Design and Implementation of an Open-Source Deduplication Platform for Research

A RESEARCH PROFICIENCY EXAM PRESENTED
BY
SONAM MANDAL
STONY BROOK UNIVERSITY

Technical Report FSL-15-03
December 2015

Abstract

Data deduplication is a technique used to improve storage utilization by eliminating duplicate data. Duplicate data blocks are not stored and instead a reference to the original data block is updated. Unique data chunks are identified using techniques such as hashing, and an index of all the existing chunks is maintained. When new data blocks are written, they are hashed and compared to the hashes existing in the hash index. Various chunking methods exist, like fixed chunking or variable length chunking, which impact the effectiveness of identifying duplicates.

Deduplication in storage systems can be done as a post-processing step, after the data has already been written to the storage device, or it can be performed in-line (i.e., before writing the data to the storage system). Deduplication can be implemented at various layers in the storage stack, such as application layer, file system layer, or block layer, each having their own trade-offs.

In earlier deduplication studies, often researchers had to either implement their own deduplication platform, or had to use closed-source enterprise implementations. Due to this difficulty in finding an appropriate deduplication platform for research, we designed and developed our own deduplication engine and decided to open-source it. We also used our deduplication engine for further deduplication research.

We present *Dmdedup*, a versatile and practical primary-storage deduplication platform suitable for both regular users and researchers. Dmdedup operates at the block layer, so it is usable with existing file systems and applications. Since most deduplication research focuses on metadata management, we designed and implemented a flexible backend API that lets developers easily build and evaluate various metadata management policies. We implemented and evaluated three backends: an in-RAM table, an on-disk table, and an on-disk COW B-tree. We have evaluated Dmdedup under a variety of workloads and report the evaluation results here. Although it was initially designed for research flexibility, Dmdedup is fully functional and can be used in production. Under many real- world workloads, Dmdedup's throughput exceeds that of a raw block device by 1.5–6×. We have also submitted Dmdedup to the Linux community for inclusion in the Linux kernel mainline.

Block-layer data deduplication allows file systems and applications to reap the benefits of deduplication without modifying each file system or application to support deduplication. However, important information about data context (e.g., data vs. metadata writes) is lost at the block layer. Passing such context to the block layer can help improve deduplication performance and reliability. We implemented a hinting interface to pass relevant context to the block layer, and evaluated two hints, NODEDUP and PREFETCH. To allow upper storage layers to pass hints based on the available context, we modified the VFS and file system layers to expose a hinting interface to user applications. Building on our open-source block-layer deduplication system, *Dmdedup*, we show that passing the NODEDUP hint can speed up applications by up to 5.3× on modern machines because the overhead of deduplication is avoided when it is unlikely to be beneficial. We also show that the PREFETCH hint can speed applications by up to 1.8× by caching hashes for data that is likely to be accessed soon.

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As storage demands continue to grow [5], even continuing price drops have not reduced total storage costs. Removing duplicate data (deduplication) helps this problem by decreasing the amount of physically stored information. Deduplication has often been applied to backup datasets because they contain many duplicates and represent the majority of enterprise data [44, 71]. In recent years, however, primary datasets have also expanded substantially [35], and researchers have begun to explore *primary* storage deduplication [33, 57].

Primary-storage deduplication poses several challenges compared to backup datasets: access locality is less pronounced; latency constraints are stricter; fewer duplicates are available (about 2× vs. 10× in backups); and the deduplication engine must compete with other processes for CPU and RAM. To facilitate research in primary-storage deduplication, we developed and here present a flexible and fully operational primary-storage deduplication system, *Dmdedup*, implemented in the Linux kernel. In addition to its appealing properties for regular users, it can serve as a basic platform both for experimenting with deduplication algorithms and for studying primary-storage datasets and workloads. In earlier studies, investigators had to implement their own deduplication engines from scratch or use a closed-source enterprise implementation [6, 57, 64]. Dmdedup is publicly available under the GPL and we submitted the code to the Linux community for initial review. Our final goal is the inclusion in the mainline distribution.

Deduplication can be implemented at the file system or block level, among others. Most previous primary-storage deduplication systems were implemented in the file system because file-system-specific knowledge was available. However, block-level deduplication does not require an elaborate file system overhaul, and allows any legacy file system (or applications like databases) to benefit from deduplication. Dmdedup is designed as a stackable Linux kernel block device that operates at the same layer as software RAID and the Logical Volume Manager (LVM).

Most deduplication research focuses on metadata management. Dmdedup has a modular design that allows it to use different *metadata backends*—data structures for maintaining hash indexes, mappings, and reference counters. We designed a simple-to-use yet expressive API for Dmdedup to interact with the backends. We implemented three backends with different underlying data structures: an in-RAM hash table, an on-disk hash table, and a persistent Copy-on-Write B-tree. In this report, we present Dmdedup's design, demonstrate its flexibility, and evalutate its performance, memory usage, and space savings under various workloads and metadata backends.

One drawback with implementing deduplication at the block layer is that the deduplication system is unaware of the context of the data it operates on. A typical I/O request contains only the operation type (read or write), size, and offset, without attached semantics such as the difference between metadata and user data. Most existing solutions are built into file systems [7, 53, 58] because they have enough information to deduplicate efficiently without jeopardizing reliability. For example, file sizes, metadata, and on-disk layout are known to the file system; often file systems are aware of the processes and users that perform I/O. This information can be leveraged to avoid deduplicating certain blocks (e.g., metadata), or prefetch dedup

metadata (e.g., for blocks likely to be accessed together).

File systems know which writes are metadata writes, and with this information deduplicating metadata can be avoided for improved reliability and performance. Deduplicating metadata can (1) potentially harm reliability [51], e.g., because many file systems intentionally save several copies of critical data such as superblocks, and (2) waste computational resources because typical metadata (inode tables, directory entries, etc.) exhibits low redundancy. In particular, in-line deduplication is expensive because forming fixed or variable-length chunks, hash calculation, and hash searches are performed before writing data to disk; it is undesirable to expend resources on data that may not benefit from deduplication. Also, the upper I/O layers often have a better idea whether a given block will deduplicate well, and that information can be useful at the block layer, especially since attempting to deduplicate unique or short-lived data will only harm performance.

To allow block-layer deduplication to take context into account, we propose an interface that allows file systems and applications to provide simple *hints* about the context in which the deduplication is being performed. Such hints require only minor file system changes, making them practical to add to existing, mature file systems. We implemented two hints: NODEDUP and PREFETCH, which we found useful in a wide range of cases. We leveraged Dmdedup to show the usefulness of this hinting interface and to show the utility of having an open-source platform for deduplication research.

The rest of this report is organized as follows. Chapter 2 discusses the background for deduplication. Chapter 3 discusses the design of Dmdedup and its metadata backends. Chapter 4 covers the evaluation of our Dmdedup system, using micro-workloads, macro-workloads, and compare it to other deduplication systems such as Lessfs. Chapter 5 details the design and implementation of our hinting interface. Chapter 6 discusses the evaluation of our hinting interface. Chapter 7 covers related work. Finally, chapter 8 concludes and discusses future work regarding our Dmdedup and hinting system.

Chapter 2

Background

The amount of data that modern organizations and users need to store and transfer grows rapidly [5, 16]. Increasing physical storage sizes and deploying higher-throughput networks are brute-force solutions to satisfy the needs of organizations. However, the brute-force approaches are expensive and can significantly hamper the speed of business development. Another method for improving storage efficiency is to compress the data so that less physical resources are required for storing and transfering the data. Local compression was used to increase storage efficiency for decades [26]. However, local compression eliminates repetitive byte strings only within a limited data window and consequently is not effective for detecting duplicates in large datasets.

About 15+ years ago a new technology called *data deduplication* started to gain popularity. The key insight that makes deduplication effective is that many real datasets contain a lot of identical objects scattered accross the entire datasets. Unlike compression, which looks for duplicates among short byte sequences (typically, few tens of bytes long) within a small window (e.g., 100–900KB for bzip2 [8]), deduplication targets large duplicated objects (typically, of several kilobytes size) within a whole dataset. After detecting the duplicates, a deduplication system replaces them by the pointers to the corresponding single instances. One can even apply compession on top of deduplication to achieve the highest level of data reduction.

The ultimate goal of deduplication, as with almost many other computer systems, is to reduce the cost of the IT infrastructure. Deduplication can reduce the cost by:

1. Reducing the amount of data that need to be stored;
2. Improving the performance of a system; and
3. Extending the lifetime of the devices.

Initially the goal of deduplication was to reduce the amount of data stored, which in turn allows to purchase and power fewer storage devices. Even nowdays space savings remain the main reason for deduplication to be applied. With the advent of virtualization and cloud technologies, we can see memory deduplication arising as a consequence. This allows different virtual machines to share resources, like code and shared libraries, and allow for better utilization of memory. In certain cases deduplication can also improve performance because the amount of information passed between system components reduces. E.g., less data need to be written by an OS to a disk drive or transferred over the network. In tiered storage, deduplication can reduce the amount of data in the upper tiers (or in the cache) and also indirectly improve performance. Finally, for the devices with a limited number of write cycles (e.g., SSDs) deduplication can extend their lifetime as less data must be written to the device.

