A Context Aware Block Layer: The Case for Block Layer Deduplication

A Thesis Presented

by

Amar Mudrankit

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

Technical Report FSL-2012-04

May 2012

Copyright by
Amar Mudrankit
2012

Stony Brook University

The Graduate School

Amar Mudrankit

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Associate Professor, Computer Science

Dr. Rob Johnson, Thesis Committee Chair
Assistant Professor, Computer Science

Dr. Donald Porter
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Thesis

A Context Aware Block Layer: The Case for Block Layer Deduplication

by

Amar Mudrankit

Master of Science

in

Computer Science

Stony Brook University

2012

The context of data is important for optimal performance of data management systems like deduplication. In typical operating systems, the block layer of the I/O stack is unaware of the context of the data it is operating on. Thanks to the simplicity and modularity of the block layer interface, it is one of the best places to implement data deduplication.

We designed an interface between file systems and the block layer that allows a file system to pass the context of the data to the underlying deduplication system at the block layer. This context is in the form of a "hint" to convey information that is useful for the block-layer deduplication system, so that it can optimize its operation. For example, the hint can indicate what data is worthy of deduplication, what data should not be deduplicated at all, or that an impending set of I/O operations are likely to generate lot of duplicates.

With hints, we observed a 1.5–2× reduction in I/Os and a 10% improvement in CPU utilization for metadata-intensive workloads, compared to a context-unaware deduplication system at the block layer. Our hinting system degraded the deduplication ratio by only 3–5%. To implement hints, we had to change fewer than 0.6% of the Linux kernel, and we changed approximately 600 LoC of file system code in two file systems (Ext3 and NILFS2). Our block-layer deduplication system is about 4,000 LoC of standalone kernel code.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without constant support and the encouragement from my advisor, Prof. Erez Zadok. I am very thankful to him for giving me opportunity to work on this project and being patient with me throughout the work. His vast experience in the file systems area was immensely helpful for this project. I am grateful to him for introducing me to the intriguing research field in the computer science.

I owe my deepest gratitude to Vasily Tarasov as well for helping me out in technical difficulties, spending time on long discussions during the design phase and benchmarking. I am very thankful for having him as a mentor. His research experience and knowledge was useful for me while making any decisions regarding the project. There were lots of technical as well as tacit teachings from him. I thank him for sharing his experience.

I thank the team of Venkatesh Kothakota, Ruchira Ravoori and Nehal Bandi who helped me implementing file system hints. Venkatesh and Ruchira are fast learners. They picked up subtle details in the Ext3 file system quickly and helped me for hint implementation in Ext3. Nehal helped me to understand the complex details of NILFS2 and then implementing hints for the same.

My elder brother, father, and mother have been very supportive for my post graduation. I thank all of them.

Thanks to all my colleagues in the File-systems and Storage Lab (FSL), especially Mandar Joshi for helping me out initially with the device-mapper interface; and Pradeep Shetty for timely advice on all possible things related to the project and valuable guidance related to journaling.

I would like to thank all the co-authors of my earlier paper related to deduplication. This includes Vasily Tarasov, Prof. Erez Zadok, Prof. Geoff Kuenning, Dr. Philip Shilane, and Will Buik. This has been one of my first research projects. I would also like to thank all the anonymous reviewers of USENIX Annual Technical Conference (ATC) who have taken the time to read our paper and gave valuable feedback.

Finally, I would like to thank my committee, Dr. Robert Johnson and Dr. Donald Porter, and my advisor, again, Dr. Erez Zadok.

This work was made possible in part thanks to National Science Foundation awards CCF-0937833 and CCF-0937854, a NetApp Faculty award, and an IBM Faculty award.

Chapter 1

Introduction

Data continues to grow at alarming rates and storage administrators are hesitant to expand the storage capacities due to financial constraints [3]. Deduplication can help to control the cost in cases where data exhibits large redundancy [12, 18, 22, 36]. Deduplication systems maintain a single copy of repeated data on disk to save disk space. Some obvious candidates for deduplication are typical backups, which often contain copies of the same files captured at different times [10]; virtualized environments storing similar virtual machines [12]; and even in primary storage, users sharing similar data [19], such as common project files or recordings of popular songs.

Deduplication can be performed offline or in-line. Offline deduplication is done after data has been written on the primary storage. In-line deduplication is done as data flows through a deduplication system. In an offline deduplication system, new data is initially stored on the storage device. The deduplication system, at a later time, analyzes the data looking for duplicates. An in-line deduplication system tries to find the duplicates in the data as the data enters the device in real time. In-line deduplication systems try hard to reduce the active data flowing across the storage and network infrastructure so as to improve capacity, bandwidth, and cost savings. In-line deduplication systems optimize the size of the data at the initial phases of its life-cycle so that any further deduplication (e.g., during backup) will result in a smaller data size.

When the duplicate-detection process is carried out at the node where data is created, it is called *source deduplication*. Characteristics of source deduplication solutions are dependent upon their position in operating system storage stack. In a typical operating system, the traditional storage stack has a hierarchical design. One of the important components of the storage stack is the file system that is responsible for organizing and managing data, ensuring data stability and reliability. Hence, source in-line deduplication systems are built around this layer. With reference to the position of the file system layer in the hierarchy, such a deduplication system can be built at three different places: at the file system layer, a layer above the file system, and a layer below the file system. Each of these approaches have their own advantages and disadvantages. The virtual file system layer, the stackable file system layer [44], and the application layer are above the file system layer in the storage hierarchy and deduplication solutions can be implemented at all three places. Deduplicating the data at these layers is favorable because the context available at these layers can help the duplicate-detection process. The same is true when performing deduplication at file system layer, by modifying the file system code.

Performing deduplication at the file system layer, however, involves lots of modifications to the file system code. This is typically strongly discouraged when it comes to the legacy and production

file systems. Data deduplication at the VFS layer or at the stackable file system layer is difficult to implement, due to their complex interface with VFS and page cache. The FUSE interface can be used to perform deduplication at the application layer [16,33]. Although the FUSE-based approach is simple, it clearly incurs performance overhead considering the number of context switches involved.

The block layer of the storage stack abstracts the representation of underlying storage hardware. It presents the file system with a linear block-address space, which is used to manage the file system data as well as metadata. The block layer is unaware of the type of data it is operating on: it just serves the requests from the file systems in terms of reading and writing blocks from and to disk. An in-line deduplication system at the block layer might end up deduplicating the data that it should not deduplicate or is not worthy of deduplication. Still, then design of a deduplication system at the block layer is modular and simpler compared to stackable file systems and more efficient than the FUSE approach [16, 33].

To address the problem of lost data context for deduplication system at the block layer, we propose an interface between the file system layer and the block layer. This interface allows upper layers to exchange some simple and trivial *hints* with the block layer in order to retain the context in which the deduplication is being performed. These hints present an opportunity to the file systems to tell a deduplication system what data to deduplicate, when to deduplicate and what set of I/O operations are likely to generate more duplicates, so that deduplication system can perform optimally.

The rest of the thesis is organized as follows. In Chapter 2 we take a look at existing in-line source deduplication solutions and their pros and cons. The design of the block layer in-line deduplication system is explained in Chapter 3 along with hint interface and hint descriptions. Chapter 4 discusses implementation details of our deduplication system that uses the Linux kernel device mapper interface; we also describe changes made in the Ext3 and NILFS2 file systems to implement hints. In Chapter 5 we present the benchmark methodology and evaluation of hint interface. We then review related work in Chapter 6 and conclude in Chapter 7.

Chapter 2

Background

In this chapter we examine existing approaches to perform in-line source deduplication and evaluate the pros and cons of each approach. We also take a look at where in the storage stack hierarchy these solutions are implemented and how their position in storage stack affects deduplication solution.

2.1 Storage Stack Hierarchy

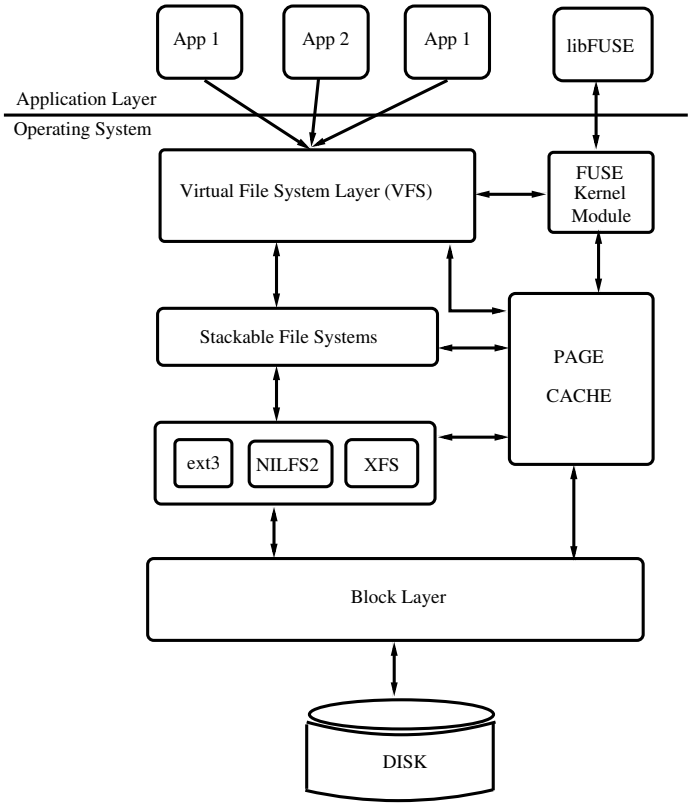


Figure 2.1: Storage Stack Hierarchy in a Typical Linux Kernel

Figure 2.1 depicts the typical storage stack hierarchy in the Linux kernel. The Virtual File System layer (VFS) in the operating system is the layer that provides file system interface to user

applications via system calls. The VFS layer ensures that user applications get uniform interface to different implementations of file systems in the kernel. For performance reasons, file system data is maintained in a kernel-level cache called the page-cache. The VFS layer and file systems are interfaced with the page cache to speed up reads and writes to files for buffered I/O operations.

Stackable file systems [44] operate at a layer between the VFS and *native* file systems. Native file systems interfaces with the block layer directly. Stackable file systems help in rapid development and maintenance of production and legacy file systems. Semantics such as encryption and decryption [11] or namespace unification [41] can be implemented at this layer without any modification to native file systems. Although stackable file systems do not interface with the block layer, they still have to be implemented in the kernel an interface with the page cache, native file systems, and the kernel's VFS layer.

Recently, user space file systems have gained lot of popularity thanks to FUSE [39], a flexible framework that is easy to use. Alas, FUSE incurs performance overhead [26] attributed solely to the number of context switches and its complex interface the with kernel's page cache.

Native file systems like Ext3 and XFS interact with the block layer, which abstracts the underlying disk hardware in terms of logical sequence of blocks. The interface of native file systems with the block layer is primarily in terms of requests to read and write specified blocks.

2.2 Deduplication Solutions

There are various deduplication solutions that are classified based on the following: when deduplication is performed, where deduplication is performed, and on what data deduplication is performed. We will briefly go over these classifications in order.

2.2.1 Offline vs. In-line

In an offline deduplication process, new data is first stored on a storage device. At a later time, a duplicate-detection process analyzes the stored data for duplicates. As signature calculations and their lookup are performed at a later time, it does not affect write latency. But this method still requires additional storage to hold new data that has not yet been analyzed.

In an in-line deduplication process, duplicate detection is carried out as data enters the storage device in real time. The signature of the data entering the device and its lookup in the signature index are performed before writing the data. Thus, in-line deduplication requires less storage but is more sensitive to write latency.

2.2.2 Primary vs. Secondary Storage

Primary storage contains data that is near active or transactional and is typically accessible to host CPU via I/O commands. Data in primary storage is frequently accessed by a host CPU. Some of the examples of primary storage data are mail servers or users' home directories. Primary storage data is generally not in large volumes. A primary storage deduplication process eliminates duplicates in this costly primary storage.

Backup and archival data are typically categorized as secondary storage data. This is not so active or historical data that it is not directly accessible to host CPU and hence not frequently

accessed. This data is generally in large volumes and contains a large number of duplicates. A secondary storage deduplication process tries to eliminate duplicates in this secondary storage.

2.2.3 Source vs. Target

Deduplication solutions can also be classified depending upon where duplicate detection process is carried out. In the source-deduplication method, duplicate detection occurs close to where the data is created; in the target-deduplication method, duplicate detection is performed where data is stored. Most of the archival and backup deduplication systems implement target deduplication method.

Venti [24], SiLo [42], and ChunkStash [8] are examples of in-line deduplication solutions for secondary storage. NetApp's ASIS [21] is an example of an offline deduplication system for primary storage data. iDedup [37] implements a primary storage in-line deduplication solution. Most of these solutions are target deduplication solutions. In this thesis, we are considering only in-line, primary storage, source-deduplication solutions.

2.3 Deduplication solutions in the Hierarchy

In-line source deduplication solutions can be implemented at different layers in the hierarchy. Some of the advantages and disadvantages of the approach are due to their position in storage-stack hierarchy.

2.3.1 Application Layer

Deduplication logic can be implemented at the application layer using the FUSE [39] framework. In this approach, all reads and writes on the file system are redirected to a deduplication library at user space, via the FUSE interface. The duplicate detection process is carried out at the user level and then the call is forwarded again to kernel space.

As the context switches involved in the I/O path are doubled, performance of such deduplication systems is not good [26]. Lessfs [16] and SDFS [33] are two open-source solutions available that implement deduplication using FUSE. We ran one small experiment using Lessfs, in which multiple Linux kernel tarball extracts were written to Lessfs mount-point. The throughput we observed was 740 KB/s on a machine with 24 cores and 64GB RAM.

2.3.2 Stackable File System Layer

Deduplication can be implemented in the stackable file system layer like Wrapfs [44]. Being part of the Linux kernel, stackable file systems do not involve additional context switches like FUSE and hence there is no performance overhead. Also, there is enough context for the data available so that deduplication can perform efficiently.

However, interfacing stackable file systems with other components in the storage hierarchy is complex. One has to implement two interfaces for such stackable file systems: (1) interface with the VFS layer, consisting of file, address space, and inode operations; and (2) interface with a native

disk file system to maintain file, dentry and inode mappings. We could not find any deduplication solution that is implemented at the stackable file system layer.

2.3.3 File System Layer

Implementing deduplication logic as an integral part of a file system incurs little performance overhead. Enough context of the process as well as the data is available to perform deduplication efficiently. ZFS [38] implements deduplication with this approach.

The important drawback of this approach is that it involves significant modifications to existing file system code. This is typically strongly discouraged for legacy and production file systems like Ext3, Ext4, and XFS.

2.3.4 Block Layer

Deduplication can be performed at the block layer, after the file system has operated on the data. Interfacing the block layer with the file system is simple and modular. File systems interact with the block layer in terms of blocks to read or write.

