendering*, New York, NY, USA, 1997, pp. 95–ff., ACM.

[109] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron, "From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006, p. 91, ACM.

[110] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton, "Caching multidimensional queries using chunks," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1998, pp. 259–270, ACM.

[111] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava, "Snakes and sandwiches: optimal clustering strategies for a data warehouse," in *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1999, pp. 37–48, ACM.

[112] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy, "Storage estimation for multidimensional aggregates in the presence of hierarchies," in *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1996, pp. 522–531, Morgan Kaufmann Publishers Inc.

[113] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1997, pp. 159–170, ACM.

[114] 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*, Berkeley, CA, USA, 2006, pp. 1–14, USENIX Association.

[115] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and H. Steven, "Parallax: Managing Storage for a Million Machines," in *HotOS X, Tenth Workshop on Hot Topics in Operating Systems*, pp. 1–11.

[116] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger, "Awarded Best Student Paper! – A Framework for Building Unobtrusive Disk Maintenance Applications," in *FAST'2004: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004, pp. 213–226.

[117] J. Saran et al., "Internet Small Computer Systems Interface (iSCSI)," Tech. Rep. RFC3720, Internet Engineering Task Force (IETF), April 2003.

[118] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels, "U-Net: a user-level network interface for parallel and distributed computing," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995, vol. 29, pp. 303–316.

[119] Gregory D. Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes, "An Implementation of the Hamlyn Sender-Managed Interface Architecture," in *Operating Systems Design and Implementation*, 1996, pp. 245–259.

[120] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, 1994, pp. 142–153.

[121] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda, "High Performance RDMA-based MPI Implementation over InfiniBand," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, 2004.

[122] *InfiniBand Architecture Specification, Volumes 1 and 2, Release 1.0.a.* , vol. 1 and 2, http://www.infinibandta.org/, 1.0.a edition.

[123] Ken Yocum and Jeffrey S. Chase, "Payload Caching: High-Speed Data Forwarding for Network Intermediaries," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 2001, pp. 305–317.

[124] Gang Peng, Srikant Sharma, and Tzi cker Chiueh, "A Case for Network-Centric Buffer Cache Organization," in *Hot Interconnect 2003*, Aug 2003.

[125] Cisco Corp., "Cisco MDS 9000 Family SANTap Service-enabling Intelligent Fabric Applications," http://www.cisco.com/.

[126] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh, "Viking: A Multi-SpanningTree Ethernet Architecture for Metropolitan Area and Cluster Networks," in *INFOCOM 2004*, 2004.

[127] W. Fenner et al., "Internet Group Management Protocol, Version 2," Tech. Rep. RFC2236, IETF, Nov 1997.

[128] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The Use of Name Spaces in Plan 9," *SIGOPS Operating Systems Review*, vol. 27, no. 2, pp. 72–76, 1993.

[129] M. Kazar, "Synchronization and Caching Issues in the Andrew File System," *USENIX Winter Conference Proceedings*, pp. 27–36, 1988.

[130] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 235–246, 1994.

[131] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir, "Deciding When to Forget in the Elephant File System," *Proceedings of the 17th ACM symposium on Operating systems principles*, pp. 110–123, 1999.

[132] K. Muniswamy-Reddy, C. Wright, A. Himmer, and E. Zadok, "A Versatile and User-Oriented Versioning File System," *FAST'2004: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 115–128, 2004.

[133] E. Zadok and J. Nieh, "FIST: a Language for Stackable File Systems," *SIGOPS Operating Systems Review*, vol. 34, no. 2, pp. 55–70, 2000.

[134] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante, "Wayback: A User-Level Versioning File System for Linux," *ATEC'2004: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pp. 27–37, 2004.

[135] Z. Peterson and R. Burns, "Ext3cow: a Time-Shifting File System for Regulatory Compliance," *ACM Transactions on Storage*, vol. 1, no. 2, pp. 190–212, 2005.

[136] C. Soules, G. Goodson, J. Strunk, and G. Ganger, "Metadata Efficiency in Versioning File Systems," *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 43–58, 2003.

[137] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer, "An Asymptotically Optimal Multiversion B-tree," *The VLDB Journal*, vol. 5, no. 4, pp. 264–275, 1996.

[138] Theodore Y. Tso Valerie Henson, Zach Brown and Arjan van de Ven, "Reducing FSCK Time for Ext2 File Systems," in *Ottawa Linux Symposium 2006*, 2006.

[139] C. B. Morrey III and D. Grunwald, "CIMStore: Content-Aware Integrity Maintaining Storage," in *Work in Progress: 23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, USA, May 15-18, 2006.

[140] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal, "Fast Consistency Checking for the Solaris File System," *Proceedings of the Annual Technical Conference on USENIX Annual Technical Conference*, pp. 7–21, 1998.

[141] R. D. Norman E. Harari and S. Mehrota, *Flash EEPROM System*, Number 5,602,987. United States Patent, 1997 Feburary.

[142] I. Corporation., *Understanding the Flash Translation Layer (FTL) Specification*, http://www.infinibandta.org/.

[143] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber, "A Design for High-Performance Flash Disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, 2007.

[144] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, and Bumsoo Kim, "Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems," in *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, Washington, DC, USA, 2003, p. 474, IEEE Computer Society.

[145] Bob Fitzgerald Jim Gray, *FLASH Disk Opportunity for ServerApplications*, http://mags.acm.org/queue/20080708/?pg=20.

[146] SSFDC Forum, *SmartMedia Specification*, http://www.ssfdc.or.jp.

[147] Bumsoo Kim and Guiyoung Lee, *Method of Driving Remapping in Flash Memory and Flash Memory Architecture Suitable Therefor*, Number 6,381,176. United States Patent, 2002.