2.1 Deduplication Techniques

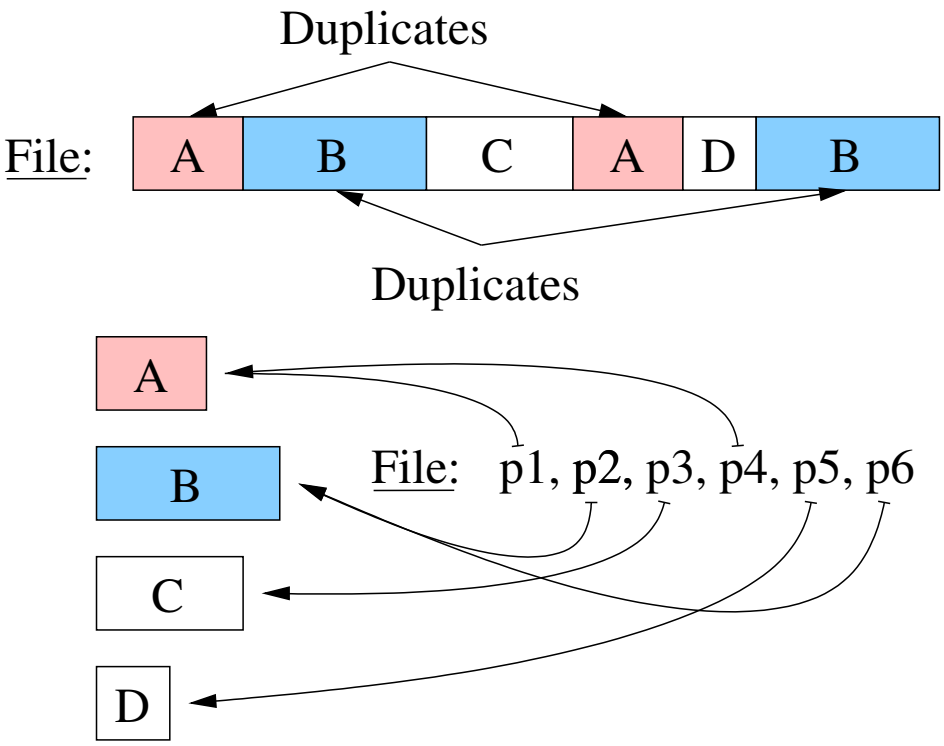


Figure 2.1: *Basic file system deduplication idea. This is how chunks can be distributed in an example file, how the actual blocks are stored, and how they can be recovered using techniques such as file recipes or other such mappings.*

The main idea of storage deduplication is to detect and then eliminate duplicates in the data. In most cases duplicates are defined as sequences of bytes with identical values, though there are rare exceptions [22]. The unit of deduplication is called *chunk* or *segment*; we will use the former one in this report. Typically, when several identical chunks are detected in the data, only a single instance is actually stored, although in some cases, more copies are kept to balance deduplication ratio with performance and reliability [57].

When users access deduplicated data, they still use older conventional interfaces, typically file system or block-based ones. Consequently deduplication is usually applied transparently, so that the original interface to access the data remains the same. For this reason, deduplication systems maintain metadata structures that allow reconstruction of the original data based on the requests coming from the original interface.

Figure 2.1 demonstrates an example of deduplication applied to a single file in a file system. There are two chunks with content A and two chunks with content B in a file. If a deduplication system detects them, it can store only one instance of every chunk and maintain (e.g., a *file recipe*) which is essentially a list of pointers to the single instances of the data. On a read request to specific offset of a file, the deduplication system can use the saved file recipe to identify where the deduplicated data is located.

Note that in some cases, notably *Content-Addressible Storage* (CAS), the interface of interactions between a user and a system changes because of the addition of deduplication [44,49]. In this case, maintaining some metadata, such as file recipes, is not needed, and information like the block's hash may be used instead to locate the block.

2.1.1 Workloads

Secondary storage. In terms of data deduplication, secondary storage refers to systems that primarily contain duplicate or secondary copies of data. For example, maintaining data backups is a form of secondary storage. Data stored in secondary storage systems is typically not very performance sensitive. Thus the overheads of deduplication are more tolerable when storing data in secondary storage systems. Also, in the case of backup workloads, large space savings are possible with deduplication since the backups are taken so often, and not many changes happen between backups. Thus the space savings obtained are significant.

Primary storage. Primary storage workloads are usually more performance sensitive, and the main data that one accesses is stored on these storage machines. Thus the overhead of deduplication may not be that beneficial. Also the space savings obtained in a primary storage workload may be comparatively lower than that in secondary storage systems. In spite of this, much research is being done on primary storage workloads. Even though the overhead of deduplication has a cost, if unnecessary I/O can be avoided, even performance improvement can be seen.

2.1.2 In-line vs. Post-processing

Deduplication can be done either in-line or as a post-process (offline) after the data is already written to the storage device.

Post-process. Incoming data is first stored to the storage device. Later a process will scan the data, identify duplicates, and reduce the space usage on the storage device. The advantage of this approach is that the throughput of the system is not affected thus allowing maximum throughput. More storage space will be required to store the full data before deduplication takes place. This can be a disadvantage if the storage device is near full capacity.

In-line. The chunking and hash calculation steps are done in real time before the data is written to the storage device. Duplicates are identified and only references to these duplicate blocks are updated. The duplicate data is not stored to disk. This requires less storage as the full data is never stored to the disk. On the downside, computational overhead is spend on hash calculations and looking up the deduplication metadata (which may require disk access based on how the metadata is stored). Also, the metadata must be looked up, which can potentially cause more overheads if I/O needs to be done. This can reduce the overall throughput of the system and affect performance. Many techniques like the use of Bloom filters and efficient metadata caching can be used to reduce the costs.

2.1.3 Layers

Due to the layered and modular structure of today's storage, different layers use their corresponding interfaces provided by the lower layers and export the interfaces that upper layers are used to. Often, when incorportating deduplication into the system, it is important to preserve the original interface provided by the system to the upper layers. In this case the whole deduplication process remains completely transparent to the upper layers. The original data to be deduplicated can be organized and accessed by the user in different ways. The level at which deduplication happens also determines which objects are available to the deduplication system. The levels discussed below are from the point of view of deduplication in top-down order.

Application. In this case applications implement deduplication for their objects themselves, and have available all the semantic information about the data. Sometimes, special building block tools, such as Venti [45], which perform deduplication, can be used to build storage applications like logical backups or snapshot file systems. The application understands the semantics of the data it is working on and has access to the application specifics of the objects. As a result, deduplication is performed on the objects that the application defines and can be done effectively without wasting resources on data that may not benefit from deduplication, or that can potentially harm reliability (e.g., in cases where having redundancy is required). For example, a mail server needs to store only one copy of an e-mail which was sent to multiple receipients in the same domain [19].

File system. In this case the deduplication system has access to information about the files. The data enters the system via POSIX read() and write() system calls. Note that deduplication can be done during read() or write() or even later (offline deduplication as a background process). In both cases, deduplication is performed at a file system level. At the file system level, the objects that the deduplication system operates on include: inodes, file boundaries, streams of files, etc. A special case is the network file system, where the client and server might be distributed.

There are many approaches to file system deduplication. It can be added to an existing file system, or implemented in a new file system. Modifying existing file systems to add new features may lead to instability in the file system which may take years to overcome. Also, modifying existing file systems can be difficult due to their complexity. Implementing a new file system in the kernel is time consuming and can take a long time to become stable. Stackable deduplication file systems can be created in the kernel [70] and is easier than trying to implement a new file system from scratch. User-space stackable file systems can be implemented as well using systems like FUSE [27, 53], but such file systems may have high performance overheads [47].

Block. In this case deduplication sees an abstract block to which reads and writes at corresponding offsets are performed. Again, deduplication can be done inline or offline. Unlike file system and application deduplication, deduplication at the block layer does not see file system structures or application objects. The block layer deduplication system sees blocks or a stream of blocks. Though only blocks are seen at this layer, one can still do whole-file chunking, if somehow file boundaries are determined. The main drawback of deduplication at this layer is the lack of semantic information to make informed decisions about what and when to deduplicate.

2.1.4 Chunking

The deduplication process in a general sense works on a sequence of bytes. The first step in this process is to determine boundaries within the sequence so that resulting chunks have a good chance to be deduplicated.

A byte sequence can be a stream from a backup software, a set of file system blocks, or other forms of data. Depending on the type of workload, whether deduplication is done inline or offline, and the layer in which deduplication is implemented, the format of the data may vary. For example, one factor that may determine the chunking method used can depend on the layer at which deduplication is implemented. The file system has information about where a file starts and ends in the stream. Many existing datasets contain many files that are identical. So, boundaries determined by the file are quited effective. Deduplication that uses file boundaries are called *whole-file chunking* [6].