The main disadvantage of this approach is that context of the process or the data is lost at the block layer. The block layer cannot differentiate between the file system meta-data and the actual data. For example, file systems maintain multiple copies of the superblock across the disk for the recoverability in case of a partial disk corruption. If a deduplication system at the block layer removes this redundancy, it directly affects file system reliability.

These kind of problems can be addressed if the file systems could pass some context of data to the block-layer deduplication system. We propose that file systems can pass this context in the form of hints to the block layer, indicating what to deduplicate, what not to deduplicate and when to deduplicate, etc. We show that, with hinting support, block layer deduplication system can be (1) simpler and modular compared to stackable solution, (2) efficient compared to FUSE-based solutions, and (3) flexible to fit in production and legacy system with minor modifications.

Chapter 3

Design

In this chapter we first take a look at the interface between the file system layer and the block layer through which hints can be passed to the block layer deduplication system. Section 3.1 discusses this interface. We describe three simple hints that we designed in Section 3.2. Section 3.3 describes the design of the deduplication system prototype with hints support.

3.1 Hinting Interface

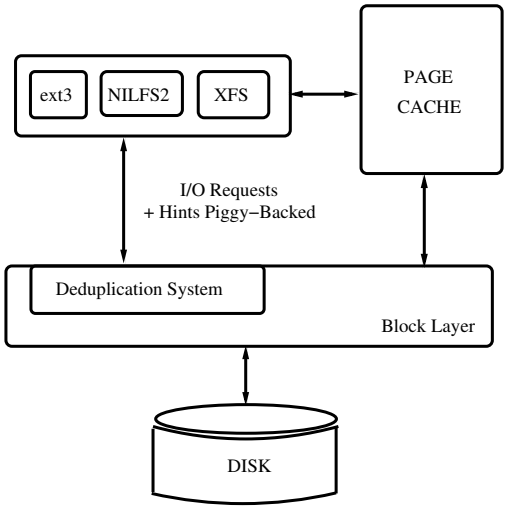


Figure 3.1: Hinting Interface

Hints are used to convey the required data and process context to the in-line deduplication system at the block layer. The file system layer and the block layer interact in terms of read and write I/O requests. Hints are piggy-backed with these I/O requests.

Hints add a negligibly small overhead on the file system and the block layer interface. I/O requests exchanged between the file system layer and the block layer typically have a flag field. This flag field indicates, for example, whether the request is read or write, or whether the request should be handled synchronously or asynchronously by the block layer and the underlying block device. Our hints are additional values in this flag field structure indicating that a given write I/O request should not be deduplicated or that a given read I/O request may shortly generate a duplicate.

As the flags field already exists and is exchanged in Linux, we simply consumed two additional bits in this flags field.

3.2 Hint Description

In this section we first describe three hints that are passed by the file systems to the block layer. We then discuss how these hints convey the required context to the block layer so that the deduplication system at the block layer can perform optimally.

3.2.1 Hint 1: Do Not Deduplicate Metadata

File system meta-data is not a good candidate for deduplication for two reasons: (1) Deduplicating meta-data affects file system reliability negatively, and (2) Meta-data typically does not exhibit redundancy.

Superblocks of a file system are duplicated across the disk for the file system recoverability. For example, the Ext3 recovery tool `e2fsck` supports an option `-b <superblock#>`. This option enables the use of secondary superblocks for recovery in case the primary superblock is corrupted. If the secondary superblocks are deduplicated, there is only a single copy of those superblock on the disk and if this one copy gets corrupted, `e2fsck` cannot recover the file system from after a crash or sector corruption.

In addition to superblocks, file systems manage variety of meta-data such as inode tables, inode bitmaps, free block bitmaps, directory blocks, checkpoint or snapshot information, and data structures to manage data blocks of the file. Superblocks are not deduplicated for the reliability reasons as explained above. The inode number, modification time, creation time, and access time of an inode introduces randomness in the inode table contents. Hence, there is a smaller probability of duplicates in the inode table. Directory blocks typically contains name of files or directories and the inode number. Due to uniqueness of the inode number, directory blocks do not exhibit redundancy. Indirect blocks (e.g., in Ext3) or B-Tree nodes (e.g., in Nilfs2) hold the block numbers, which are unique across disk device and hence do not deduplicate well. Furthermore, taking into account the size of this meta-data and its frequency of updates, this data is not worthy of deduplication. To confirm this hypothesis, we conducted two small experiments that we describe in the evaluation Section 5.4.1.

Hence, whenever file system synchronizes its meta-data to the disk, we set a flag for that I/O request indicating not to deduplicate these blocks.

3.2.2 Hint 2: Do not Deduplicate Journal Writes

For faster recovery purposes after a crash, modern file systems implement a write-ahead log to keep information about pending updates, called a *journal*. Due to performance reasons, this journal is implemented by reserving sequential blocks on the disk. A deduplication process at the block layer might interfere with such sequential writes in the journal and performance might degrade.

File systems generally log meta-data updates in the journal for faster recovery. As mentioned in Section 3.2.1, meta-data does not exhibit redundancy. Hence, meta-data writes to the journal are not worthy of deduplication.

Some file systems do log data as well in the journal for performance reasons. For example, Ext3 running in `journal` mode performs better for random write workload, because random writes are written sequentially in the journal [34]. Deduplicating the data in such sequential write streams degrades the performance of the file system.

To address these problems, we modified file systems so that they instruct a deduplication system to not to deduplicate the writes to the journal. Deduplication can be performed at a later time, when the journal is replayed or check-pointed.

3.2.3 Hint 3: Prefetch Hashes

A deduplication system detects duplicate writes by comparing the signature of the incoming block with the signatures of the other, previously written blocks. The number of comparisons required is directly proportional to the chunk size used by the deduplication system and the number of blocks written so far.

When a deduplication system supports a disk of size hundreds of gigabytes or larger, the signature store for that system is going to be large, and cannot entirely fit in RAM. Only a small amount of signatures can be maintained in RAM, and rest of the signatures are maintained on secondary storage like disk, SSD, or Flash. For an incoming block, if its signature does not match with the signatures cached in RAM, the deduplication system still needs to check if it matches with any of the signatures on the secondary storage. This requires deduplication system to issue costly I/O requests to secondary storage. These lookup I/O requests can be optimized using Bloom filters [2], but the Bloom filter can fill up over time and its false-positive rate starts to increase beyond acceptable levels.

File systems and the VFS layer are aware of the context in which data-read and data-write operations occur. For example, a process like `/bin/cp` generates duplicates of existing data. Any read I/O request to the block layer in the context of this process can serve as a hint to the deduplication system, to bring the signature of the block being read into RAM. In this way, the deduplication system ensures that the signature of the block that may be written shortly is available in RAM: costly I/O for signature lookup is not required to handle that write. Such prefetch hints help the deduplication system to keep the most promising set of hashes in its cache.

We modified the VFS layer and the `open` system call to implement a prefetch hint. When a `/bin/cp` process issues an open system call for a file, we mark the inode of the file being opened in read-only mode as a prefetch candidate. When the read I/O request is being issued to the block layer, we identify the inode of the file for which this read was being issued. If the inode is marked as a prefetch candidate, then we set the prefetch flag in the read I/O request. This flag is a hint to the block layer deduplication system to prefetch the signature of the block into RAM. In this manner, our prefetch hint propagates all the way from the system-call layer, down through the VFS, file systems, and to the deduplicating block layer.

3.3 Block layer Deduplication System

In this section we describe the design of our block layer deduplication system prototype and how hints are interpreted in our deduplication system. To the best of our knowledge, there has been no

open source implementation of a deduplication system at the block layer. Ours is one of the earliest known prototypes of an in-line deduplication system at the block layer.

3.3.1 Components

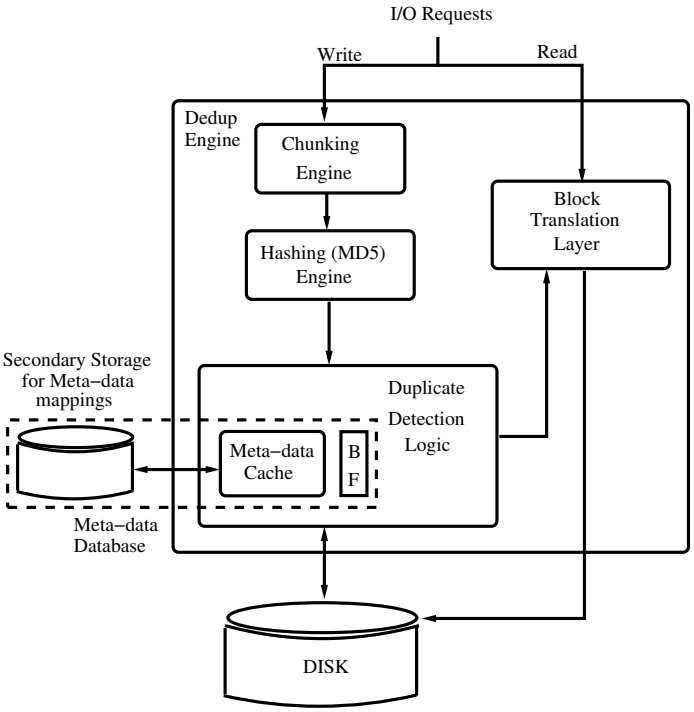


Figure 3.2: Deduplication System: Block Diagram

Our deduplication system consists of multiple components, as seen in Figure 3.2. These components work together to eliminate duplicates in the data. Some of the important components in our deduplication system are: chunking engine, hashing engine, meta-data store, and block translation layer. We discuss each component's role next.

Chunking Engine Chunking is the process in which input data is divided into chunks: the unit of duplicate detection. Fixed-size chunking simply divides the input data into fixed size blocks. This chunking method is used in Venti [24]. Variable-size chunking determines the chunk boundaries using a Rabin fingerprint approach [25] or a similar type of function. This results in blocks of variable length. Deduplication systems like LBFS [20] uses variable chunking. This approach results in better deduplication as it is resistant to insertions and modifications in the middle of the chunk, but generates chunks that are typically not block aligned.

Due to a lower computational overhead, in-line deduplication systems prefer fixed size chunking. Also, studies shows that the degradation in deduplication ratio due to fixed chunking, compared to variable-length chunking, is within permissible limits [19].

Our block layer deduplication system prototype uses fixed chunking of 512 bytes or one disk sector. This chunk size is chosen for simplicity. The minimum granularity of disk access of most of the block devices is one sector (512 bytes typically). So, the block layer cannot receive and process I/O requests of smaller sizes. In case of a chunk size larger than 512 bytes, I/O requests

spanning multiple chunks or having a size smaller than chunk size, would require a read-modify-write sequence. This adds I/O overhead for the chunking operation.

Choosing a smaller chunk size increases the amount of meta-data kept for a deduplication system, but improves the deduplication ratio. For this research prototype of a deduplication system, we were interested more in studying effects of hints on deduplication ratio than the size of deduplication system's meta-data to be managed. Hence, we chose chunk size of 512 bytes.

Hashing Engine To detect duplicates in data, the hashing engine calculates a cryptographically secure hash of the incoming write block. This hash is the basis of comparison for the duplicate-detection process. In our deduplication system prototype, we use the MD5 [30] hashing algorithm to calculate hash of the incoming write request; other hashes could be easily substituted.

Meta-data Store This is a data store that manages the hashes of the in-use blocks present on the physical disk, along with the their location on the disk, and a reference count. The reference count indicates how many duplicates of given physical blocks exist in the system. This is a form of a content-addressable storage that allows us to reference a block based on the contents of the block.

The Meta-data store is divided in three components: (1) a Bloom Filter (BF) to reduce I/O accesses for lookup; (2) an in-RAM component that holds a subset of hash entries in RAM, called a meta-data cache; and (3) an on-disk component that ensures all hash entries in the deduplication system are persistent.

Bloom filters [2] are a space-efficient probabilistic data structure used to query the membership of a hash entry in the meta-data store. When the hashing engine outputs a new hash, we first check if the hash is already part of meta-data store by querying the Bloom filter. If the Bloom filter replies to the query indicating that hash is not part of the meta-data store, then we add the hash to the store and update the Bloom filter. In this case, there is no need to look up in meta-data cache or the on-disk component of the meta-data store. But, if the Bloom filter indicates that the hash may exist (a false positive), then we need to perform a lookup in the meta-data cache—and later, if required, into on-disk component.

The classic bloom filter design has a disadvantage that entries cannot be deleted from it. Counting Bloom filters [4] are one of the alternatives to address this problem, but require more space. Quotient filters [1] are space efficient and also support deletes, but are not available inside the Linux kernel; porting it into kernel requires significant effort.

As RAM size is limited, the meta-data cache keeps the most promising set of hashes in RAM. Our meta-data cache implements two replacement algorithms: Least Recently Used (LRU) and Adaptive Replacement Cache (ARC) [17]. The size of this meta-data cache and the replacement algorithm are configurable.

When the Bloom filter indicates that a hash may exist, we compare it with hashes in the meta-data cache. When a match is found in the cache, the hash entry in the meta-data store is updated and synchronized with the on-disk component. When the entry is not found in the meta-data cache, our deduplication system performs I/O operations to check to see if the hash entry exists in the on-disk component. If the entry exists, it is updated; if not, we add a new hash entry record to the meta-data store.

In our prototype deduplication system, the on-disk component of the meta-data store is emulated by keeping all hash entries in RAM. Because we do not write the meta-data to actual non-

volatile storage, we emulate the access to the on-disk component by adding an additional latency. This flexible latency emulation allowed us to gain insights into the hints' behavior when the on-disk component is stored on many different kinds of secondary storage devices. We can emulate slow SATA disks, medium-speed SAS drives, fast SSDs, and more.

The latency value serves as one of the characteristics of that secondary storage device. For example, configuring a latency in the range of micro-seconds emulates a Flash SSD; configuring it to a few milli-seconds emulates the mechanical disk (e.g., SATA). This latency value is configurable at the granularity of nano-seconds.

Block Translation Layer The Block Translation Layer (BTL) of our deduplication system manages the physical disk blocks by virtualizing the physical block device. This layer hides the complexities of duplicate management in the deduplication system by providing a logical block interface to the file system layer. The BTL maps blocks in the Logical Block Address (LBA) space of the file system to hash entries. The meta-data store maps these hash entries to the Physical Block Address (PBA) on the physical device. The BTL is similar in functionality to a typical Flash Translation Layer (FTL) in Flash SSD drives. However, in this prototype, we have not implemented any garbage-collection. All BTL mappings from LBA to hash entry, and hash entry to PBA, are maintained in RAM. For the sake of a generic design, our prototype implements a cache to manage the most promising mappings. Thus, there is no BTL cache-miss penalty in our prototype.

For any read I/O request to LBA X, the BTL layer finds out the corresponding PBA, say Y, and issues a read request for block Y to the physical device.

In case of write I/O requests to LBA M, whose contents are a duplicate of PBA N in the system, our deduplication logic tells the BTL to add a mapping from M to N. If the write to LBA P is unique, then the BTL adds a new mapping from P to PBA, say Q.