[148] Eran Gal and Sivan Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.

[149] Michael Wu and Willy Zwaenepoel, "eNVy: a Non-Volatile, Main Memory Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 86–97, 1994.

[150] Gretta Bartels and Timothy Mann, *Cloudburst: A Compressing, Log-Structured Virtual Disk for Flash Memory*, HP Corporation, SRC Technical Note, 2001.

[151] Sean Quinlan and Sean Dorward, "Awarded best paper! - venti: A new approach to archival data storage," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002, p. 7, USENIX Association.

[152] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[153] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li, "Decentralized deduplication in san cluster file systems," in *ATC'09: 2009 USENIX Annual Technical Conference*, Stanford, CA, USA, 2009, pp. 101–114, USENIX Association.

[154] Network Appliance Inc., "Netapp Deduplication (ASIS)," http://www.netapp.com/us/products/platform-os/dedupe.html, 2004.

[155] 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*, Berkeley, CA, USA, 2000, pp. 2–2, USENIX Association.

[156] 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 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, Washington, DC, USA, 2002, p. 617, IEEE Computer Society.

[157] 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: Proccedings of the 7th conference on File and storage technologies*, Berkeley, CA, USA, 2009, pp. 197–210, USENIX Association.

[158] 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*, Berkeley, CA, USA, 2008, pp. 143–156, USENIX Association.

[159] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," .

[160] 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*, Washington, DC, USA, 2007, pp. 171–184, IEEE Computer Society.

[161] Darren Erik Vengroff and Jeffrey Scott Vitter, "I/O-Efficient Algorithms and Environments," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 212, 1996.

[162] 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*, London, UK, 2002, pp. 88–100, Springer-Verlag.

[163] Edscott Wilson Garca, *DBH - Disk Based Hashtables*, http://dbh.sourceforge.net/, 2006.

[164] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton, "Cache-oblivious b-trees," in *SIAM Joural of Computing*, 2005, vol. 35(2), pp. p. 341–358.

[165] 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*, London, UK, 1995, pp. 334–345, Springer-Verlag.

[166] Open Source Development Labs (OSDL), "Database Test Suite: DBT-[1,2,3,4,5]," http://osdldbt.sourceforge.net/, 2003.

[167] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks," Institute of Electrical and Electronics Engineers, 1998.

[168] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Supplement to Media Access Control (MAC) Bridges: Traffic Class Expediting and Multicast Filtering," Institute of Electrical and Electronics Engineers, 1998.

[169] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Multiple Spanning Trees," Institute of Electrical and Electronics Engineers, 2002.

[170] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Rapid Configuration of Spanning Tree," Institute of Electrical and Electronics Engineers, 2000.

[171] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges," Institute of Electrical and Electronics Engineers, 1990.

[172] Daniel Bovet and Marco Cesati, *Understanding the Linux Kernel, Third Edition*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[173] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.

[174] Richard McDougall and Jim Mauro, *Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*, Prentice Hall, Inc., 2006.

[175] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer, "Passive NFS Tracing of Email and Research Workloads," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003, pp. 203–216, USENIX Association.

[176] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer, "Empirical evaluation of multi-level buffer cache collaboration for storage systems," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2005, pp. 145–156, ACM Press.

[177] J. Katcher, "PostMark: A New File System Benchmark," *Technical report TR-3022*, 1997, Network Appliance Inc.

[178] N. Zhu and J. Chen and T. Chiueh and D. Ellard, "TBBT: Scalable and Accurate Trace Replay for File Server Evaluation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 392–393, 2005.

[179] University of New hampshire InterOperability Laboratory, "iSCSI Initiator and Target Reference Implementations for Linux," http://sourceforge.net/projects/unh-iscsi.

[180] Yamini Shastry, Steve Klotz, and Robert D. Russell, "Evaluating the Effect of iSCSI Protocol Parameters on Performance," in *Parallel and Distributed Computing and Networks*, 2005, pp. 159–166.

[181] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 203–216, 2003.

[182] "SPEC SFS (System File Server) Benchmark," http://www.spec.org/osg/sfs97/, 1997, Standard Performance Benchmark Corporation.

[183] Maohua Lu and Tzi-cker Chiueh, "File Versioning for Block-Level Continuous Data Protection," *the 29th IEEE International Conference on Distributed Computing Systems (ICDCS'2009)*, 2009.

[184] Microsoft Corporation, "Microsoft Exchange Load Generator," http://www.msexchange.org/articles/Microsoft-Exchange-Load-Generator.html, Jan, 2007.

[185] Transaction Processing Performance Council, "TPC Benchmark E," http://www.tpc.org/tpce/tpc-e.asp, 2006.

[186] Mark Wong and Rilson Nascimento, "Digesting an Open-Source Fair-Use TPC-E Implementation: DBT-5," http://www.pgcon.org/2007/schedule/attachments/35-TPC-Ek-Wong-Rilson-Nascimento.pdf, 2007.

[187] L. Dubois and R. Amatruda, "IDC Backup and Recovery/Data Deduplication report," Technical Report, IDC and EMC, Feb 2010.

[188] 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 Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2009, pp. 56–68, IEEE.

[189] Oracle Inc., "RMAN Incremental Backups," http://www.dba-oracle.com/t_block_change_tracking.htm, 2004.

[190] Andrei Z. Broder, "Some applications of rabin's fingerprinting method," in *Sequences II: Methods in Communications, Security, and Computer Science*. 1993, pp. 143–152, Springer-Verlag.

[191] MemCacheD Group, "MemCacheD: a distributed memory object caching system," http://memcached.org/, 2010.