Whole-File Chunking (WFC). When deduplication is performed at a file system level, one of the simplest implementation options is to consider each file as a separate chunk. The chunk size is naturally variable and differs significantly depending on a specific dataset. If there are two absolutely identical files in the system,

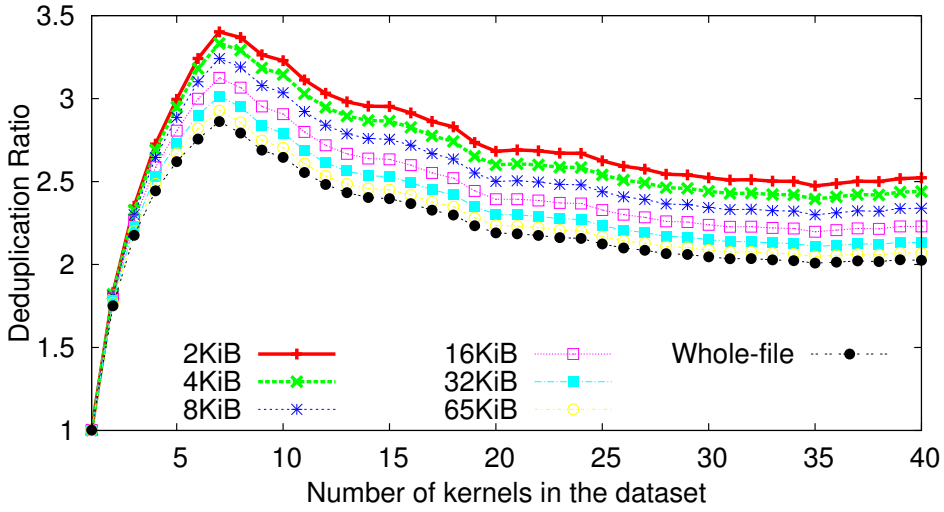


Figure 2.2: *Deduplication ratio provided by fixed chunking and whole file chunking for different chunk sizes for a Linux kernel dataset.*

they will be deduplicated. However, if even one byte differs, both of the files will be stored in their entirety. This can considerably reduce the deduplication ratio for certain datasets. Consider the following example. If you have 40 Linux kernels stored in unpacked form on a file system with whole-file deduplication, deduplication ratio will be about 2.02. WFC also has the advantage of having a smaller hash index than other chunking techniques since, especially for large files, the number of hashes that need to be stored is less than if fixed-length chunking or variable-length chunking are used.

The main drawback in whole-file chunking is that if a file is slightly modified, then identifying another file with mostly duplicate chunks will no longer be possible and more redundant data will have to be stored.

Fixed Chunking (FC). In fixed chunking, the chunk boundaries are broken based on a fixed length size. This approach is content-agnostic, and easy to implement, especially in the file system or in memory, where data is already divided into fixed length chunks (e.g. page size or file system block size). Chunk size plays a role in the deduplication ratio obtainable by this method. As can be seen in Figure 2.2, as the chunk size increases, the deduplication ratio decreases. This experiment was run with a 40 Linux kernels dataset. With smaller chunk sizes, the number of hashes that need to be stored in the hash index increases, which requires more metadata to be stored. Thus the choice of chunk size becomes a parameter that needs to be tweaked and there exists a tradeoff between chunk size and metadata overheads. The drawback of this approach is that if even a one byte change is made in a given chunk, all sequential chunks following that chunk will be different since effectively all of the following data has shifted by one byte. Thus even though the remaining data is duplicated, it will not be identified due to this shift.

Content Defined Chunking (CDC). Similar to whole-file chunking, Content Defined Chunking also produces chunks of variable size. The basic algorithm is depicted by Figure 2.3. The basic algorithm proposed in Figure 2.3 involves the notion of *anchors*. An anchor is a sequence of symbols that is encountered in a dataset with a stable periodicity. Consider an example in Figure 2.3, where the byte sequence 0x11 is used as anchor. The anchor can be used to define boundaries for each chunk. When the anchor is found, we can split the data at this boundary and create a chunk. It may happen that we come across the anchor too often in a certain part of the dataset, leading to very small chunks. Or we do not come across it enough, leading to very large chunks. To solve this problem, we can set a minimum and maximum chunk size. We then make

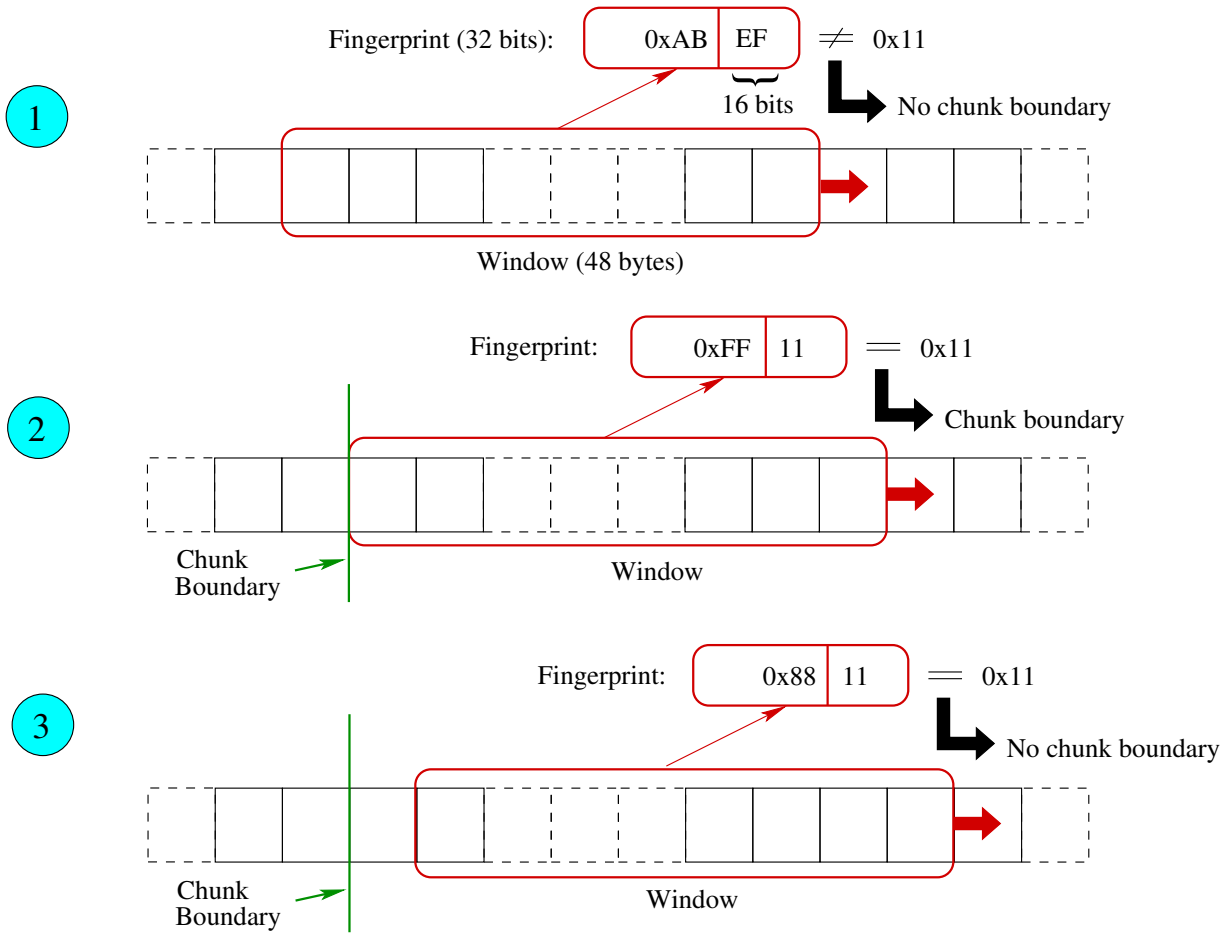


Figure 2.3: *Content-Defined Chunking (CDC) Basics. (1) shows the state of an example fingerprint, and last new byte added which does not match the anchor, thus including this byte in the current chunk. (2) shows what happen when the anchor is found, and the chunk boundary is created at this point. (3) shows what happens if the anchor is seen very soon after the last anchor was seen, and the chunk boundary is not created since minimum amount of data is not filled in this chunk.*

sure we have at least the minimum number of bytes in our chunk when deciding the boundary, irrespective of having seen the anchor or not, and we also make sure we have at most a maximum number of bytes in our chunk in case we have not seen the anchor. Often an average chunk size is chosen, and the variable length chunks are created such that this average chunk size is maintained. An example of how deduplication ratio varies based on the average chunk size for CDC is shown in Figure 2.4. It can be seen that as the average chunk size increases, the deduplication ratio decreases for the 40 Linux kernels dataset. Similar tradeoffs exist when choosing average chunk size and the total metadata required to store the deduplication information as with fixed chunking.

The advantage of this approach is that even if small changes in the data are made, only a few chunks will be affected by the change, whereas the remaining chunks will remain the same. This is because chunking is done on the basis of the actual content, and the anchors will remain in the same spots irrespective of the new changes. Thus this method is the most effective way to maximize the chances of finding duplicates.