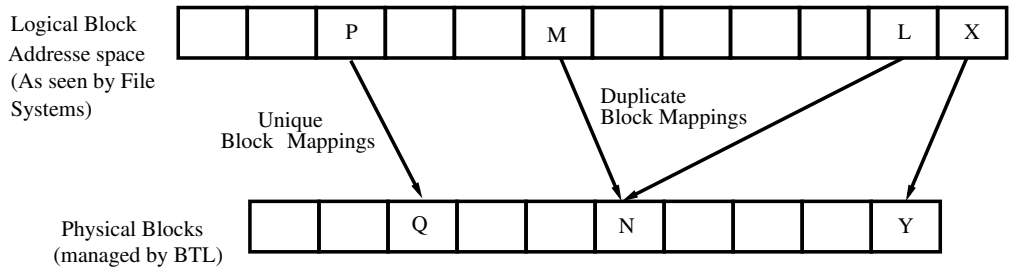


Figure 3.3: Block Translation Layer (BTL) Mappings

3.3.2 Hint Support

To support hints, we modified the typical read and write paths of the deduplication system. These path changes are depicted in Figure 3.4.

Read/Write Path To process a write I/O request, our deduplication system first divides the request into chunks of size 512 bytes. We then calculate the MD5 signature of the chunk. The duplicate-detection logic searches for the duplicate of an incoming block in the meta-data store. If a match is found, the reference count for the corresponding entry in the meta-data store is increased by one. If the match is not found, a new hash entry is created and added to the meta-data store. In

either case, the BTL is updated to add a new mapping entry. If the existing LBA is being modified, we perform the same operations, and then we update the BTL mapping for a given LBA to correct the PBA.

To process a read I/O request, the BTL finds out the PBA corresponding to the LBA requested by the file system, from its BTL mapping. Then, the BTL issues a read I/O request to that PBA.

Write Path Changes The write path of the deduplication system gets affected by hints 1 and 2 as described in Section 3.2. The meta-data and journal-write requests should not be deduplicated. We set a flag in these write requests to indicate not to deduplicate these blocks. After such blocks are identified and chunked by our chunking engine, we directly create unique mappings for these blocks in the BTL and write them out to disk. So, for these blocks, the hashing and duplicate detection steps are skipped and there is no need for any lookups for hash entries in the write path of such requests.

Read Path Changes The read path of the deduplication system gets affected by hint 3 as described in Section 3.2. For read I/O requests issued in the context of a `/bin/cp` process, we set a flag in the request asking to prefetch the hash entry of the block being read. In this scenario, the deduplication system prefetches the corresponding hash entry from the on-disk component of the meta-data store, into the meta-data cache, if it does not already exist in the cache. The write I/O request that shortly follows, having the same data, results in a cache-hit with high probability and we avoid a lookup I/O during the duplicate detection.

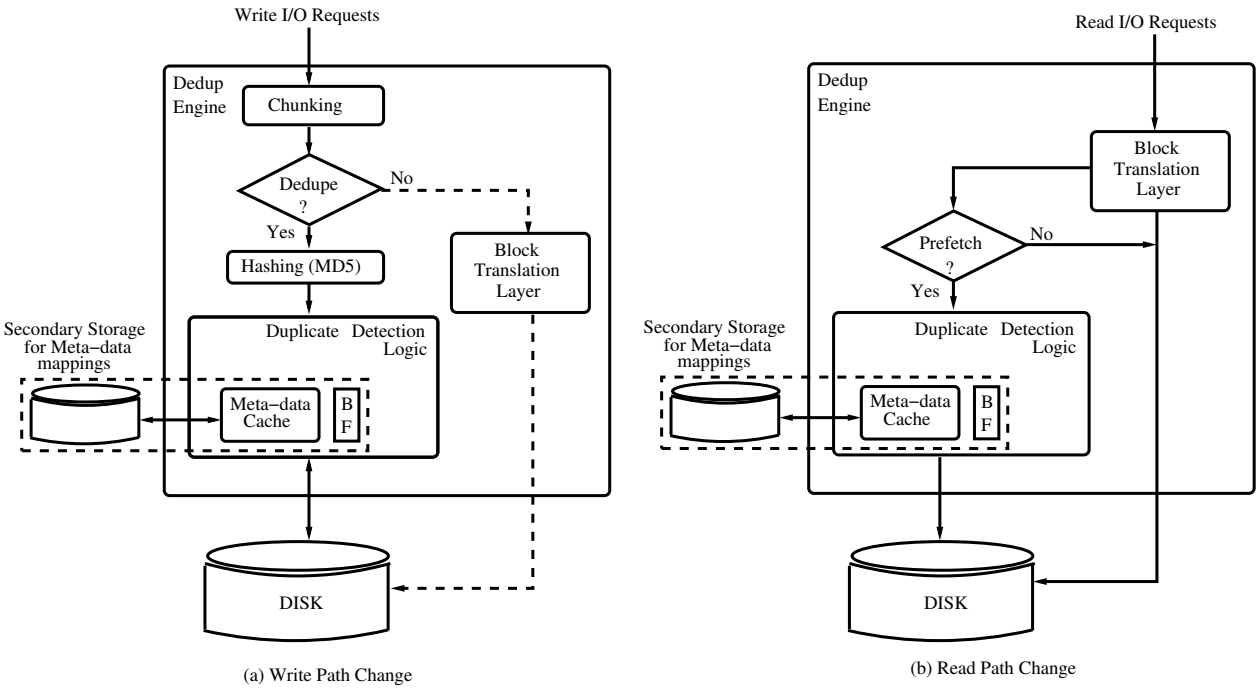


Figure 3.4: Deduplication System Modifications to support Hints

Chapter 4

Implementation

We implemented the deduplication system and the hinting mechanism for the Linux kernel. In Section 4.1 we discuss the changes in the interface between the file system and the block layer to implement hints. Then, in Section 4.2, we describe the implementation details of the deduplication system at the block layer of the Linux kernel. In Section 4.3 we describe how the hints are implemented for Ext3 and NILFS2. Implementation details of the prefetch hint are described separately in the last section, Section 4.4.

4.1 Hint Interface for the Linux Kernel Block Layer

In the context of the Linux operating system, the file system layer passes I/O requests to the block layer in the form of a `struct bio`. The `bio` structure is uniformly used for all I/O at the block layer. Some important fields in the `bio` structure from the deduplication system's perspective are:

`bi_sector`: Logical Block Address (LBA) where to write data to or read data from.

`bi_io_vec`: A vector representation pointers to an array of tuples of ⟨`page`, `offset`, `length`⟩, which describes I/O buffer in RAM to write data from or read data into.

`bi_size`: The size of the read or write I/O operation.

`bi_flags`: Flags indicating the type of operation—read or write, and some additional information.

The `bi_flags` field in the `bio` structure can carry more information than just to indicate whether the request is to read or write. For example, some flag bits are reserved to indicate device barrier requests (REQ_SOFTBARRIER), trim requests (REQ_DISCARD), and flush requests (REQ_FLUSH). The block device recognizes and respect these flags as per the standards.

To convey the context of the data being deduplicated, we define two additional flags: (1) deduplicate or do not deduplicate request (REQ_NODEDUP) and (2) prefetch hash entry (REQ_PREFETCH). The deduplication system implemented at the block layer interprets these two flags as follows:

- If the REQ_NODEDUP bit is 0, then *deduplicate* the data in `bio` structure. Else, if the bit is set to 1, then *do not deduplicate* the data in `bio`. By default, this bit is set to 0, so that we deduplicate every `bio`. This bit is relevant only in case of a write I/O request.

- If the REQ_PREFETCH bit is 0, then *do not prefetch* the hash entry of the block being read. Else, if the bit is set to 1, then, *prefetch* the hash entry of the block being read. By default, this bit is set to 0, indicating there is no need to prefetch the hash entry of the block being read. This bit is relevant only in case of a read I/O request.

Based on the context available with the file system layer, we set the appropriate bit(s) in the `bi_flags` field of the `bio` structure and then pass the request to the block layer deduplication system for further processing.

4.2 Linux Kernel Block Layer Deduplication System

In this section, we describe the implementation details of our deduplication system at the block layer. In Section 4.2.1, we explain the device-mapper interface where we have implemented the deduplication semantics. Then, we explain the implementation details of our deduplication system components in Section 4.2.2.

4.2.1 Device Mapper Layer

Device mapper [27] is a new infrastructure in the Linux 2.6 kernel that facilitates the creation of virtual layers of the block-device layer on top of real block devices. We can implement different semantics like striping, snapshotting, mirroring, and encryption at this virtual block-device layer. This is a framework to implement filter drivers, also known as *targets*. Examples of such targets are software RAID, the logical volume manager (LVM2 [28]), and the dm-crypt disk encryption target [32].

Figure 4.1 depicts the architecture of the device mapper framework. A device mapper registers itself as a block device driver with the block layer. The block layer can then forward I/O requests from file systems to the device mapper. Mapping tables are one of the important components of the device-mapper interface. This table specifies a target for each sector (512 bytes) in the virtual device created. Typically, a range of sectors of the logical device is mapped to a single target. The simplest use case of this mapping table is to join two physically separate disks together. For example, we can create a mapping so that sectors 0 to 99 of the virtual block device are mapped to `/dev/hda` and sectors 100 to 199 of the virtual device are mapped to `/dev/hdb`. In this way, we can construct a virtual block device of 200 sectors by using two separate disks with 100 sectors each. Any I/O request to the sector range from 0 to 99 is mapped to `/dev/hda`, and requests for sectors 100 to 199 are mapped to `/dev/hdb`.

The block layer forwards `bio`s received from the file system to the device mapper. Based on the `bio->bi_sector` value, the device mapper interface consults the mapping tables and finds out the target to forward this `bio` to. It then *clones* this `bio` from the original one, and forwards the cloned `bio` to the appropriate target by calling a target-specific I/O function, usually a *map* function.

A user level application `dmsetup` is used to manage these targets. The device mapper exports an `ioctl` interface for the `dmsetup` utility to use. This utility controls all aspects of the device mapper target. It can configure the sector mappings in the device-mapper mapping table as well as the target-specific configuration parameters (e.g., encryption key for the dm-crypt target). We can

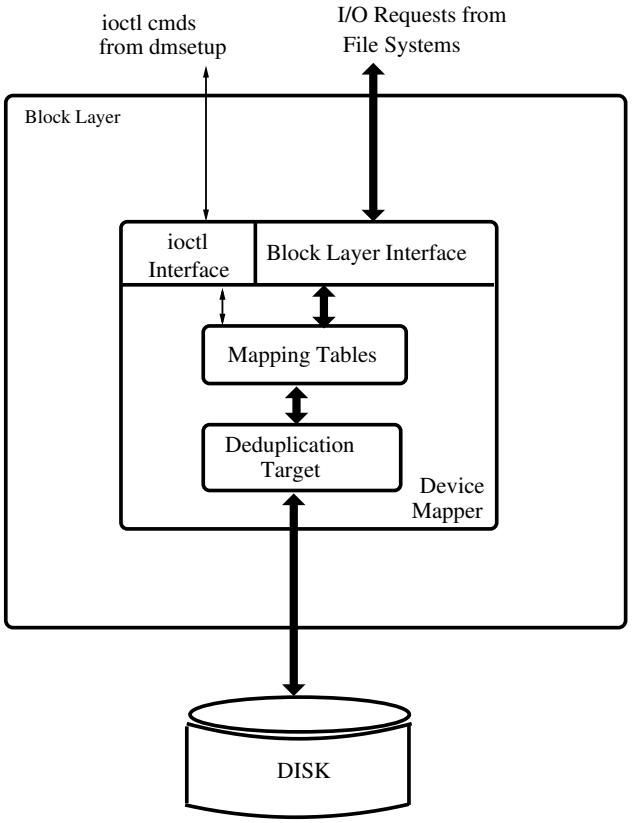


Figure 4.1: Device Mapper Interface

query the status of a device-mapper target using this utility. The `dmsetup` tool also supports other operations like device suspend, resume, delete, etc.

4.2.2 Deduplication at the Device Mapper Layer

We implemented deduplication semantics at the block layer using the device mapper (DM) interface. We first take a look at the configuration parameters of our deduplication system and then we discuss implementation details of the individual component in the system.

Configuration Our deduplication system is configured using `dmsetup` utility as shown in the following shell command. This command creates a virtual block device with deduplication capability at `/dev/mapper/dedup_disk` location.

```
# echo "0 20971520 dedup /dev/sdb 512 16777216 lru 1048576 arc
200 200" | dmsetup create dedup_disk
```

The semantics of each parameter in the `echo` string are:

`/dev/sdb:` The underlying physical block device used to store data. This is the real block device over which a virtual deduplication block device is built.

`0 20971520:` This deduplication target manages sector numbers from 0 to 20971520 of the `/dev/sdb`. This is the range of sectors obtained using command `# blockdev --getsize`

`/dev/sdb`. Unlike a striping target, our deduplication target manages all sectors of the real block device.

`dedup:` A string identifier for our deduplication target.

`512:` The chunk size for deduplication, in bytes.

`16777216 lru:` BTL cache details. The number of LBA to PBA mappings in the BTL cache should be 16777216, and the BTL cache should use an LRU replacement strategy. As mentioned in Section 3.3.1, there is no penalty on BTL cache misses in our prototype.

`1048576 arc:` Meta-data cache details. The number of hash entries in the meta-data cache will be 1048576. Our meta-data cache uses the ARC replacement strategy.

`200:` The memory pool size in MB to manage the BTL mappings. As mentioned in the Section 3.3.1, our deduplication system maintains all BTL mappings in RAM. This pool holds these BTL mappings.

`200:` The memory pool size in MB to manage the on-disk component of the meta-data store. As mentioned in the Section 3.3.1, the on-disk component is also kept in the RAM: however, access to this memory area includes an injected overhead of access latency, as if it were a regular I/O device.

The above is just one of the configurations possible for our deduplication system. One can choose an optimal configuration that suits their environment, the size of available RAM, and the workload characteristics.

Chunking With help from the device mapper interface, we break the incoming I/O request into chunks of size 512 bytes. The device mapper interface has an option through which the targets register the I/O request size they can process. If any I/O request is of size greater than what the target can process, the device mapper splits the request into multiple requests of size registered by the target. Striping targets from the software RAID device mapper use this feature.

Our deduplication system uses the same feature for chunking. During initialization, the deduplication target registers an I/O size of 512 bytes with the device mapper interface. The device mapper interface then splits each I/O request into chunks of size 512 bytes before mapping the request to the deduplication target.

Hashing Linux supports cryptoAPI [29], a cryptographic framework for various kernel components that are related to cryptography. We use the MD5 cryptographic API in Linux to compute the hash of the I/O request.

Meta-data Store As explained in Section 3.3.1, the meta-data store consists of three components: (1) a Bloom filter, (2) a meta-data cache, and (3) an on-disk component.