One technique used for CDC is to use Rabin- fingerprinting [46]. This technique uses the idea of anchor discussed above, and a window that moves over a fixed-length region of the input. The rabin-fingerprint is a rolling hash, which means that to compute the rabin-fingerprint of a given window, some of the computation

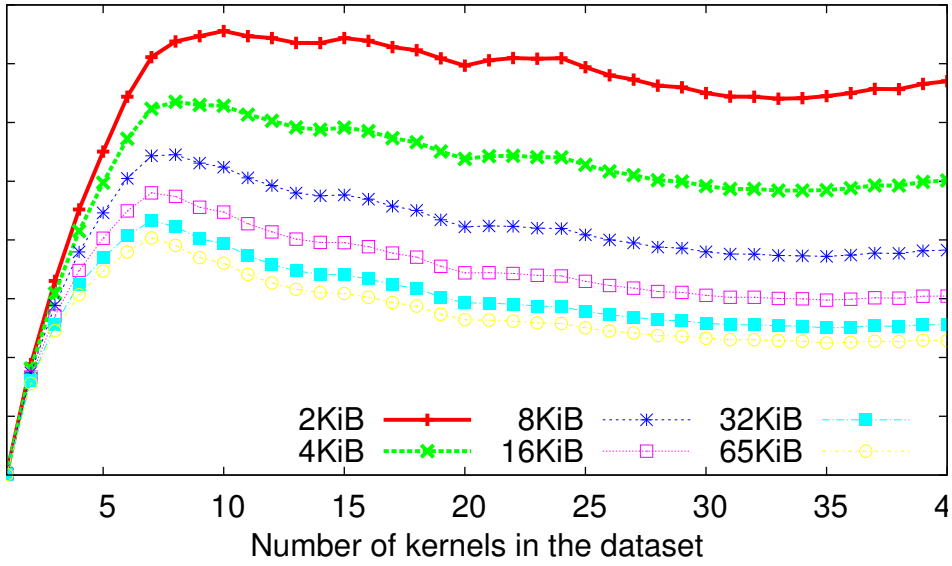


Figure 2.4: *Deduplication ratio provided by variable chunking for different chunk sizes for a Linux kernel dataset.*

of the previous overlapping window can be reused, thus reducing the overall complexity.

The main drawback of this approach is that there is more overhead in doing the actual chunking. For example, it has been shown that the overhead of performing Rabin fingerprint based CDC is about five times more than fixed chunking or whole-file chunking [43]. This is because, for fixed chunking and whole-file chunking, a majority of the computation is devoted to hash calculation (which is used to identify duplicate blocks), whereas in CDC the overhead of hash calculation is present along with the overhead of identifying boundaries for dividing the data into chunks.

In many cases, the way the data is split into chunks is the only factor that affects deduplication ratios. In fact, neither hashing nor indexing affects the amount of duplicates. The exception are the solutions where only part of the index is searched for the duplicates for the sake of improving the performance. Figure 2.5 shows the variance of deduplication ratio between variable chunking, fixed chunking, and whole-file chunking, where the chunk size for variable and fixed chunking is 4KB. The dataset used is the 40 Linux kernels.

2.1.5 Hashing

A popular method used to identify duplicate chunks is to use cryptographic hash functions, like SHA-1 or SHA-256. The entire chunk is hashed using these cryptographic hash functions. An index of all the hashes seen so far is maintained. When new chunks come in, their hashes are compared to the existing hashes stored in the hash index. If the same hash already exists in the hash index, then the chunk can be marked as a duplicate and the references can be updated instead of writing the chunk to the storage device. If the hash is not found, then it is added to the hash index, and the chunk is written to the storage device.

Strong cryptographic hash functions are used because they are collision resistant. Collision resistance is the property such that it is hard to find two different inputs such that their hash values are the same. Since every hash function will have more inputs than possible outputs (since the total number of outputs depends on the size of the hash), there will definitely be some collisions. A collision is basically the possiblility of

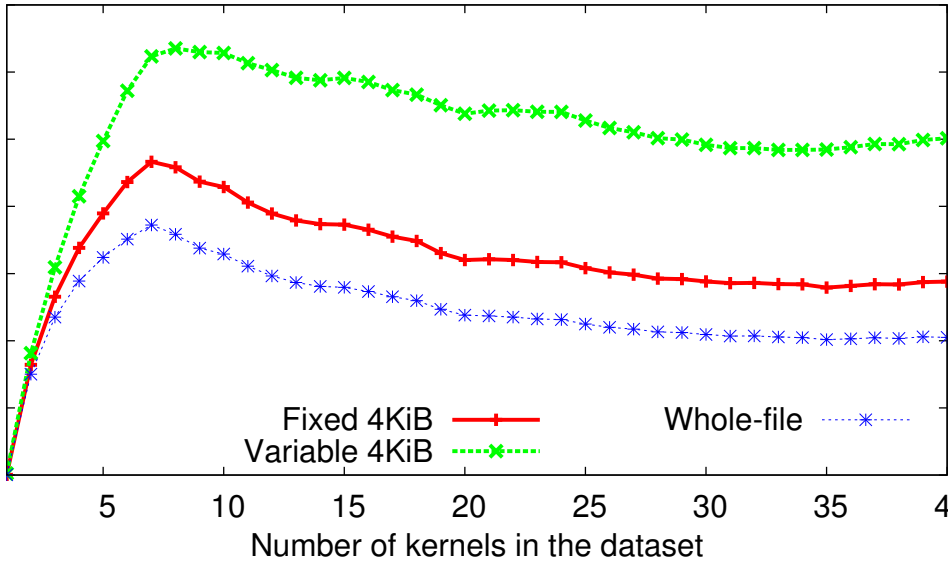


Figure 2.5: *Comparison of deduplication ratios for variable, fixed, and whole-file chunking methods.*

two different chunks of data having the same hash value. In such a case, if the reference is just updated for the new data chunk without checking the actual data, it will lead to a data corruption. To mitigate the affects of this, some approaches also use some form of byte-by-byte data comparison to make sure that the chunk is really a duplicate.

$$p \leq \frac{n(n-1))}{2} \times \frac{1}{2^b} \tag{2.1}$$

The probability of collision is bounded by the number of pairs of blocks possible, multiplied by the probability that a given pair will collide, as shown in Equation 2.1, where n is the number of data blocks and b is the number of bits in the hash function being used. Using a hash function like SHA-1 having 160 bits, over a petabyte of unique data (10^{15} bytes) stored as 1KB blocks (about 10^{12} blocks) will result in a collision rate of less than 3.4×10^{-25}. Standard algorithms like SHA-1 or SHA-256 provide a far lower probability of data loss than the risk of hardware errors (corruptions) like disk errors, which has been empirically shown to be in the range 10^{-18}–10^{-15} [17] Thus the use of crptographic hashes is popular in deduplication.

2.1.6 Overheads

Deduplication has its own costs and overheads. Computational costs are involved in chunking and hashing the data. Storage overheads are involved since we need to maintain metadata to track the hashes and track where the data is being stored. A tradeoff between chunk sizes and deduplication effectiveness needs to be made, since more metadata must be maintained for smaller chunk sizes. If the metadata storage needs exceed the actual space savings obtained from deduplication, then it may be better to avoid deduplication altogether. A decision about the hash function to be used must also be made for similar reasons. A hash index must be maintained to track all the hashes. Since the hash index can grow too large to keep in memory, various techniques can be used to cache parts of it while keeping the rest on disk. Efficient caching is necessary for the hash index since it is a critical data structure that must be accessed often, and I/O is slow and can become

	Ext2	Ext3	Ext4	Nilfs2
Unique metadata chunks (%)	98.9	84.7	84.6	99.9

Table 2.1: *Percentage of unique metadata in different file systems. All file systems contained over four million files.*

a major performance overhead. Some systems add a Bloom filter [4] to avoid the overhead of accessing this hash index if the hash has definitely not been seen before, but Bloom filters take up memory.

Since only references are kept for blocks that are duplicates, there needs to be some metadata to reconstruct a file or data stream based on the actual hashes that make it up. For this purpose file recipes or metadata mappings need to be maintained.

Another problem faced in deduplication is that since data is deduplicated on the write path, and references are just updated for duplicate data, the data reads can become randomized. This happens because the data might be spread across the disk if duplicates are found, even though logically they are supposed to be present sequentially. This can affect the read performance since sequential I/O is converted into random I/O.

2.2 Hints Background

Context Recovery The simplicity of the block-level interface has been both a blessing and a curse over the years. On one hand it is easy to create and interact with various block-level solutions. On the other hand, file system and application context cannot be easily propagated through such a limited interface.

Previous research has addressed the semantic gap between the block layer and a file system and has demonstrated that restoring all or part of the context can substantially improve block-level performance and reliability [3, 34, 54–56, 62]. We built on this observation by recovering partial file system and application context to improve block-level deduplication.

Context recovery can be achieved either by *introspection* or via *hinting*. Introspection relies on block-layer intelligence to infer file system or application operations. For example, one can identify metadata writes if the entire file-system layout is known. The benefit of introspection is that it does not require any file system changes; the disadvantage is that a successful implementation can be difficult [59, 61].

In contrast, *hinting* asks higher layers to provide small amounts of extra information to the deduplication system. Although file systems and perhaps applications must be changed, the necessary revisions are small compared to the benefits. Application changes can be minimized by interposing a library that can deduce hints from information such as the program name, file format, etc., so that only a few applications would need to be modified to notify the deduplication system of unusual needs.

In this work, we used hinting to recover context at the block layer. We present a list of possible hints in Section 2.2.1, including two useful ones that we support: NODEDUP and PREFETCH.

2.2.1 Potential Hints

Bypass deduplication Some writes are known *a priori* to be likely to be unique. For example, file system metadata such as inodes (which have varying timestamps and block pointers), directory entries, and indirect blocks rarely have duplicates. Table 2.1 shows the percentage of unique metadata blocks (4KB block size) for several file systems generated using Filebench's [15] file-server workload (4M files, mean directory width 20, mean file size 128KB using a gamma distribution, $\gamma = 1.5$). In all cases at least 84% of the metadata was unique. Ext3 and Ext4 have more metadata duplicates than Ext2 and Nilfs2 (14% vs. 1%), a phenomenon caused by journaling. Ext4 initially writes metadata blocks to the journal and then writes the same blocks to their proper location on the disk.

Attempting to deduplicate unique writes wastes CPU to compute hashes, and I/O bandwidth by searching for and inserting them in the index. Unique hashes also increase the index size, requiring more RAM cache and bandwidth for lookup, insertion, and garbage collection. Moreover, metadata writes are more critical to overall system performance than data writes because metadata writes are often synchronous.

Avoiding excessive metadata deduplication also improves reliability because uncorrupted metadata is critical to file systems. If several logical metadata blocks are deduplicated and the associated physical block is corrupted, several parts of the file system may become inconsistent. For that reason, many file systems store several copies of their metadata for reliability (e.g., Ext2/3/4 keeps multiple superblocks, ZFS explicitly duplicates metadata to avoid corruption); deduplicating those copies would obviate this feature. Likewise, file system journals are intended to enhance reliability, so deduplication of their blocks might be counterproductive. The journal is usually laid out sequentially so that it can be read sequentially on recovery. Deduplicating journal blocks may prevent data from being sequentially laid out, and thus hurt recovery speeds.

Applications might also generate data that should not or cannot be deduplicated. For example, some applications write random, compressed, or encrypted data; others write complex formats (e.g., virtual disk images) with internal metadata that tends to be unique [18]. HPC simulations generate many checkpoints that can potentially contain unique data based on the nature of the simulation. Sensor data may also contain unique data. Such applications may benefit from avoid deduplication. Another case is short-lived files, which should not be deduplicated because the cost paid does not justify the short-term benefits. In summary, if a block-level deduplication system can know when it is unwise to deduplicate a write, it can optimize its performance and reliability. We implemented a NODEDUP hint that informs our system that a corresponding request should not be deduplicated.

Prefetch hashes When a deduplication system knows what data is about to be written, it can prefetch the corresponding hashes from the index, accelerating future data writes by reducing lookup delays. For example, a copying process first reads source data and then writes it back. If the deduplication system can identify the behavior at read time, it can prefetch the corresponding hash entries from the index to speed up the write path. We implemented this hint and refer to it as PREFETCH. Another interesting use case for this hint is for segment cleaning in log-structured file systems such as Nilfs2, since it involves copying data from some segments to other segments to free up space.

Bypass compression Some deduplication systems compress chunks to save further space. However, if a file is already compressed (easily determined), additional compression consumes CPU time with no benefit.

Cluster hashes Files that reside in the same directory tend to be accessed together [24]. In a multi-user environment, a specific user's working set is normally far smaller than the whole file system tree [25]. Based on file ownership or on which directories contain files, a deduplication system could group hashes in the index and pre-load the cache more efficiently.

Partitioned hash index Partitioning the hash index based on incoming chunk properties is a popular technique for improving deduplication performance [1]. The chance of finding a duplicate in files of the same type is higher than across all files, so one could define partitions using, for example, file extensions.

Intelligent chunking Knowing file boundaries allows a deduplication system to efficiently chunk data. Certain large files (e.g., tarballs, VM images) contain many small files. Passing information about content boundaries to the block layer would enable higher deduplication ratios [29].

Chapter 3

Dmdedup: Design and Implementation

In this section, we classify Dmdedup's design, discuss the device-mapper framework, and finally present Dmdedup's architecture and its metadata backends.

3.1 Classification

Workloads. Storage deduplication has traditionally been applied to *secondary* workloads such as backup and archival storage [44,71]. Nowadays, however, deduplicating *primary* storage (e.g., user home directories and mail servers) has become a hot research topic [33, 38, 57, 64]. Dmdedup performs primary-storage deduplication, but due to its flexible design, it can also be tuned to store secondary data efficiently.

Levels. Deduplication can be implemented at the *application*, *file system*, or *block* level. Applications can use specialized knowledge to optimize deduplication, but modifying every application is impractical.

Deduplication in the *file system* benefits many applications. There are three approaches: (1) modifying an existing file system such as Ext3 [38] or WAFL [57]; (2) creating a stackable deduplicating file system either in-kernel [70] or using FUSE [27,53]; or (3) implementing a new deduplicating file system from scratch, such as EMC Data Domain's file system [71]. and Oracle's Solaris ZFS [7] which is now maintained by OpenZFS. Each approach has drawbacks. The necessary modifications to an existing file system are substantial and may harm stability and reliability. Developing in-kernel stackable file systems is difficult, and FUSE-based systems perform poorly [47]. A brand-new file system is attractive but typically requires massive effort and takes time to reach the stability that most applications need. Currently, this niche is primarily filled by proprietary products.

Implementing deduplication at the *block* level is easier because the block interface is simple. Unlike many file-system-specific solutions, block-level deduplication can be used beneath any block-based file system such as Ext4 [14], GPFS [52], BTRFS [42], GlusterFS, etc., allowing researchers to bypass a file system's limitations and design their own block-allocation policies. For that reason, we chose to implement Dmdedup at the block level. Our design means that Dmdedup can also be used with databases that require direct block-device access.

The drawbacks of block-level deduplication are threefold: (1) it must maintain an extra mapping (beyond the file system's map) between logical and physical blocks; (2) useful file-system and application context is lost; and (3) variable-length chunking is more difficult at the block layer. Dmdedup provides several options for maintaining logical-to-physical mappings. We also plan to recover some of the lost context using file system and application hints, which we discuss in Section 5.

Timeliness. Deduplication can be performed *in-line* with incoming requests or *off-line* via background scans. In-line deduplication saves bandwidth by avoiding repetitive reads and writes on the storage device and permits deduplication on a busy system that lacks idle periods. It also simplifies free-space accounting because all data is deduplicated immediately. In off-line deduplication systems, some portion of the data is temporary stored in a non-deduplicated state, and hence the exact amount of free space is not known until deduplication takes place. But it risks negatively impacting the performance of primary workloads due to the overhead of chunking, hashing, etc. Only a few studies have addressed this issue [57, 67]. Dmdedup performs inline deduplication; we discuss its performance in Section 4.

3.2 Device Mapper

The Linux Device Mapper (*DM*) framework, which has been part of mainline Linux since 2005, supports stackable block devices. To create a new device type, one builds a *DM target* and registers it with the OS. An administrator can then create corresponding *target instances*, which appear as regular block devices to the upper layers (file systems and applications). Targets rest above one or more physical devices (including RAM) or lower targets. Typical examples include software RAID, the Logical Volume Manager (LVM), and encrypting disks. We chose the DM framework for its performance and versatility: standard, familiar tools manage DM targets. Unlike user-space deduplication solutions [27, 33, 53] DM operates in the kernel, which improves performance yet does not prohibit communication with user-space daemons when appropriate [47]. The DM framework provides infrastructure that can be leveraged by virtual block devices. We use `persistent data` for metadata management in our metadata backend with transactional support. We use `dm-bufio` for performing buffered I/O for the metadata management in our disk-table metadata backend.

3.3 Dmdedup Components

Figure 3.1 depicts Dmdedup's main components and a typical setup. Dmdedup is a stackable block device that rests on top of physical devices (e.g., disk drives, RAIDs, SSDs), or stackable ones (e.g., encryption DM target). This approach provides high configurability, which is useful in both research and production settings.

Dmdedup typically requires two block devices to operate: one each for *data* and *metadata*. Data devices store actual user information; metadata devices track the deduplication metadata (e.g., a hash index). Dmdedup can be deployed even if a system has only one storage device, simply by creating two partitions. Although any combination of data and metadata devices can be used, we believe that using an HDD for data and an SSD for metadata is practical in a modern system, especially since access to metadata is highly critical and needs to be fast. Deduplication metadata sizes are much smaller than the data size—often less than 1% of the data—but metadata is critical enough to require low-latency access. This combination matches well with today's SSD size and performance characteristics, and ties into the growing trend of combining disk drives with a small amount of flash [41, 65, 66].