We implemented a classic Bloom filter [2] with a false positive rate of 1%. We directly use the MD5 hash of the block, calculated by hashing engine, as a key to lookup and insert in the Bloom filter. The MD5 signature is 128 bits long and hence, the total key space for the Bloom filter is large

(2^{128}). To limit the size of the bloom filter, we first count the number of unique MD5 signatures present in the workload under test, using a tool we developed separately [40]. Then, we configure the Bloom filter with that many keys in our block layer deduplication system and run the workload on the deduplication system. Ideally, multiple Bloom filter solution [6] can be used to handle this issue, and for Bloom filter aging; this is subject of future work.

The number of entries and the replacement strategy for the meta-data cache are configurable parameters. We implement both the LRU and ARC replacement strategies using the Linux kernel's `list_head` data structure and the corresponding APIs to operate on it.

We emulate the on-disk component of the meta-data store completely in RAM. To check whether there exists a duplicate of the given chunk, we perform a lookup in the meta-data store. We build an index to perform this lookup using a combination of a hash table and Red-Black Trees. With our data structures, the lookup latency is negligibly small compared to the on-disk component latency values emulated.

As described in the Section 3.3.1, we emulate the access to the on-disk component by adding a latency overhead. This design emulates the scenario when an on-disk component is managed on a range of secondary storage devices with different access latencies.

We emulate this latency using a sleep call at the granularity of nano-seconds in the meta-data cache-miss path. We used the high-resolution timer feature of the Linux kernel to implement nano-second sleep function. This latency value is configured via `sysfs` interface of our deduplication system.

Block Translation Layer The BTL manages the mappings from the logical block address (LBA) to the physical block address (PBA). We maintain all of these mappings entirely in RAM. We build an index for these mappings using a combination of hash table and Red-Black Trees. The lookup overhead for BTL mappings is negligibly small. Although our implementation supports the BTL cache, there is no penalty on a BTL cache miss.

4.3 File System Changes

Linux supports a large number of file systems. The hint interface can be implemented for almost all of them. In this section we describe the hint implementation for two different kinds of file systems: Ext3 and NILFS2. Ext3 is one of the most widely used, production, and legacy file systems. NILFS2 is a more recent file system, and an example of one of the more complex file systems. Our set of changes are simple and as minimal as possible so that they can be safely accommodated in any production file system.

The disk organization of a file system is an important factor for our implementation of the hint interface. The disk organization tells us which blocks occupy the meta-data of the file system and which blocks hold the user data. To avoid deduplicating journal blocks, we had to understand how a particular file system implements a journal. Next, we describe these details for each of the two file system.

4.3.1 Ext3 File System Changes

Disk organization Figure 4.2 shows the disk layout of the Ext3 file system. The Ext3 file system divides the disk into set of block groups. Each block group consists of meta-data blocks at the start of block group and data blocks following them. The first block group starts with a superblock located at the byte offset 1,024 from the start of the disk volume. The superblock in the first block group is logical block number (LBA) 1, for an Ext3 file system formatted with the block size of 1KB. If the Ext3 file system is formatted with the block size greater than 1KB, then the superblock is at LBA 0. Blocks following the superblock contain the block group descriptor table (GDT). This table holds the information about a complete disk volume in terms of how the volume is split into the block groups and where to locate the inode bitmap, the block bitmap, and the inode table for *each* block group. A small number of blocks are reserved at the end of the group of descriptor table to support the file system re-size operation.

The first version (revision 0) of the Ext3 file system stored the copy of the superblock, the block group descriptor table and the reserved group descriptor tables at the start of every block group. But, in the sparse representation of the Ext3 file system (revision 1), copies of the superblock, the GDT, and the reserved GDT are not kept in all the block groups; instead, they are kept only in block group numbers 0, 1, and those that are powers of 3, 5, or 7. All block groups still contain the block bitmap, the inode bitmap, and the inode table to manage the data in that block group. The free block bitmap and the inode bitmap consume one block each in the block group. The inode table occupies more than one block in the block group. The number of inodes in the block group is decided when the file system is formatted using the `mkfs` utility.

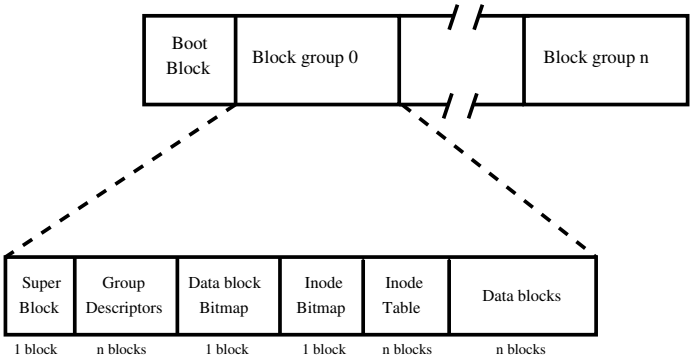


Figure 4.2: Ext3 File System: Disk Layout

Journal The Ext3 file system is an Ext2 file system with addition of the journal for faster recovery after crash, and enforced ordering of writes. The Ext3 file system journal can reside on a block device separate from the block device that is used by the file system; or the journal could be part of the same file system. For our hints implementation, we do not consider the case when the Ext3 journal resides on the separate block device. We address this case by not enabling block layer deduplication for that journaling block device.

When the journal is part of the Ext3 file system, it is treated as a special file with fixed inode number 8. All journal blocks are allocated sequentially when Ext3 is formatted using the `mkfs` utility. The Ext3 file system supports three journaling modes: (1) `journal`, (2) `ordered`, and (3) `writeback`. Ext3's meta-data is always journaled. The `ordered` mode is the default

journaling mode. The other two journaling modes can be set using the mount time option `-o data=journal` or `-o data=writeback`. In `journal` mode, the Ext3 file system writes data as well as meta-data to the journal. During the check-point operation, at a later time, this data is written back to the main file system. This mode of journaling converts random writes to the disk into sequential writes and can improve performance benefits under certain random-write workloads. In the `ordered` mode of journaling, the Ext3 file system forces data writes directly to the main file system before writing meta-data to the journal. In the `writeback` journaling mode, the Ext3 file system does not preserve write ordering for the meta-data and the data. This mode still guarantees internal file system integrity, but have weaker consistency semantics compared to the other two journaling modes. The Ext3 journal also has a `commit=nrsecs` option, which syncs meta-data as well as data every `nrsecs` seconds.

Hints for meta-data The position of meta-data blocks in the Ext3 file system is fixed. We take advantage of that fact. By performing simple calculations, we find out the position of the logical block being written in the block group. If the LBA in the write I/O request indicates that it is a meta-data write, then we set the REQ_NODEDUP flag for that write I/O request. Otherwise, it is possibly a directory block, an indirect block, a journal block, or a data block—and we do not set the flag.

We analyzed the Ext3 file system code to find out all of the places in the code where Ext3 writes the directory blocks and the indirect blocks. We set the REQ_NODEDUP flag for these blocks at the corresponding positions in the file system code.

Hints for journal writes When the journal is part of the Ext3 file system, we identify the journal writes using their LBA numbers. As the Ext3 file system places journal blocks sequentially on the disk, we note down the range of journal blocks at the mount time. If the LBA in the write I/O request is within the limits of this sequential journal block range, then we set the REQ_NODEDUP flag for such write I/O requests.

All write I/O requests which do not fall in the above two categories are considered data blocks and those write I/O requests are not modified; they are flagged for deduplication.

4.3.2 NILFS2 File System Changes

Conventional file systems try hard to make in-place changes to their data structures. The design of the log-structured file system [31] is based on the fact that I/Os in modern computers contain more writes than reads. As RAM sizes are increasing, most of the read requests are satisfied from the memory cache. The log-structured file system treats the block device as a large circular log and all data as well as meta-data writes are written sequentially in this log. Thus, write-dominated workloads (sequential or random) see improved throughput because all writes are sequential. An inherent log structure used to manage the file system helps the recovery process. NILFS2 [43] is one of the implementations of the log-structured file system on Linux.

Next we discuss the disk layout of the NILFS2 and how NILFS2 implements its journal.

Disk organization Figure 4.3 shows the disk organization for the NILFS2 file system. NILFS2 divides the disk into a set of segments, each segment consisting of set of circular logs. The log is

the disk area where writes are made sequentially. Each log in NILFS2 consists of regular file data blocks, file B-tree node blocks, inode blocks, inode B-tree node blocks, directory blocks, and some meta-data files. Meta-data files are used to maintain the file system meta-data.

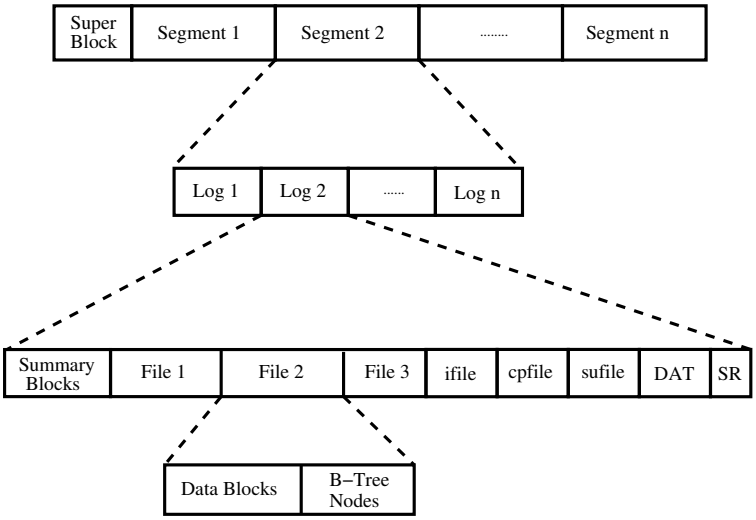


Figure 4.3: NILFS2: Disk Layout

The inode file (`ifile`) stores the information about on-disk inodes. File or directory creation or deletion causes change in the `ifile` and a new copy of the `ifile` is created and written in a new segment. An older copy of the block is marked obsolete. NILFS2 supports a continuous snapshotting feature via checkpoint mechanism. These older obsolete copies are retained for the check-pointing purpose. The `cpfile` in the log maintains the information about the current checkpoint. The allocation state of the segments is stored in the `sufile`. NILFS2 virtualizes the physical block device. The mapping of the virtual block numbers to physical block numbers is stored in the data-address translation file (DAT). The super root (SR) block tracks the current working copies of `cpfile`, `sufile`, and the DAT.

Journal NILFS2 writes segments sequentially on the disk like an append log. Because the disk space is limited, we need to clean up the deleted and obsolete copies of the blocks. For example, as shown in Figure 4.4, when the file B is created, the old root directory block and the old `ifile` block become obsolete; new copies of both the blocks are created in the new segment. These two blocks need cleanup. A garbage-collection process has the responsibility to clean up these blocks and reclaim free space. In this example, the garbage-collection process copies the valid A1 and A2 blocks into a new segment; then the whole Segment 1 is available for a long sequential write. In this way, garbage collection avoids fragmentation in the disk and ensures that empty segments are always available to accommodate large sequential writes. Garbage collection is implemented in NILFS2 as a user level daemon that runs at specific time intervals. This daemon process sends `ioctl` commands to the file system to trigger garbage-collection activity.

Hints for meta-data In the context of the hinting mechanism for the block layer deduplication system, `ifile`, `cpfile`, `sufile`, the DAT, super root, and B-tree node blocks—all constitute the meta-data and we do not deduplicate them. We identified the places in the code where NILFS2 writes these meta-data blocks and we set the REQ_NODEDUP flag in these write I/O requests.

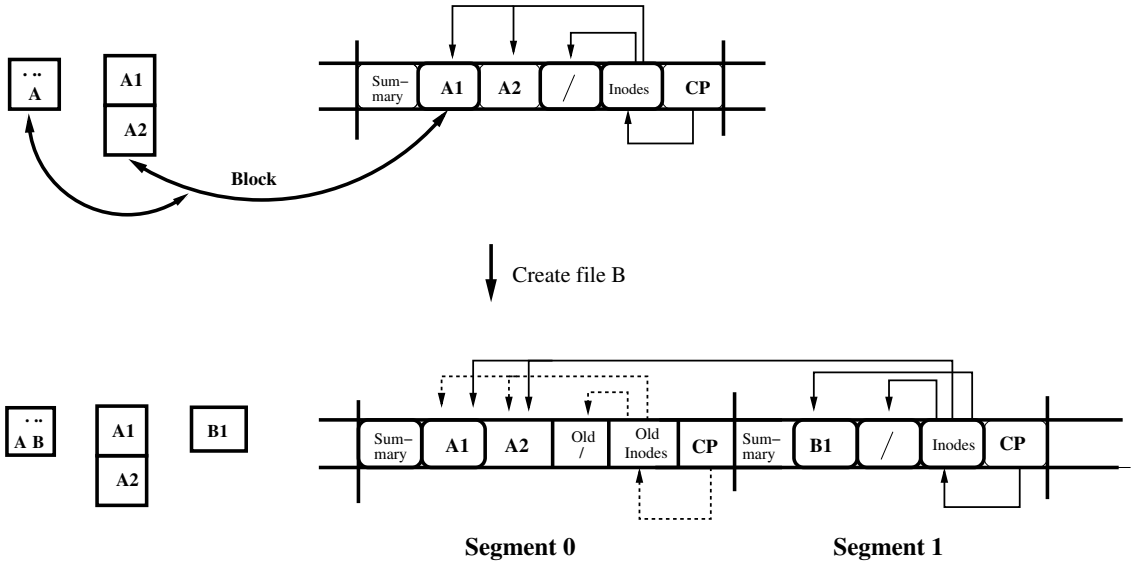


Figure 4.4: NILFS2: File Creation

Hints for journal writes NILFS2 writes everything in the log format, characterized by large sequential writes. If the deduplication logic interferes with these writes, any performance benefits from large sequential writes could be lost. Hence, we added a mount time option `dedupongc` to NILFS2 to perform deduplication only during the garbage-collection phase. When we mount NILFS2 with this option, we defer the deduplication of the data blocks. We set the REQ_NODEDUP bit in the write I/O requests for data blocks as well. When the garbage-collection process selects the segment consisting of such data blocks for cleaning, it re-writes the valid blocks from the old segment into the new segment. When the garbage collector issues write I/O requests to re-write these data blocks into the new segment, we reset the REQ_NODEDUP flag in the request so that the block layer deduplication system looks out for duplicates in these blocks. Thus, we modified the NILFS2 code in such a way that data write I/O requests issued in the context of the process or the kernel pdflush daemon have REQ_NODEDUP flag set so that these writes are not deduplicated. If the garbage-collection process issues the data write requests, then we reset the REQ_NODEDUP flag in the write I/O request so that the block-layer deduplication system deduplicates them. We do not deduplicate meta-data blocks irrespective of this mount time option and the context in which meta-data writes are issued.

Deferring the deduplication during garbage collection has an advantage that we do not end up deduplicating the data that is short lived, transient, or frequently changing. Only valid data blocks that are persistent on disk for a longer time are deduplicated. This approach is similar to a typical offline primary storage deduplication system.