To upper layers, Dmdedup provides a conventional block interface: reads and writes with specific sizes and offsets. Every write to a Dmdedup instance is checked against all previous data. If a duplicate is detected, the corresponding metadata is updated and no data is written. Conversely, a write of new content is passed to the data device and tracked in the metadata. Since only one instance of a given block is stored, if it gets corrupted, multiple files may be affected. To avoid this problem, *dmdedup* can be run over RAID or all data blocks can be replicated to minimize loss of data.

Dmdedup main components are (Figure 3.1): (1) *deduplication logic* that chunks data, computes hashes, and coordinates other components; (2) a *hash index* that tracks the hashes and locations of the chunks; (3) a

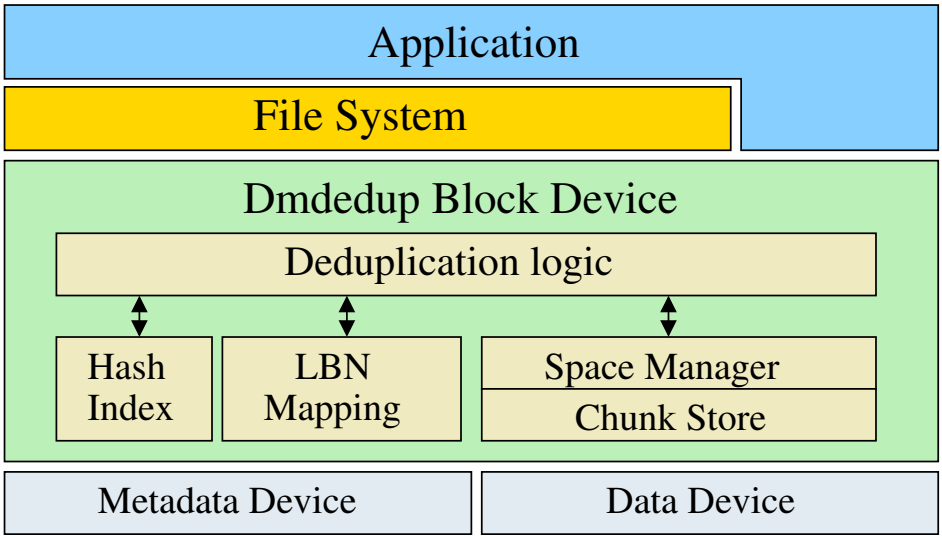


Figure 3.1: *Dmdedup high-level design.*

mapping between Logical Block Numbers (LBNs) visible to upper layers and the Physical Block Numbers (PBNs) where the data is stored; (4) a *space manager* that tracks space on the data device, maintains reference counts, allocates new blocks, and reclaims unreferenced data; and (5) a *chunk store* that saves user data to the data device.

3.4 Write Request Handling

Figure 3.3 shows how Dmdedup processes write requests.

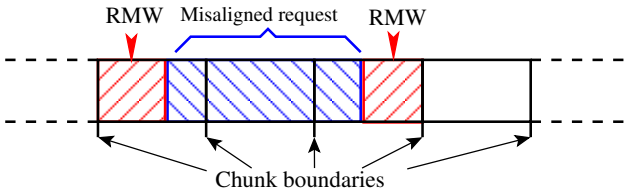


Figure 3.2: *Read-modify-write (RMW) effect for writes that are smaller than the chunk size or are misaligned.*

Chunking. The deduplication logic first splits all incoming requests into aligned, *subrequests* or chunks with a configurable power-of-two size. Smaller chunks allow Dmdedup to detect more duplicates but increase the amount of metadata [35], which can harm performance because of the higher metadata cache-miss rate. However, larger chunks can also hurt performance because they can require *read-modify-write* operations. As shown in Figure 3.2, to deduplicate a misaligned or small write request, Dmdedup must read the rest of the chunk from disk so that it can compute the complete (new) chunk's hash. Fortunately, most modern file systems use fixed and aligned block sizes of at least 4KB, so the RMW problem can easily be avoided. To achieve optimal performance, we recommend that Dmdedup's chunk size should match the block size of the file system above. In our evaluation we used 4KB chunking, which is common in many modern file systems.

Dmdedup does not currently support Content-Defined Chunking (CDC) [39] although the feature could be added in the future. We believe that CDC is less viable for inline primary-storage block-level deduplication because it produces a mismatch between request sizes and the underlying block device, forcing a read-modify-write operation for most write requests.

After chunking, Dmdedup passes subrequests to a pool of working threads. When all subrequests originating from an initial request have been processed, Dmdedup notifies the upper layer of I/O completion. Using several threads leverages multiple CPUs and allows I/O wait times to overlap with CPU processing (e.g., during some hash lookups). In addition, maximal SSD and HDD throughput can only be achieved with multiple requests in the hardware queue.

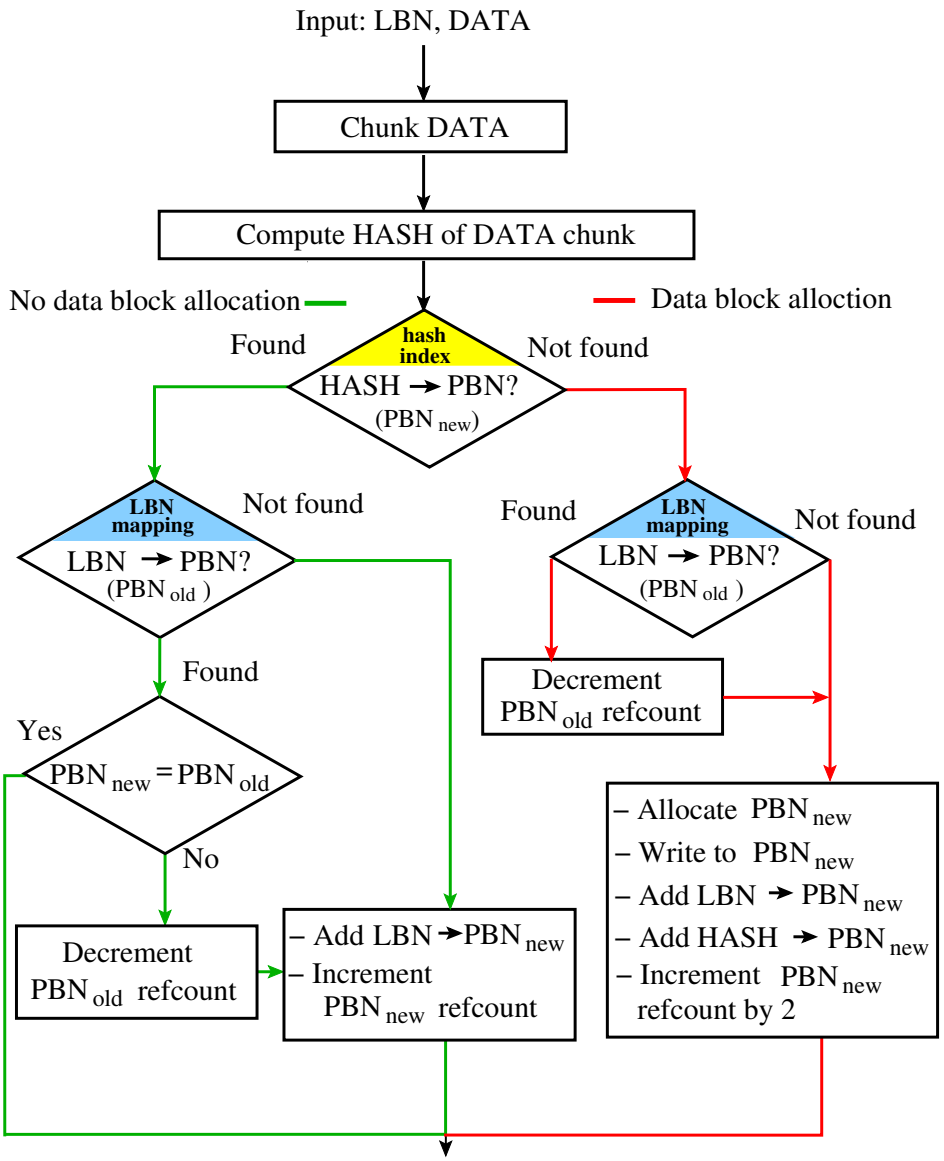


Figure 3.3: *Dmdedup write path. PBN_{new} is the PBN found in the hash index: a new physical location for incoming data. PBN_{old} is the PBN found in the LBN mapping: the old location of data, before the write ends.*

Hashing. For each subrequest, a worker thread first computes the hash. Dmdedup supports over 30 hash functions from the kernel's crypto library. Some are implemented using special hardware instructions (e.g., *SPARC64 crypt* and *Intel SSE3* extensions). As a rule, deduplication hash functions should be collision-resistant and cryptographically strong, to avoid inadvertent or malicious data overwrites. Hash sizes must be chosen carefully: a larger size improves collision resistance but increases metadata size. It is also important that the chance of a collision is significantly smaller than the probability of a disk error, which has been empirically shown to be in the range 10^{-18}–10^{-15} [17]. For 128-bit hashes, the probability of collision is less than 10^{-18} as long as the number of unique chunks is less than 2.6×10^{10}. For 4KB chunking, this corresponds to almost 100TB of unique data. Assuming a primary-storage deduplication ratio of $2\times$ [31], Dmdedup can support up to 200TB of logical space in such configuration. In our experiments we used 128-bit MD5 hashes.