One limitation of this approach is that duplicate blocks might be present on the disk until the garbage-collection process finds them out. But, we make this trade-off to keep up with the sequential write performance. This approach has a performance impact in one extreme case: when most of the segments on the file system are half full and garbage collector has to do a lot of cleaning to sustain the application or pdflush writes, it asks the deduplication system at the block layer to deduplicate them. Deduplicating the data in this context degrades the overall system throughput. This is because the garbage collector should clean up the segments as fast as it can so as to sustain the application writes, but the block layer deduplication system might introduce additional

latencies. This problem can still be addressed by having a separate garbage-collection policy: for example, the garbage collector could issue write I/O requests with REQ_NODEDUP flag set in such contexts. These enhancements are left for future work. Another way to address this problem is to over-provision the segments in the file system as done in Flash and SSDs.

4.4 Prefetch Hint

While analyzing the file system code and the VFS code, we found that the prefetch hint can be implemented in the VFS layer instead of modifying individual file systems. Hence, we implemented the prefetch hint in a file-system–independent way.

We add an extra prefetch bit for `i_flags` field in the inode structure `struct inode`. When the process `/bin/cp` issues an `open` system call, we mark the inode of the file being opened in O_RDONLY mode as a prefetch candidate. This is done by setting the prefetch bit in the `i_flags` field of the given inode structure.

In the `submit_bio` function, if a READ request is being issued to the block layer, we check the inode of the file being read: `bio->bi_io_vec->page->mapping->host`. If the `i_flags` field of that inode indicates that it is marked as a prefetch candidate, then we set the REQ_PREFETCH bit for that READ request.

One limitation of this approach is this: if the file to be copied is already cached into the page-cache, then no READ I/O requests are issued to the deduplication system at the block layer. In that case, we do not get chance to send a prefetch request in this scenario.

In the context of the block layer deduplication system, if the entry to be prefetched is already in the meta-data cache, then we do not perform any prefetch operation. But, if the entry to be prefetched is not in the meta-data cache, we emulate an asynchronous prefetch operation from the on-disk component by sleeping for a given latency period using the Linux `workqueue` data structure. As mentioned in the Section 3.3.1, this latency period is the characteristic of the secondary storage device used to manage the on-disk component of the meta-data store. The nano-seconds granularity of the high resolution timer allows us to emulate a wide range of secondary storage devices for the on-disk component.

The prefetch operation should be consistent with the cache replacement policy used for the meta-data cache. As mentioned in Section 3.3.1, we implemented two cache-replacement algorithms for the meta-data cache: LRU and ARC. We take now discuss how do we implemented the prefetch operation for each replacement algorithm.

Prefetch in LRU Cache Incorporating the prefetch logic for the LRU replacement algorithm was simple. We put the prefetched entry at the head of the LRU queue. If the queue is full, then the LRU entry at the tail of the queue is evicted.

Prefetch in ARC Cache Ensuring a consistent prefetch operation for the ARC replacement is a more complex task, especially when it involves two active cache lists (T1 and T2) and two ghost cache lists (B1 and B2). We used the same prefetch logic as used in the OS buffer cache prefetch study [5] for ARC replacement:

Component	LoC (approx.)
Dedup System	3,939
Hint Interface	4
Ext3 FS Changes	343
NILFS2 FS Changes	278
VFS Changes	54
Filebench Changes	312
Test Scripts	455
Total	**5,383**

Table 4.1: Implementation Statistics

1. If the entry to be prefetched is not in any of the lists T1, T2, B1, or B2, then it is placed at the head of the T1 list with a special flag indicating that the entry is in T1 due to a prefetch operation. The flag ensures that we do not move this prefetched entry to the T2 list on the next on-demand access and thus corrupt the T2 list.

2. If the entry to be prefetched is in the ghost cache (B1 or B2), then we move the entry to the head of the T1 list and not to the T2 list. If the entry has been prefetched correctly, then it gets moved to the T2 list on subsequent on-demand access. But, if it does not get accessed, then it does not pollute the T2 list.

3. If the prefetched entry never gets accessed even once, before evicting it from the T1 or T2 list, then that entry is not moved to the ghost cache (B1 or B2), but evicted from the cache directly.

The usefulness of the prefetch hint depends on several factors such as the frequency that the `pdflush` daemon runs, the size of RAM, percentage of dirty pages allowed in the system, any parallel workloads running on the system, etc. For example, consider a workload, say A, running parallel to a `/bin/cp` process and performing writes on the same mount-point created over a virtual deduplication device. Write requests from workload A might evict some of the prefetched hashes useful to `/bin/cp` writes from the meta-data cache. Hence, when an actual write request for the duplicate generated by the `/bin/cp` process reaches the deduplication layer, it might result in a cache-miss; the deduplication system then has to perform the lookup into the on-disk component to ensure that it is a duplicate. Thus, the prefetch operation done earlier for that hash value would not have been useful.

Implementation Statistics Table 4.1 shows the lines of code written or modified for major components of this system:

Chapter 5

Evaluation

In this chapter we describe the tests we used to evaluate the benefits of our hinting mechanism and results of those tests. In Section 5.1, we briefly describe the challenges we faced to evaluate our block layer deduplication system. We describe the modification required to the benchmarking tool in order to evaluate our deduplication solution in Section 5.2. Configuration parameters and physical machine description are mentioned in Section 5.3. In Section 5.4, we describe our micro-benchmark as well as the macro-benchmark results and their analysis.

5.1 Challenges in Benchmarking Deduplication Systems

Deduplication is the process of eliminating duplicates in the data. The data contents are one of the primary factors affecting the performance of an in-line deduplication systems, because it affects different components in the system like meta-data cache behavior, meta-data store size, bloom filter, etc. Most of today's file system benchmarking systems either use 0's or random data to write. Clearly, these benchmarks are not suitable to evaluate the deduplication systems. The research into benchmarking the deduplication systems is in its infancy. So far, there are a few attempts in this field, including our own [40].

In the context of an in-line deduplication systems, the order in which chunks arrive to the deduplication system also matters. Block traces are useful in this scenario. But, block traces do not contain information about block contents. There are few traces available [14] that include block signatures. But, they are not suitable to evaluate our hinting mechanism, because the hinting mechanism is implemented at the file-system layer. We needed file system traces with signatures of the write I/O requests. To the best of our knowledge, such traces are not available.

In our experiments to evaluate the hints support for the deduplication system, we addressed the problem of data contents by changing the benchmarking tool. To minimize the effects of the order in which the deduplication system receives chunks, we noted the average values over three runs and ensured that the standard deviation in the results is less than 5% of the mean.

5.2 Filebench changes

To evaluate our deduplication system and the hinting mechanism, we needed a flexible benchmarking system that generates a variety of workloads so that the benefits of individual hints can

be demonstrated. We used Filebench [9] to evaluate our system. Filebench supports a workload-modeling language (WML) that allows us to specify minute details of the application behavior. However, filebench also uses random data for its write operations.

To evaluate our hinting mechanism for the block layer deduplication system, we modified filebench to write the output data with a specified duplicate distribution among chunks. Filebench refers to this chunk distribution before issuing a write system call. This chunk distribution is in the form of $\langle$`#Duplicates`,`#Chunks`$\rangle$ pairs. For example, a simple distribution of the form

```
1,183825
2,8238
3,738
...
```

is an input to filebench. It means that filebench writes 183,825 chunks with 1 duplicate or 183,825 unique chunks; 8,238 chunks with two duplicates; and 738 chunks with three duplicates in the duration of one run. It is possible that filebench may not consume all chunks during the workload run. While issuing a write system call, filebench chooses a chunk randomly out of this distribution pool. We configured all threads and processes in the workload to use the same distribution pool.

We obtained the distribution of chunks by analyzing real data using the tool we developed [40]; for the experiments described in this chapter, we used *homes* directory distribution. This is the chunk distribution in the home directories of users on our NFS server. These directories contain a variety of data ranging from source code, binaries, office documents, and virtual machine images.

5.3 Test Description

As mentioned in Section 5.1, the order in which the deduplication system receives chunks is not preserved, because we select them randomly out of a pool. To address this problem and ensure the stability and reproducibility of our results, we report average numbers over three runs, ensuring that standard deviations were less than 5% of the mean.

All experiments were run on the warm meta-data cache. Before starting each experiment, we copied a compiled source tree of a Linux kernel version 3.2.1 and a CentOS 5.7 root directory into the mount point created on the virtual deduplication device. This operation fills up the meta-data cache completely and even adds some entries in the on-disk component of the meta-data store.

For all the experiments, we report the results in the form of bar graphs. We compare `ops/sec` of the block-layer deduplication system *without* hints support to the `ops/sec` of the block layer deduplication *with* hints support. We do this comparison over a wide range of latencies varying from 10μs (micro-seconds) to 10ms (milli-seconds). The 10μs latency emulates a high-end Flash device; the 10ms latency emulates a low-end, much slower mechanical disks. We include intermediate values of latencies of 100μs, 500μs, and 1ms because the physical disk we used in our experiments showed a variation in latency in this range for different workloads. We report CPU utilization for some tests in terms of micro-seconds per operation. The higher the value of micro-seconds per operation, the higher the CPU utilization is.

We used a meta-data cache size of 1GB for most of the micro-benchmarks. For some experiments, we show the plots for both replacement algorithms: LRU and ARC. For some of the experiments, we show the numbers for ARC only. ARC always perform better than LRU. In case

of a deduplication system without hints support, ARC benefits more than LRU. Hence, for the experiments involving ARC, we compare the performance of the deduplication system with hints support with the best possible performance we can get out of deduplication system without hints support that used ARC.

As mentioned in Section 4.2.2, although our implementation supports the BTL cache, there is no penalty on a BTL cache miss. File system hints share the data context with the block-layer deduplication system. BTL manages only the LBA to PBA mappings. Hints regarding data context are not relevant for BTL. Hence, to evaluate the benefits of hints only, we do not account for any BTL cache misses.

We formatted both file systems with a 4KB block size. We left the NILFS2 segment protection period to the default of 1 hour; this means that NILFS2's garbage collection operation runs every 1 hour.

We conducted all the experiments on a machine with an Intel Xeon X5680 3.3GHz CPU and 64GB of RAM. The physical disk used for the experiments was a single Seagate Savvio 15K RPM disk drive. We modified the Linux kernel version 3.2.1 to implement the hinting mechanism and the deduplication system at the block layer. For NILFS2 related experiments, we used nilfs-utils-2.1.1. We installed the CentOS 6.2 distribution on the machine the experiments ran on.

5.4 Benchmark Results

Our goal in this thesis is to show that hints from the file system layer to the block layer deduplication system provide enough context so that the block-layer deduplication system can perform efficiently and that the reliability semantics of the file system are preserved. We would expect to see performance improvement for meta-data–intensive workloads, because we will not be deduplicating them. For journal-intensive workloads, in which data is written twice to the disk, we expect to see better CPU utilization in addition to the performance improvement. Our implementation of the prefetch hint is tightly coupled to `/bin/cp` process, hence we expect to see an improvement in copy time.

In Section 5.4.1, we describe the experiment that we used to confirm our hypothesis about meta-data uniqueness. We describe the micro-benchmarks results for both the file systems: Ext3 and NILFS2 in Section 5.4.2 and Section 5.4.3, respectively. Evaluation of the prefetch hint for both file systems is described in Section 5.4.4. Then, in Section 5.4.5, we describe the results of the fileserver macro-benchmarks for both file systems.

5.4.1 Uniqueness in Meta-data

In this section, we describe one small experiment that we performed to find out the redundancy in the meta-data. In this experiment, we created one 1GB file filled with 0's only. We set up a loop device on this large file using `losetup` and formatted it with the corresponding file system's formatting utilities. We analyzed this disk for unique chunks of 512 byte size and noted down the number of unique chunks for each file system.

Then, we created 1 million empty files in one flat directory on this disk device using filebench. We again analyzed the disk for unique chunks of 512 byte size and noted down the number of unique chunks on the disk after file creation. Table 5.1 shows the findings of our experiments.

File System	Total Chunks on Disk (512 Bytes)	Number of Files Created	Unique Chunks			
			Before File Creation		After File Creation	
			#	%	#	%
Ext3	2,097,152	1,000,000	43	0.002	548,623	26
NILFS2	2,097,152	1,000,000	22	0.001	309,842	14

Table 5.1: Unique 512 byte chunks analysis.

The creation of empty files in a file system increases the percentage of unique chunks by almost 26% in Ext3. This meta-data is written primarily in the directory blocks and the inode tables; see Figure 4.2. For our Ext3 tests, the inode size was of 256 bytes, so two inodes fit in one 512 byte chunk. Each inode consists of an inode number that is unique across file system, thus 500,000 chunks were unique. The remaining 48,000 chunks were mostly occupied by directory blocks consisting of file names and an inode number, both of which are unique across the file system.

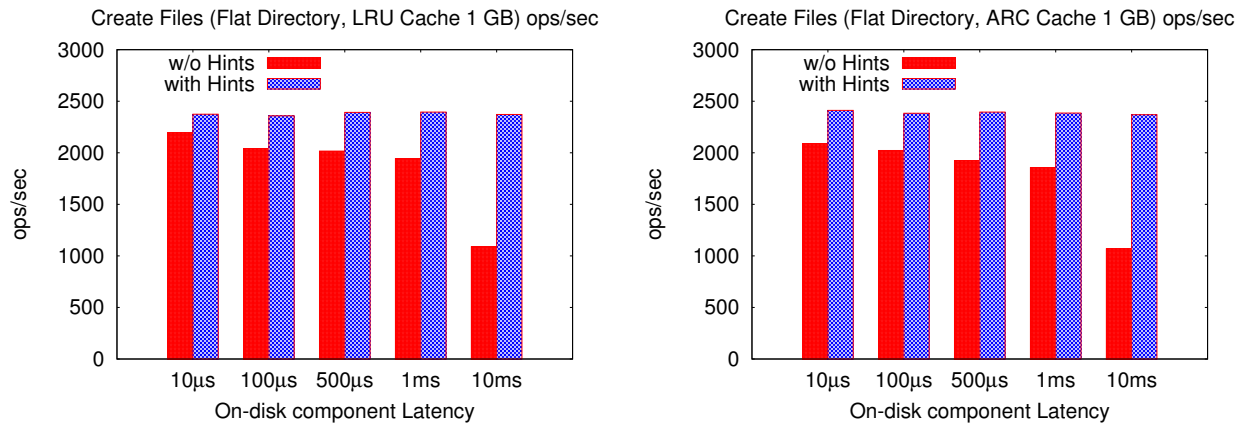
We observed a similar behavior for NILFS2, although the increase in the percentage of unique chunks was smaller (14%) compared to the Ext3 file system (26%). NILFS2 is a continuous-snapshotting file system. It creates a number of checkpoints every few seconds or after a synchronous write. This creates more duplicates: a high redundancy is observed for the chunk size of 512 bytes. In later benchmarking experiments, we observed a similar behavior. As the number of checkpoints increased, we observed some redundancy even in the meta-data managed by NILFS2 for the 512-byte chunk size.