Hash index and LBN mapping lookups. The main deduplication logic views both the hash index and the LBN mapping as abstract key-value stores. The hash index maps hashes to 64-bit PBNs; the LBN map uses the LBN as a key to look up a 64-bit PBN. We use PBN_{new} to denote the value found in the hash index and PBN_{old} for the value found in the LBN mapping.

Metadata updates. Several cases must be handled; these are represented by branches in Figure 3.3. First, the hash might be found in the index (left branch), implying that the data already exists on disk. There are two sub-cases, depending on whether the target LBN exists in the LBN→PBN mapping. If so, and if the corresponding PBNs are equal, the upper layer overwrote a location (the LBN) with data that was already there; this is surprisingly common in certain workloads [33]. If the LBN is known but mapped to a different PBN, then the data on the LBN must have changed; this is detected because the hash-to-PBN mapping is one-to-one, so PBN_{old} serves as a proxy for a different hash. Dmdedup decrements PBN_{old}'s reference count, adds the LBN→PBN_{new} mapping, and increments PBN_{new}'s reference count. On the other hand, if the hash→PBN mapping is found but the LBN→PBN one is not (still on the left side of the flowchart), we have a chunk of data that has been seen before (i.e., a duplicate) being written to a previously unknown LBN. In this case we add a LBN→PBN_{new} mapping and increment PBN_{new}'s reference count.

The flowchart's right side deals with the case where the hash is not found: a data chunk hasn't been seen before. If the LBN is also new (right branch of the right side), we proceed directly to allocation. If it is not new, we are overwriting an existing block with new data, so we must first dereference the data that used to exist on that LBN (PBN_{old}). In both cases, we now allocate and write a PBN to hold the new data, add hash→PBN and LBN→PBN mappings, and update the reference counts. We increment the counts by two in these cases because PBNs are referenced from both hash index *and* LBN mapping. For PBNs that are referenced only from the hash index (e.g., due to LBN overwrite) reference counts are equal to one. Dmdedup decrements reference counts to zero during garbage collection.

Garbage collection. During overwrites, Dmdedup does not reclaim blocks immediately, nor does it remove the corresponding hashes from the index. This approach decreases the latency of the critical write path. Also, if the same data is rewritten after a period of non-existence (i.e., it is not reachable through the LBN mapping), it can still be deduplicated; this is common in certain workloads [40]. However, data-device space must eventually be reclaimed. We implemented an offline garbage collector that periodically iterates over all data blocks and recycles those that are not referenced.

If a file is removed by an upper-layer file system, the corresponding blocks are no longer useful. However, Dmdedup operates at the block layer and thus is unaware of these inaccessible blocks. Some modern file systems (e.g., Ext4 and NTFS) use the SATA TRIM command to inform SSDs that specific LBNs are not referenced anymore. Dmdedup takes advantage of these TRIMs to reclaim unused file system blocks.

3.5 Read Request Handling

In Dmdedup, reads are much simpler to service than writes. Every incoming read is split into chunks and queued for processing by worker threads. The LBN→PBN map gives the chunk's physical location on the data device. The chunks for the LBNs that are not in the LBN mapping are filled with zeroes; these correspond to reads from an offset that was never written. When all of a request's chunks have been processed by the workers, Dmdedup reports I/O completion.

3.6 Metadata Backends

Backend API. We designed a flexible API that abstracts metadata management away from the main deduplication logic. Having pluggable metadata backends facilitates easier exploration and comparison of different metadata management policies. When constructing a Dmdedup target instance, the user specifies which backend should be used for this specific instance and passes the appropriate configuration parameters. Our API includes ten mandatory and two optional methods—including basic functions for initialization and destruction, block allocation, lookup, insert, delete, and reference-count manipulation. The optional methods support garbage collection and synchronous flushing of the metadata.

An unusual aspect of our API is its two types of key-value stores: *linear* and *sparse*. Dmdedup uses a linear store (from zero to the size of the Dmdedup device) for LBN mapping and a sparse one for the hash index. Backend developers should follow the same pattern, using the sparse store for key spaces where the keys are uniformly distributed. In both cases the interface presented to the upper layer after the store has been created is uniform: **kvs_insert**, **kvs_lookup**, and **kvs_delete**.

To create a new metadata backend, one needs to implement the following twelve API methods (ten mandatory, two optional):

1, 2. **init_meta**, **exit_meta**: Called when a Dmdedup target instance is created or destroyed, respectively. The backend initializes or reconstructs metadata, or saves it to the metadata device.

3. **kvs_create_linear**: Creates a linear key-value store for LBN mapping. The key space is linear, from zero to the total size of the Dmdedup device.

4. **kvs_create_sparse**: Creates a sparse key-value store for the hash index. The backend developer should optimize this store for a sparse key space because chunk hashes are distributed uniformly.

5, 6, 7. **kvs_insert**, **kvs_lookup**, **kvs_delete**: Called to insert, look up, or delete key-value pairs, respectively. These methods are used, for example, to lookup hashes in the index or insert new LBN mapping entries. Linear and sparse key-value stores may implement these functions differently.

8. **kvs_iterate** (Optional): Called during garbage collection when Dmdedup needs to iterate over all hashes in the index. If not implemented, garbage collection is disabled, which corresponds to a storage system with a *write-once* policy where the deletion of data is prohibited [44].

9. **alloc_data_block**: Called to allocate a new block on a data device. This usually happens when new (unique) data is written.

10. **inc_refcount**: Called to increment the reference count of a specific block. Every time a duplicate is found, the reference count of the corresponding physical block is increased.

11. **dec_refcount**: Called during LBN overwrite or garbage collection to decrement a physical block's reference count. When the reference count reaches zero, the backend should consider the block unused and can return it in future **alloc_data_block** calls.

12. **flush_meta** (Optional): Asks the backend to write its metadata synchronously to persistent storage. To provide the consistency guarantees expected by the upper layers, Dmdedup flushes its metadata when a file system issues a synchronous write, a barrier, or a drop-cache request. If the method is not implemented then flushing is skipped.

When designing the metadata backend API, we tried to balance flexibility with simplicity. Having more functions would burden the backend developers, while fewer functions would assume too much about metadata management and limit Dmdedup's flexibility. In our experience, the API we designed strikes the right balance between complexity and flexibility.

We consolidated key-value stores, reference counting, and block allocation facilities within a single metadata backend object because they often need to be managed together and are difficult to decouple. In particular, when metadata is flushed, all of the metadata (reference counters, space maps, key-value stores) needs to be written to the disk at once. For backends that support transactions this means that proper ordering and atomicity of all metadata writes are required.

Dmdedup performs 2–8 metadata operations for each write. But depending on the metadata backend and the workload properties, every metadata operation can generate zero to several I/O requests to the metadata device.

Using the above API, we designed and implemented three backends: INRAM (in RAM only), DTB (disk table), and CBT (copy-on-write B-tree). These backends have significantly different designs and features; we detail each backend below.

3.6.1 INRAM Backend

INRAM is the simplest backend we implemented. It stores all deduplication metadata in RAM and consequently does not perform any metadata I/O. All *data*, however, is still stored on the data device as soon as the user's request arrives (assuming it is not a duplicate). INRAM metadata can be written persistently to a user-specified file at any time (e.g., before shutting the machine down) and then restored later. This facilitates experiments that should start with a pre-defined metadata state (e.g., for evaluating the impact of LBN space fragmentation). The INRAM backend allows us to determine the baseline of maximum deduplication performance on a system with a given amount of CPU power. It can also be used to quickly identify a workload's deduplication ratio and other characteristics. With the advent of DRAM backed by capacitors or batteries, this backend can become a viable option for production.

INRAM uses a statically allocated hash table for the sparse key-value store, and an array for the linear store. The linear mapping array size is based on the Dmdedup target instance size. The hash table for the sparse store is allocated (and slightly over-provisioned) based on the size of the data device, which dictates the maximum possible number of unique blocks. We resolve collisions with linear probing; according to standard analysis the default over-provisioning ratio of 10% lets Dmdedup complete a successful search in an average of six probes when the data device is full.

We use an integer array to maintain reference counters and allocate new blocks sequentially using this array.

3.6.2 DTB Backend

The disk table backend (DTB) uses INRAM-like data structures but keeps them on persistent storage. If no buffering is used for the metadata device, then every lookup, insert, and delete operation causes one extra I/O, which significantly harms deduplication performance. Instead, we use Linux's *dm-bufio* subsystem, which buffers both reads and writes in 4KB units and has a user-configurable cache size. By default, dm-bufio flushes all dirty buffers when more than 75% of the buffers are dirty. If there is no more space in the cache for new requests, the oldest blocks are evicted one by one. Dm-bufio also normally runs a background thread

that evicts all buffers older than 60 seconds. We disabled this thread for deduplication workloads because we prefer to keep hashes and LBN mapping entries in the cache as long as there is space. The dm-bufio code is simple (1,100 LOC) and can be easily modified to experiment with other caching policies and parameters.