5.4.2 Micro-Benchmarks: Ext3

Meta-data hint. In this section we evaluate the benefits of not deduplicating the meta-data. We ran a workload using filebench in which 16 parallel threads create 1.5 million empty files in one single directory. As the files are empty, only meta-data blocks and directory blocks are written to the disk and no data blocks are written. Thus, the distribution of data chunks used in the experiment does not matter for this experiment. We ran the experiment with meta-data cache size of 1GB and for both the LRU and ARC replacement policies.

From the bar graphs shown in Figure 5.1, it is clear that the hinting mechanism is efficient for all latency values of the on-disk component. For the 10ms latency value, the block layer deduplication system with hint support performs $2\times$ better than the one without hints support. A similar behavior is observed for both cache replacement policies: because all of the meta-data written is unique, ARC act as a simple LRU. For the block-layer deduplication system without hinting support, the deduplication ratio decreased from 1.46 to 1.41. On the other hand, for the deduplication system with hinting support, the deduplication ratio remained constant at 1.17. A decrease in the deduplication ratio for the system without hints support is observed because the Ext3 meta-data exhibited a lot of redundancy in the form of blocks of 0's before the experiment. But, when files are created, these meta-data blocks lost their redundancy.

The deduplication ratio is lower for the system with the hints support compared to the system without hints support. This is because the meta-data blocks with 0's are not deduplicated when the file system was initially formatted and the cache is warmed. The deduplication system with the hints support exhibits a consistent performance, because it bypasses the bloom filter as well as meta-data cache which introduces additional latency overheads. It does not perform any I/O



(a) LRU Cache

(b) ARC Cache

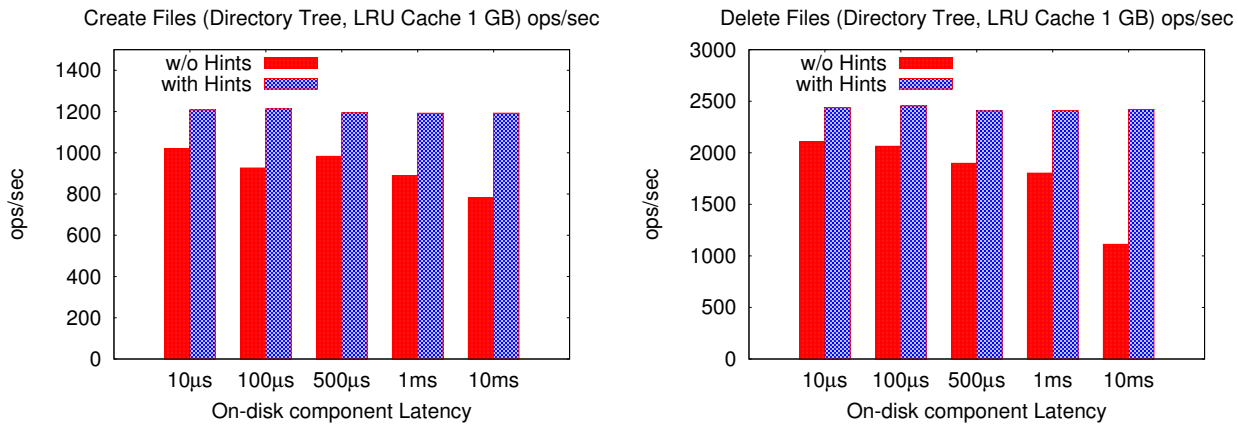
Figure 5.1: Creating Empty Files in a Single directory in Ext3

lookups for duplicate detection. It directly writes meta-data to the underlying physical disk.

Meta-data hint: directory blocks. We categorize the directory blocks as meta-data. Directory blocks in Ext3 contain the name of the file or directory and its inode number. Because the inode number is unique across the file system, these blocks do not exhibit redundancy. We ran an experiment using filebench where 8 threads are creating files and 8 threads are deleting files from the file set consisting of 1.5 million empty files in the directory tree with average directory depth of 1.3. This modifies the directory blocks frequently and there are more directory blocks compared to meta-data blocks. The meta-data cache size was the same at 1GB. The distribution of chunks does not matter here as well because the test operates on empty files. We ran the experiment with both LRU and ARC replacement algorithms. We noted a similar behavior and performance numbers irrespective of cache replacement strategy. This is because the test involved only meta-data, which contains negligible redundancy. Because the workload consists of large number of unique chunks, ARC behaves almost a an LRU.

As shown in Figure 5.2, the behavior for the create file operations is similar to the one observed in case of the flat directory experiment. The create file operation improved by almost 18% for the smallest latency (10μs) and more than 50% for the largest latency (10ms). The delete file operation show improved performance when the block-layer deduplication system receives the hints from Ext3. The improvement in delete throughput is almost 2$\times$ for on-disk component latency value of 10ms. Even for the smallest latency value of 10μs, delete file operations improved by 15%.

The deduplication ratio for the system without hints support degraded from 1.46 to 1.42, but it was constant for the system with the hints support at 1.17. The decrease in the deduplication ratio is smaller compared to the file-creation experiment ran previously. In this experiment, a large number of directory blocks were generated that are hardly filled, because the tree is deeply rooted. For the 4 KB block size of the file system, only the initial 1 or 2 chunks of 512 bytes were found to be occupied in most of the cases; these were the only sectors that kept changing during the experiment. The remaining 6–7 sectors of the directory blocks got deduplicated for every update of the block.



(a) Create Files

(b) Delete Files

Figure 5.2: File Operations in a directory hierarchy in Ext3

Journaling hint: `ordered` mode. Ext3 informs the block layer deduplication system to not deduplicate journal writes. In the `ordered` mode of journaling, Ext3 logs only the meta-data into the journal. Because the meta-data blocks do not exhibit redundancy, they are not worthy of deduplication. We ran a simple experiment in which a single thread is appending 4KB blocks to a single file. After every 100 such appends, the thread calls `fsync`, so that the data is written to the disk and the modified meta-data is written to the journal. We used the *homes* directory distribution for this experiment.

For this experiment, the benefits of our hinting mechanism were observed for the on-disk component latency values greater than or equal to 500μs. When Ext3 uses the `ordered` journal mode, meta-data is written twice to the disk: once in the journal during the commit phase and again at the fixed location in the file system during the check-point phase. For the block layer deduplication system without hints support, the second write gets deduplicated and not written to the disk. It may or may not need a lookup to find out that it is a duplicate write. But, in case of the block layer deduplication system with this hint support, both writes are made to the disk. The performance of the system is then dependent on the on-disk component latency and the disk-write latency.

When the on-disk component latency is smaller than the disk write latency, then performing a lookup is beneficial during the check-point phase. This is because the duplicate of the meta-data being check-pointed definitely exists (it was written during commit phase) and the write during check-point is not required. But, if the on-disk component latency is greater than the disk-write latency, then performing the second write is beneficial. In this particular experiment, we observed that the write latency of the disk we used was in the range of 200–300μs. Thus, a lookup of 10μs or 100μs for the deduplication system without hints support was beneficial because a write during the check-point phase was avoided. But, when the on-disk component latency was higher than the disk write latency, performing a disk write proved more beneficial than performing a lookup to avoid the second disk write.

The degradation in performance for latency values of 10μs and 100μs was around 1–2% for the deduplication system with hints support. The average improvement in the performance for latency values of 500μs, 1ms, and 10ms is around 15%. The deduplication system with hints support used 6% additional disk space compared to the deduplication system without hints support. Similar

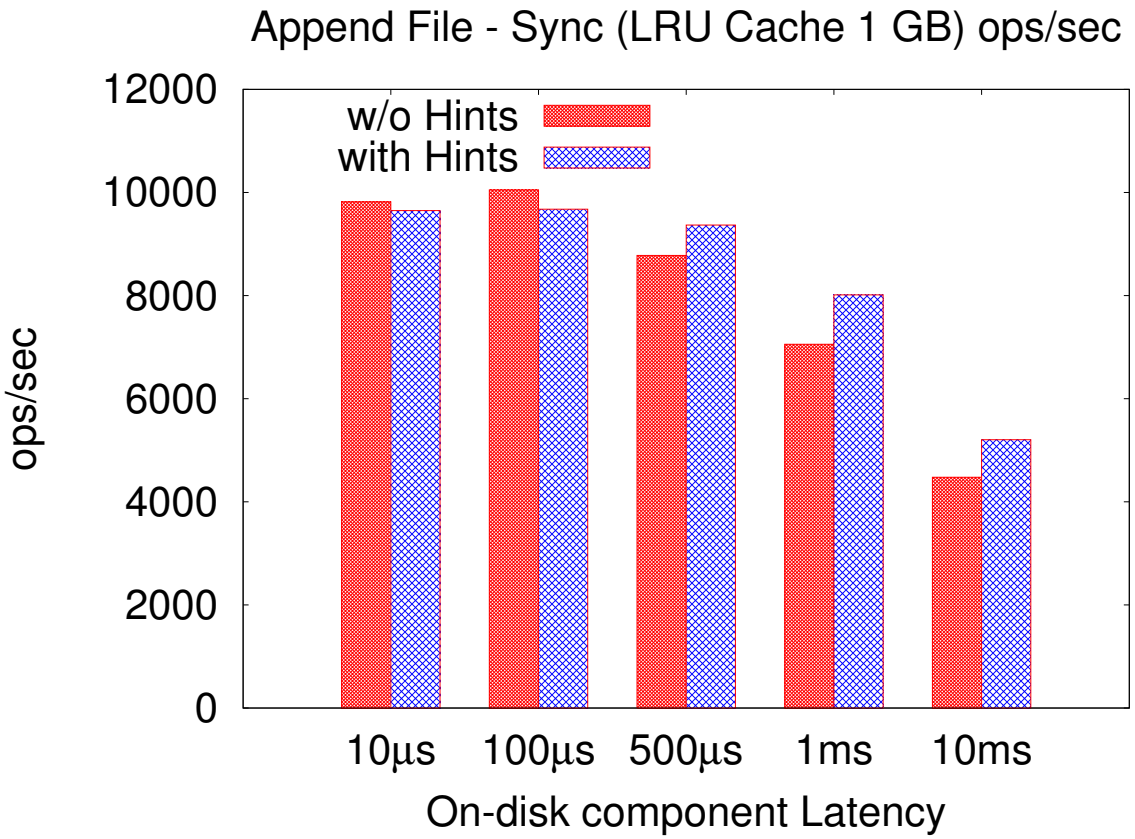


Figure 5.3: Append-sync sequence workload results

behavior is observed for the ARC cache-replacement strategy.

Journaling hint: `journal` mode. In `journal` mode, the Ext3 file system logs the data updates into the journal in addition to the meta-data updates. During the check-point operation, this data in the journal is written back to its fixed position on the disk. This mode of journaling converts random writes to the disk into sequential writes and often improves random-write throughput.

To evaluate the benefits of hints in this journaling mode, we mounted Ext3 using the `-o data=journal` mount option. We ran a workload consisting of 16 threads appending 4KB blocks to 200,000 different files. After 10 appends to a file, we called `fsync` to flush data as well as meta-data updates to the journal. We used the *homes* directory distribution for this test.

We measured the throughput and the CPU usage of the workload. Throughput of the deduplication system with hinting support was 2–3× better than the throughput of the deduplication system without hints support. This is because, with hints support, the duplicate detection logic does not interfere with the writes performed during the commit phase; duplicate detection is performed during the check-point phase. Although the block-layer deduplication system with hints support performs two writes (once during commit phase and once during checkpoint phase, if unique) and one lookup (during the check-point phase), it uses less CPU, because it calculates the hash of the data block only once (during the checkpoint phase). But, the block layer deduplication system without hints support always computes the hash of every block twice: once during commit phase and once during

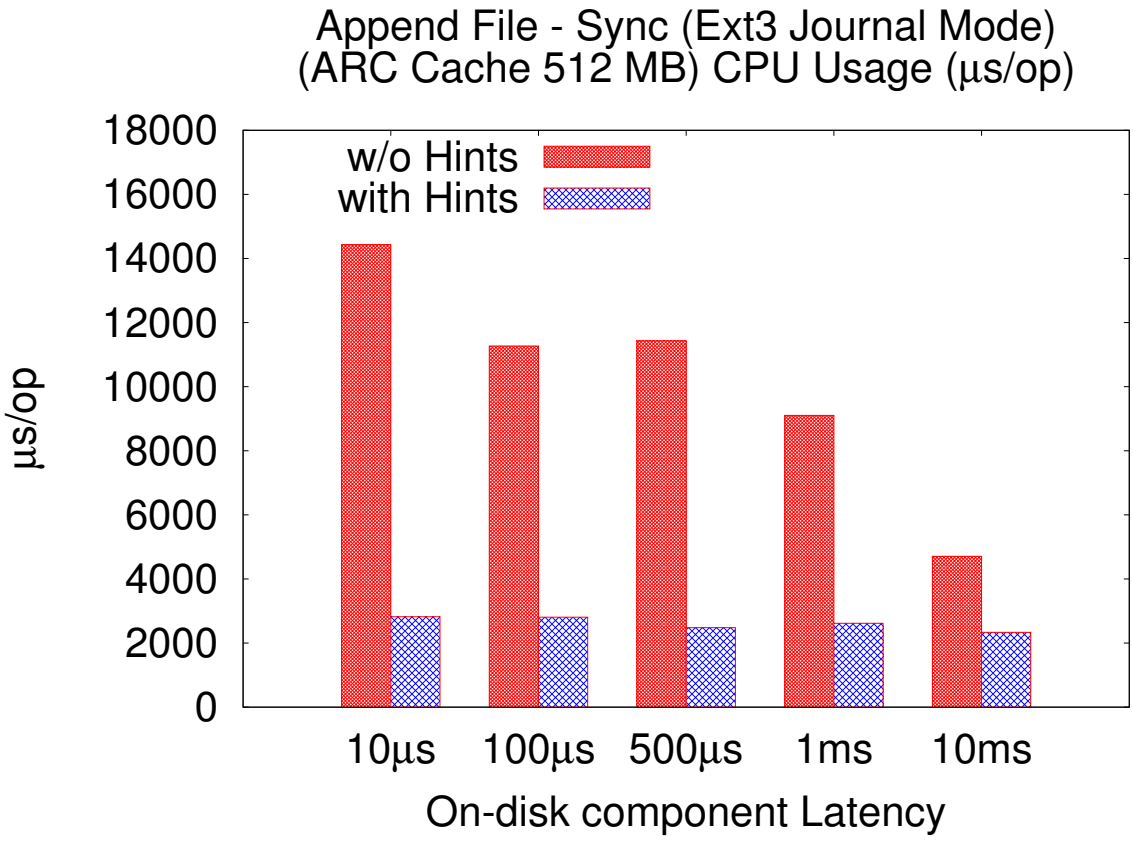
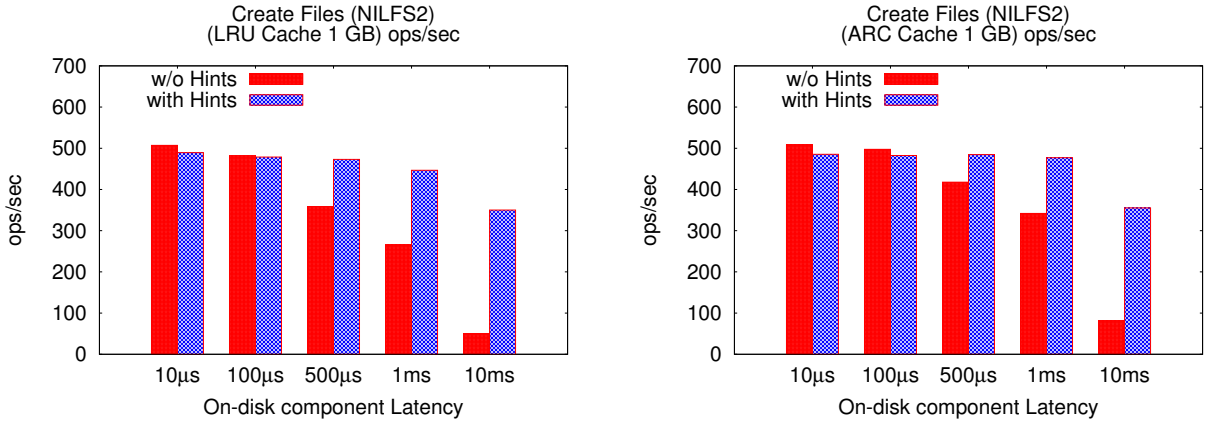


Figure 5.4: Append-sync sequence workload in `journal` mode of Ext3

checkpoint phase. Hence, the deduplication system without hints support gets bottlenecked on the CPU for such a heavy write traffic to the journal. This behavior is observed for all latency values. The deduplication system with hints support used almost 80% fewer CPU cycles for the small latency value (10μs) and almost 50% fewer CPU cycles for larger latency values (10ms) compared to the deduplication system without hint support. We noted only 2% additional disk space usage by the deduplication system with hint support in this experiment.

5.4.3 Micro-Benchmarks: NILFS2

Meta-data hint. In this section we evaluate the benefits of not deduplicating the NILFS2 meta-data. Using filebench, we ran a workload consisting of 8 parallel threads that create 150,000 1-byte files in one single directory. NILFS2 frequently creates the checkpoints on the disk every few seconds. This creates a large number of snapshots and the disk fills up quickly. Hence, we created a smaller number of small files compared to the Ext3 experiments. Another interesting observation is that NILFS2 manages a B-Tree node cache using the Linux `address_space` operations. Because the meta-data amount is small and our test machine has a large RAM, NILFS2's meta-data was not getting written to the disk regularly, but in bursts. We needed to issue explicit `sync` calls to flush this meta-data onto the disk. To avoid dependencies of the results on the frequency of `sync` calls, we used a file size of 1 byte instead of empty files in the experiment. As the page cache starts getting filled up, data and the meta-data were getting written to the disk regularly. We used the *homes* directory distribution for the test.



(a) LRU Cache

(b) ARC Cache

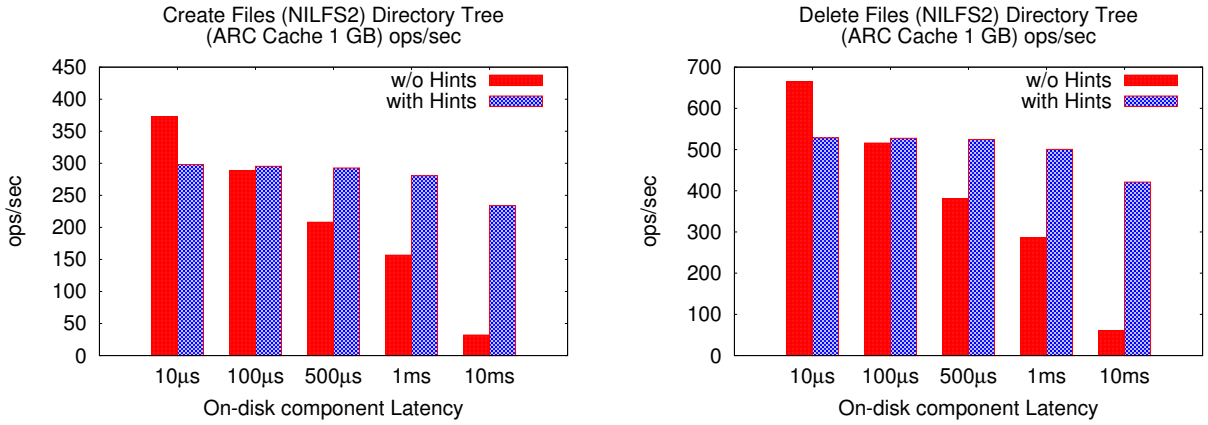
Figure 5.5: Creating 1-byte Files in a Single directory on the NILFS2 File System

As mentioned in the Section 5.4.1, NILFS2 meta-data does exhibit some redundancy. In addition to this, the frequent checkpointing mechanism in the NILFS2 generates lots of duplicates of the meta-data. We observed that while this experiment was running, NILFS2 created almost 140,000 checkpoints. Hence, deduplicating this meta-data using small lookup latencies like 10μs and 100μs proved beneficial for the deduplication system without hints support. But, when the lookup latency for the on-disk component starts increasing beyond the physical disk-write latency (100–500μs), the deduplication system with hints support starts performing more than 2$\times$ better. This behavior is observed for both cache replacement policies.

One of the important purposes of the meta-data hint is to ensure the file system's reliability. Not deduplicating the meta-data is consistent with the file system like NILFS2, which implements a checkpointing mechanism for reliability purposes. There is only 3–5% performance penalty observed for the smaller latency values of 10μs and 100μs. This penalty is acceptable in order to ensure the file system reliability guarantees.

The deduplication ratio for the system without hints support was 2.15, but that of the system with hints support was only 1.06. All the redundancy introduced by NILFS2 for reliability is removed by the deduplication system without hints support.

Meta-data hint: directory operations. Similar to the Ext3 file system, NILFS2 passes the hint to the block-layer deduplication system to not to deduplicate the directory blocks. NILFS2 directory blocks primarily contain file or directory names and the inode number, which is unique across the file system. So, directory blocks are not worthy of deduplication. Using filebench, we configured a workload in which 4 threads are creating files and 4 threads are deleting files from a file-set consisting of 500,000 files. All files are of size 1 byte and are created and deleted from the directory tree with average directory depth 2.1. This test, thus, modifies a large number of directory blocks frequently. We used the *homes* directory distribution and used the ARC replacement policy for the meta-data cache.



(a) Create Files (b) Delete Files

Figure 5.6: File Operations in a directory hierarchy for NILFS2

For small lookup latency of 10μs, the deduplication system without hints support performs almost 20% better than the deduplication system with hints support. Because the directory tree used in the experiment was deeply rooted, there are a small number of files or directories within a single directory. Thus, for the file system formatted with 4KB block size, only 1 or 2 chunks of 512 bytes were found to be occupied by the file or directory names and their corresponding inode numbers. These are the only sectors that kept changing during the test. The remaining chunks, almost 6 to 7 chunks of the directory block, are not changed and got deduplicated on every update. NILFS2 keeps creating checkpoints every few seconds. So, copies of these 6 to 7 sectors per directory block started accumulating on the disk and this increased the redundancy in the directory blocks by a significant amount. Deduplicating this redundant blocks proved beneficial for smaller lookup latencies. But, when the lookup latency is greater than the disk write latency (100μs), the

benefits of hints support to the deduplication system manifest. We see more than 2× improvement in performance of create and delete operations when the on-disk component latency value is 10ms.

The deduplication ratio of the system without hints support was 3.17; this was higher compared to the 1.06 deduplication ratio of the system with hints support. During this experiment, NILFS2 created 240,000 checkpoints. The checkpoint operation in NILFS2 created a lot of data duplicates as well as meta-data ones.

5.4.4 Micro-Benchmark: Prefetch Hint

In the context of the `/bin/cp` process, both Ext3 and NILFS2 pass the prefetch hint to the block layer deduplication system to prefetch the hash of the block being read. This is because `/bin/cp` creates a duplicate of the block shortly after the read; when the duplicate is written, the hash is available in the meta-data cache. This avoids expensive lookups in the on-disk component of the meta-data store.

We ran the same experiment for both file systems using both LRU and ARC, and a cache size of 64MB. There was no parallel workload running while `/bin/cp` was running. There are 3 steps for this experiment:

1. We copy the compiled sources of all file systems in the Linux kernel into the mount point created over our virtual deduplication device. This fills up the meta-data cache and also pushes some cache entries onto the on-disk component.

2. Then, we write unique data into the mount point. The amount of data written is large enough to replace all the hash entries for the kernel data written in step 1 from the meta-data cache. We `sync` the page cache to the disk and then drop the caches using

   ```
   # echo 3 > /proc/sys/vm/drop_caches
   ```

3. Lastly, we issue the `/bin/cp` command to create the copy of the kernel data written in step 1 in the same mount point. After the copy is finished we issue the `sync` command to flush all data into the disk. We note the time required to copy the data.

The compiled sources of file system code we used from the Linux kernel 3.2.1 occupies 554MB of data. There were 3,767 files with an average size of 150KB. The physical machine on which the experiment ran had 2GB of RAM. Using a smaller RAM ensures that the pages written by the `/bin/cp` process are flushed frequently to the disk and we see noticeable difference in copy time.

Table 5.2 lists the results of this experiment for LRU cache replacement strategy. The improvement in the copy time is around 25% for the on-disk component latencies of 500μs, 1ms, and 10ms, for Ext3. For NILFS2, the improvement was approximately 15–20% for on-disk component latency values of 500μs and 1ms.

One point to note is that the meta-data hint and the journaling hint were not enabled in this experiment. So, any meta-data writes or journal writes forced prefetched entries to be replaced. NILFS2 generates lot of meta-data and hence would evict a large number of prefetched entries. This is why we see a poor copy performance for NILFS2 compared to Ext3. Meta-data and journal writes replace prefetched entries from the meta-data cache. Hence, when the deduplication system

On-Disk Component Latency	Ext3 File System			NILFS2 File System		
	w/o Hints	with Hints	Improvement	w/o Hints	with Hints	Improvement
10μs	0.74	0.65	12%	0.79	0.53	32%
100μs	1.91	1.15	39%	2.35	1.31	44%
500μs	4.25	3.13	26%	9.66	7.26	25%
1ms	8.73	6.55	25%	18.35	15.13	17%
10ms	78.28	58.09	25%	177.31	161.14	9%

Table 5.2: Time required (In Minutes) for copying 554MB of data

receives an actual write request for the duplicate generated by `/bin/cp`, that write request experiences a cache-miss. Lookup into on-disk component is required to confirm the redundancy of such a write request. Hence, for larger latency values of the on-disk component, the improvement in the copy time is small. Also, the order in which pages are selected for flushing by `pdflush` also affects the meta-data cache behavior of the deduplication system at the block layer. For NILFS2, the improvement in copy time decreased to 9% for the 10ms latency, because over the period of 3 hours more meta-data is generated in the form of a number of checkpoints taken.

The results looked similar for the ARC replacement strategy, because we cleared up the meta-data cache in step 2 of the experiment by writing unique data and flushing it to the disk. Every write from `/bin/cp` almost turned out to be either unique or a cache-miss due to small cache-size, and ARC behaved almost as an LRU.

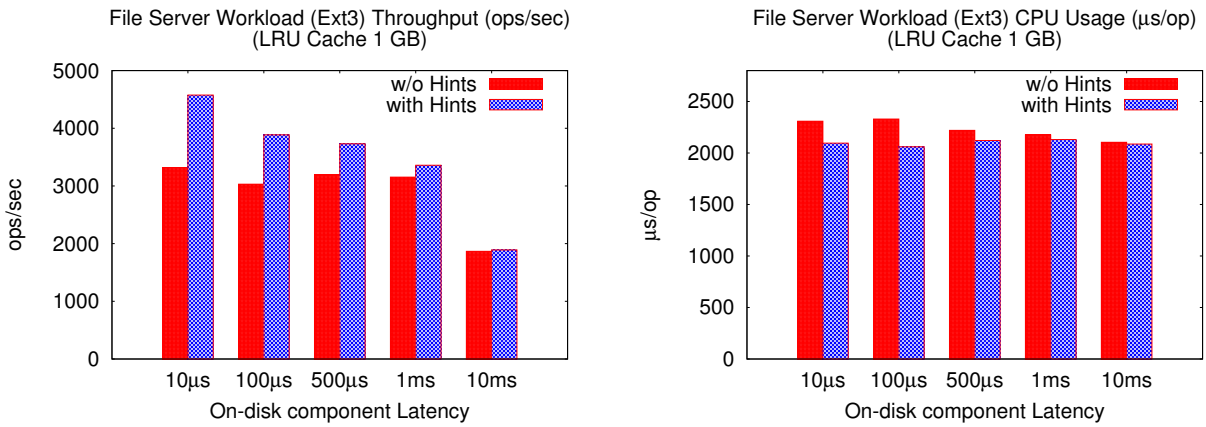
5.4.5 Macro-Benchmarks

In this section we evaluate the deduplication system with hints support using general purpose macro-benchmarks. Many realistic workloads can be configured using Filebench. We chose the fileserver workload to evaluate the hinting mechanism, because we had the duplicate chunk distribution for the *homes* directories on an NFS file server. Also, this workload is a good representation of what an in-line deduplication system might see.

The fileserver workload consists of 50 threads operating on 10,000 files with a mean file size of 128KB. Each thread performs a range of operations on files like creating a file, writing the whole file, appending to the file, reading back the whole file, deleting files, collecting file statistics, etc.

We analyzed how the deduplication system with hints performs in terms of throughput and CPU usage, compared to a deduplication system without hints support. We analyzed both file systems: Ext3 and NILFS2. For this workload test, we configured Ext3's journal in the `ordered` mode. NILFS2 was mounted with `dedupongc` mount option.

Ext3 file system Figure 5.7 shows the results of our benchmark test for Ext3. For Ext3, we found out that the deduplication system without hints support was bottlenecked on the CPU for lower on-disk component latency values. The physical machine used for the test had 24 cores and 50 threads were competing for the CPU. Threads performing write operations require us to calculate an MD5 hash of the block being written. The deduplication system with hints support was using 10–12% fewer CPU cycles and performed 35–40% better than the deduplication system without hints support. Because the deduplication system with hints support does not deduplicate the meta-



(a) Throughput

(b) CPU Usage

Figure 5.7: File Server Workload on Ext3

data and journal blocks, it uses fewer CPU cycles. The deduplication system without hints support was unable to leverage the benefits of using on-disk component with the smallest latency value, due to a higher CPU utilization. Although these plots are only for the LRU cache replacement policy, we observed a similar behavior with the ARC cache replacement policy. This is because this workload involves a significant number of meta-data updates. The deduplication system without hints support becomes CPU bound for such workload. Hence, deduplication system without hints support could not leverage the benefits of the intelligent caching used by the ARC algorithm. The deduplication system with hints support bypasses the meta-data cache for meta-data updates, and hence does not get affected significantly by changes in cache replacement policy.

As the latency of the on-disk component starts increasing, deduplication systems with and without hints support were bottlenecked on I/O—specifically lookups into the on-disk component. Throughput and CPU usage of both systems were comparable for the on-disk component latency of 10ms. But, the deduplication system with hints support ensured the reliability of the meta-data.

We noted 6.3% additional disk space usage by the deduplication system with hints support compared to the deduplication system without hints support for the average data set size of 800MB. This was an acceptable trade-off for us to get better performance and better CPU utilization.

NILFS2 file system The performance characteristics of the fileserver workload on NILFS2 with hints support are similar to those of the journaling hint for NILFS2. Because the file system was mounted with `dedupongc` option, more than 2× performance improvement was seen for the 10ms latency value. Thanks to some duplication even in the meta-data, deduplicating this meta-data for lower latency values of the on-disk component (10μs) was beneficial.

Figure 5.8 shows the CPU utilization when for the deduplication system with and without hints support. The CPU utilization was almost similar in both cases. The reason for this is an implementation bottleneck in NILFS2. NILFS2 always writes the data and the meta-data sequentially in the log. There is a single thread, `segctord`, which writes this log into the segment. Thus, all writes to the deduplication disk device get serialized at this point in NILFS2, and the deduplication system with and without hints support ends up using the same amount of CPU.

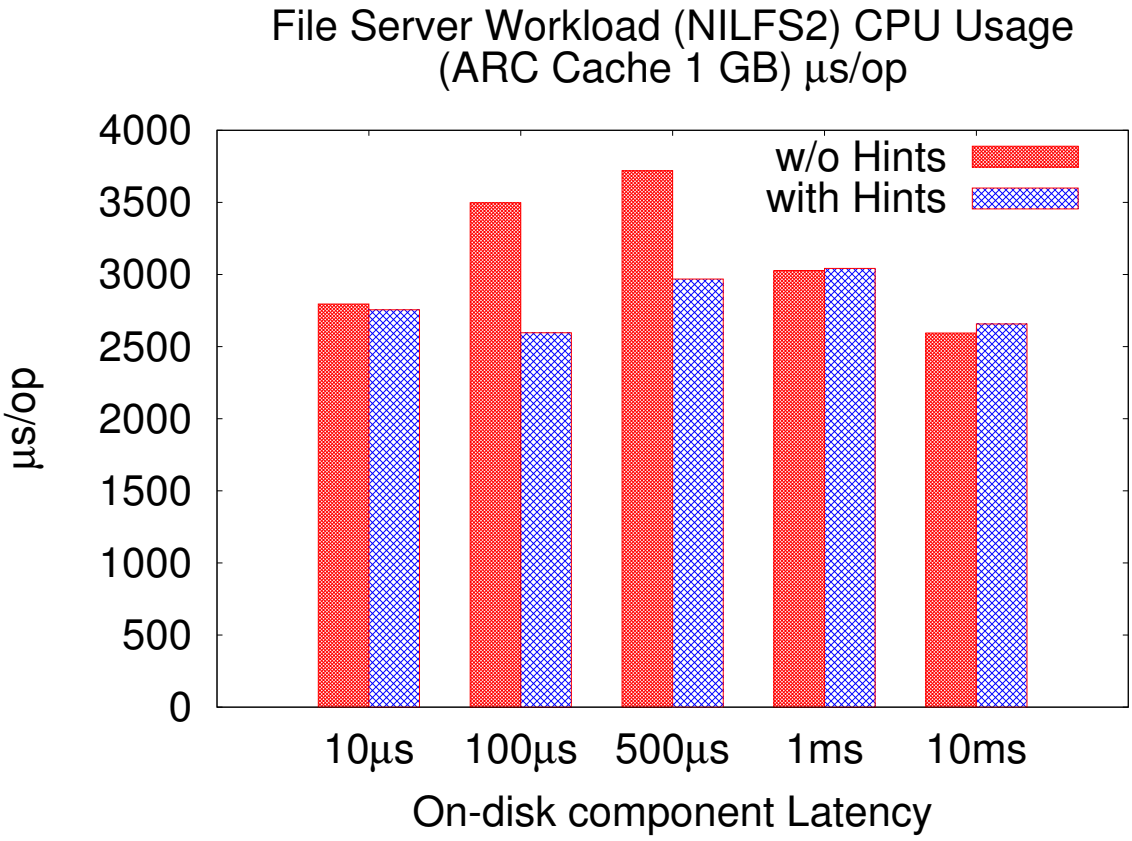


Figure 5.8: File Server Workload on NILFS2

Evaluation summary. Our micro-benchmark results for both file systems show the benefits of individual hints passed by file systems. Our macro-benchmark results show that hinting mechanism is performance efficient even in the real world scenarios. Our context-aware deduplication system performs 40% better for Ext3 and almost 2× better for NILFS2 for a realistic fileserver workload compared to its context-unaware counterpart. We see 10–12% improvement in the CPU utilization for Ext3 with hints support. Hints also provide an opportunity to file systems to ensure their reliability semantics.

Chapter 6

Related Work

Data deduplication is one of the important techniques to improve storage efficiency by eliminating duplicates in data. Redundancies in the data can be seen in a variety of places like backup and archived data, virtual disk images, collaborative work environments, etc. There are different optimizations in the deduplication solution possible based on the availability and efficiency of the resources, performance requirements, and specific characteristics of the workload or the data.

For the backup and archived workload, Zhu et al. [46] implement a prefetching technique for their deduplication file system. They take advantage of the observation that chunks tend to reappear in almost the same sequence as seen in the previous backup. When a duplicate of a particular segment, say S, is detected, then with high probability, other segments in the same stream are duplicates of segment S's neighbors. This duplicate locality observed in the backup workload serves as a hint to the deduplication system so that it prefetches the hashes of the neighbors of segment S into RAM.

ADMAD [7] is an example of an application-driven deduplication system for archival storage. This system implements a variant of variable chunking so that chunk boundaries are decided based on the meta-data position in the file. For example, in the HTML file, chunk boundaries are placed in such a way that semantic information like HTML tags is preserved after chunking. This system tries to reduce inter-file–level duplications by leveraging the meta-data information.

NetApp's iDedup [37] implements selective deduplication by deduplicating only a sequence of blocks instead of individual blocks. This system leverages the spatial locality among the duplicate blocks on the disk. It performs the deduplication only if the duplicate of each of the blocks in the sequence is placed sequentially onto the disk. The sequence of blocks to be deduplicated must be sequential in the file as well. This reduces the disk fragmentation and amortizes the random seeks incurred during read processing. In-line deduplication in log-structured file systems for primary storage [13] takes the similar approach. A group of consecutive file-system data blocks are deduplicated only if another group having blocks in the same consecutive order is duplicate. This prevents additional time delays in the write-paths and reduces fragmentation on the disk, but reduces the dedup ratio. Both approaches expose a trade-off between capacity savings and performance.

In-line file-system–based deduplication solutions like ZFS [38], Lessfs [16], or SDFS [33] have enough context available at their layer in the storage stack hierarchy, so that deduplication process at their respective layers can be optimized. But, FUSE-based solutions like Lessfs and SDFS incur performance overheads. The ZFS port for Linux [45] is implemented using a FUSE interface and

hence suffers from the same problem.

The block layer of the Linux kernel implements a set of flags using the `bi_flags` field in the I/O request structure. This field is typically used by the file system layer to convey additional information to the block layer. Some examples of these flag bits are REQ_RAHEAD, a read ahead request; REQ_SYNC, a synchronous read and write request; or REQ_META, a meta-data I/O request. Our hinting mechanism adds two flag bits to this field, indicating (1) whether to deduplicate a given request or not (REQ_NODEDUP) and (2) whether to prefetch the hash of the block being read (REQ_PREFETCH).

Lakshmanan [15] implements file system hints to reduce the failure recovery time for existing file systems. They place the meta-data blocks of the file system together; in an event of recovery, `fsck`'s seek latency can be controlled. Simple hints from the file system differentiate between the meta-data block and data blocks, so that the block layer can cluster the meta-data blocks together.

Type-Safe Disks (TSDs) [35] also maintain semantic information about the disk blocks in terms pointer relationships between them imposed by the file systems. This semantic information allows the system to apply concepts of type-safety in the context of the disk storage. To obtain these pointer relationships, a TSD separates meta-data blocks from the data blocks. Knowledge about the pointer relationships among disk blocks is useful in wide-range of applications like capability-based access control at the disk level, secure deletion, intelligent data placement, etc.

Prabhakaran et al. [23] perform semantic block analysis to evaluate journaling file systems. They leverage the information about the block type in the read and write request to understand the internal behavior of the file systems. Semantic block analysis combines the information from block traces with the semantic information about the blocks. Semantic information of the block, used in the analysis, includes whether the block is an inode block, a journal block, or a data block. Combining this semantic information with block trace provides insights about the internal behavior of the file system.

Chapter 7

Conclusions

Performance is one of the critical aspects of any in-line and primary storage-deduplication systems. The performance characteristics as well as the implementation complexity of in-line and primary storage deduplication systems depend on the position of the system in the storage-stack hierarchy. In this thesis, we demonstrate an efficient and simple implementation of a prototype deduplication system at the block layer of the Linux kernel.

The block layer of the storage stack hierarchy is unaware of the context of the data received from file systems above it. A context-unaware deduplication system at the block layer might end up deduplicating file system meta-data it should not deduplicate (e.g., superblocks), thereby hurting the reliability of the file system. It might also deduplicate short-lived data that is not worthy of deduplication in a given context, thereby hurting performance of the system.

To address this problem, we developed our block layer deduplication system prototype with hints from the file system layer. These hints add negligibly small overhead and convey useful semantic information about those I/O requests to the block layer deduplication system. Using this semantic information, block layer deduplication systems perform more efficiently and ensure the file system's reliability.

Our hinting interface is generic and simple enough so that it can be implemented for any file system without major functional modifications into the file system code. We evaluated our hinting mechanism for two different types of file systems: Ext3 and NILFS2. We confirmed the performance improvement and file system reliability from these experiments.

Future Work In the near future, we plan to prepare our code for release and provide maintenance support for some period of time. We also plan to implement hints for file systems to encourage more users to use this system. This will give us an idea about what other kinds of hints users need and will help us increase the usefulness of this hinting mechanism. This will also strengthen the acceptance of our system into the Linux kernel.

The hinting interface for the deduplication system will prove valuable in virtualized environments. For example, in virtualized environments, implementing an in-line deduplication system at the hypervisor layer would be beneficial because of the redundancy exhibited by virtual machines. A hinting mechanism will allow the guest operating systems to communicate the context of the data being read or written to the hypervisor layer. A meta-data hint could be one of the first implementations. Typically, hypervisors virtualize a large flat file into a block disk device for the guest operating system. The guest operating systems can pass a hint to not to deduplicate the meta-data

writes so that a deduplication system at the hypervisor layer would ignore such write requests to the file. This will be helpful to improve the performance of the deduplication system at hypervisor layer. Selective deduplication of this form can be thought in one more scenario: writes to the swap-device in the guest operating system are short-lived and thus should not be deduplicated.

Other than virtualized environments, any system or application that deals directly with the block layer and bypasses the file system, can also benefit from data deduplication. A hinting interface offers the flexibility to such applications so that they can selectively deduplicate or not deduplicate any desired data.

Bibliography

[1] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *HotStorage '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage*, June 2011.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] R. E. Bohn and J. E. Short. How much information? 2009 report on american consumers. `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf`, December 2009.

[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *ECA*, 2006.

[5] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 33(1):157–168, June 2005.

[6] Ashish Chawla, Benjamin Reed, Karl Juhnke, and Ghousuddin Syed. Semantics of caching with spoca: a stateless, proportional, optimally-consistent addressing algorithm. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[7] Chuanyi Liu and Yingping Lu and Chunhui Shi and Guanlin Lu and Du, D.H.C. and Dong-Sheng Wang. ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System. In *Storage Network Architecture and Parallel I/Os, 2008. SNAPI '08. Fifth IEEE International Workshop on*, pages 29 –35, sept. 2008.

[8] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *Proceedings of the USENIX Annual Technical Conference (ATC '10)*, Boston, MA, USA, June 2010.

[9] Filebench. `http://filebench.sourceforge.net`.

[10] Advanced Storage Products Group. Identifying the hidden risk of data deduplication: how the HYDRAstor solution proactively solves the problem. Technical Report WP103-3_0709, NEC Corporation of America, 2009.

[11] M. Halcrow. eCryptfs: a stacked cryptographic filesystem. *Linux Journal*, 2007(156):54–58, April 2007.

[12] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[13] Stephanie N. Jones. Online De-duplication in a Log-Structured File System for Primary Storage. Master's thesis, University of California, Santa Cruz, 2011. Technical Report UCSC-SSRC-11-03.

[14] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[15] Vivek Lakshmanan. Exploiting File System Awareness for Improvements to Storage Virtualization. Master's thesis, University of Toronto, 2009.

[16] Lessfs, January 2012. `http://www.lessfs.com`.

[17] Nimrod Megiddo and Dharmendra Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 115–130, San Francisco, CA, March 2003. USENIX Association.

[18] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[19] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, pages 1–1, San Jose, CA, February 2011. USENIX Association.

[20] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001. ACM.

[21] NetApp. NetApp deduplication for FAS. Deployment and implementation, 4th revision. Technical Report TR-3505, NetApp, 2008.

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

[23] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Analysis and evolution of journaling file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 105–120, Anaheim, CA, April 2005. USENIX Association.

[24] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 89–101, Monterey, CA, January 2002. USENIX Association.

[25] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[26] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[27] Device mapper Resource page. `http://sources.redhat.com/dm/`.

[28] LVM2 Resource page. `http://sources.redhat.com/lvm2/`.

[29] H. V. Riedel. The GNU/Linux CryptoAPI site. `www.kerneli.org`, August 2003.

[30] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. In *Internet Activities Board*. Internet Activities Board, April 1992.

[31] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[32] C. Saout. dm-crypt: a device-mapper crypto target. `www.saout.de/misc/dm-crypt/`, January 2005.

[33] Opendedup - SDFS, January 2012. `http://www.opendedup.org`.

[34] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads extensions. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[35] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[36] M. Smith, J. Pieper, D. Gruhl, and L. Real. IZO: Applications of large-window compression to virtual machine management. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2008.

[37] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[38] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[39] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[40] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, June 2012. USENIX Association.

[41] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[42] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[43] A. Yoshiji, R. Konishi, K. Sato, H. Hifumi, Y. Tamura, S. Kihara, and S. Moriai. Nilfs, 2009.

[44] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[45] ZFS for Linux, January 2012. `http://www.zfs-fuse.net`.

[46] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, USA, 2008.