The downside of DTB is that it does not scale with the size of deduplication metadata. Even when only a few hashes are in the index, the entire on-disk table is accessed uniformly during hash lookup and insertion. As a result, hash index blocks cannot be buffered efficiently even for small datasets.

3.6.3 CBT Backend

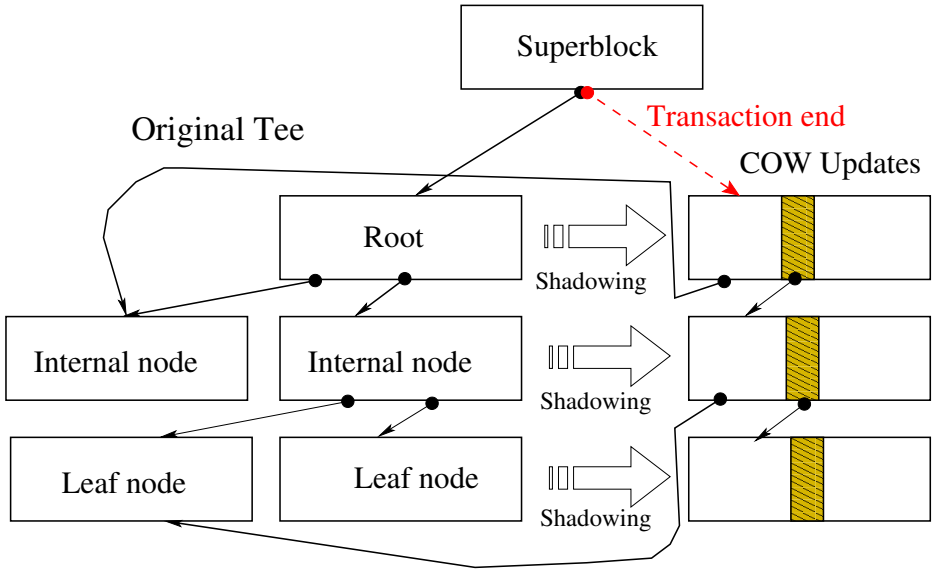


Figure 3.4: *Copy-on-Write (COW) B-trees.*

Unlike the INRAM and DTB backends, CBT provides true transactionality. It uses Linux's on-disk Copy-On-Write (COW) B-tree implementation [11, 63] to organize its key-value stores (see Figure 3.4). All keys and values are stored in a B^+-tree (i.e., values are located only in leaves). When a new key is added, the corresponding leaf must be updated. However, the COW B-tree does not do so in-place; instead, it *shadows* the block to a different location and applies the changes there. Next, the internal nodes of the B-tree are updated to point to the new leaf, again using shadowing. This procedure continues up to the root, which is referenced from a pre-defined location on the metadata device—the *Dmdedup superblock*. Multiple changes are applied to the B-tree in COW fashion but the superblock is not updated until Dmdedup explicitly ends the transaction. At that point, the superblock is atomically updated to reference the new root. As a result, if a system fails in the middle of a transaction, the user sees old data but not a corrupted device state. The CBT backend also allocates data blocks so that data overwrites do not occur within the transaction; this ensures both data and metadata consistency.

Every key lookup, insertion, or deletion requires $\log_b N$ I/Os to the metadata device (where b is the branching factor and N is the number of keys). The base of the logarithm is large because many key-pointer entries fit in a single 4KB non-leaf node (approximately 126 for the hash index and 252 for the LBN mapping). To improve CBT's performance we use dm-bufio to buffer I/Os. When a transaction ends, dm-bufio's cache is flushed. Users can control the length of a transaction in terms of the number of writes.

Because CBT needs to maintain intermediate B-tree nodes, its metadata is larger than DTB's. Moreover, the COW update method causes two copies of the updated blocks to reside in the cache simultaneously. Thus,

for a given cache size, CBT usually performs more I/O than DTB. But CBT scales well with the amount of metadata. For example, when only a few hashes are in the index, they all reside in one block and can be easily cached.

Statistics. Dmdedup collects statistics on the number of reads and writes, unique and duplicated writes, overwrites, storage and I/O deduplication ratios, and hash index and LBN mapping sizes, among others. These statistics were indispensable in analyzing our own experiments (Section 4).

Device size Most file systems are unable to dynamically grow or shrink in response to changing device size. Thus, users must currently specify the Dmdedup device's size at construction time. However, the device's logical size depends on the actual deduplication ratio, which changes dynamically. Some studies offer techniques to predict dataset deduplication ratios [69], and we provide a set of tools to compute deduplication ratios in an existing dataset. If the deduplication ratio ever falls below expectations and the free data space nears to zero, Dmdedup warns the user via the OS's console or system log. The user can then remove unnecessary files from the device and force an immediate garbage collection, or add more data devices to the pool.

3.7 Implementation

The latest Dmdedup code has been tested against Linux 3.14 but we performed experimental evaluation on version 3.10.9. We have also updated the Dmdedup code to Linux 3.19 kernel. We kept the code base small to facilitate acceptance to the mainline and to allow users to investigate new ideas easily. Dmdedup's core has only 1,400 lines of code; the INRAM, DTB, and CBT backends have 500, 1,400, and 600 LOC, respectively. Over the course of two years, fifteen developers of varying skill levels have worked on the project. Dmdedup is open-source and was submitted for initial review to `dm-devel@` mailing list. The code is also available at `git://git.fsl.cs.sunysb.edu/linux-dmdedup.git`.

When constructing a Dmdedup instance, the user specifies the data and metadata devices, metadata backend type, cache size, hashing algorithm, etc. In the future, we plan to select or calculate reasonable defaults for most of these parameters. Dmdedup exports deduplication statistics and other information via the device mapper's `STATUS ioctl`, and includes a tool to display these statistics in a human-readable format.

Chapter 4

Dmdedup: Evaluation

In this section, we first evaluate Dmdedup's performance and overheads using different backends and under a variety of workloads. Then we compare Dmdedup's performance to Lessfs [27]—an alternative, popular deduplication system implemented using FUSE.

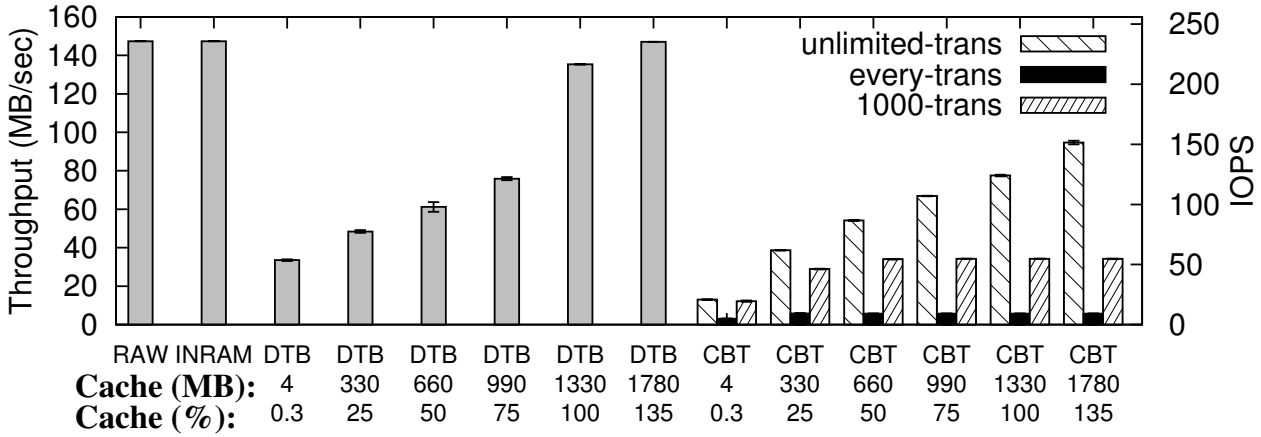
4.1 Experimental Setup

In our experiments we used three identical Dell PowerEdge R710 machines, each equipped with an Intel Xeon E5540 2.4GHz 4-core CPU and 24GB of RAM. Using lmbench we verified that the performance of all machines was within 4% of each other. We used an Intel X25-M 160GB SSD as the metadata device and a Seagate Savvio 15K.2 146GB disk drive for the data. Although the SSD's size is 160GB, in all our experiments we used 1.5GB or less for metadata. Both drives were connected to their hosts using Dell's PERC 6/i Integrated controller. On all machines, we used CentOS Linux 6.4 x86_64, upgraded to Linux 3.10.9 kernel. Unless otherwise noted, every experiment lasted from 10 minutes (all-duplicates data write) to 9 hours (Mail trace replay). We ran all experiments at least three times. Using Student's t distribution, we computed 95% confidence intervals and report them on all bar graphs; all half-widths were less than 5% of the mean.

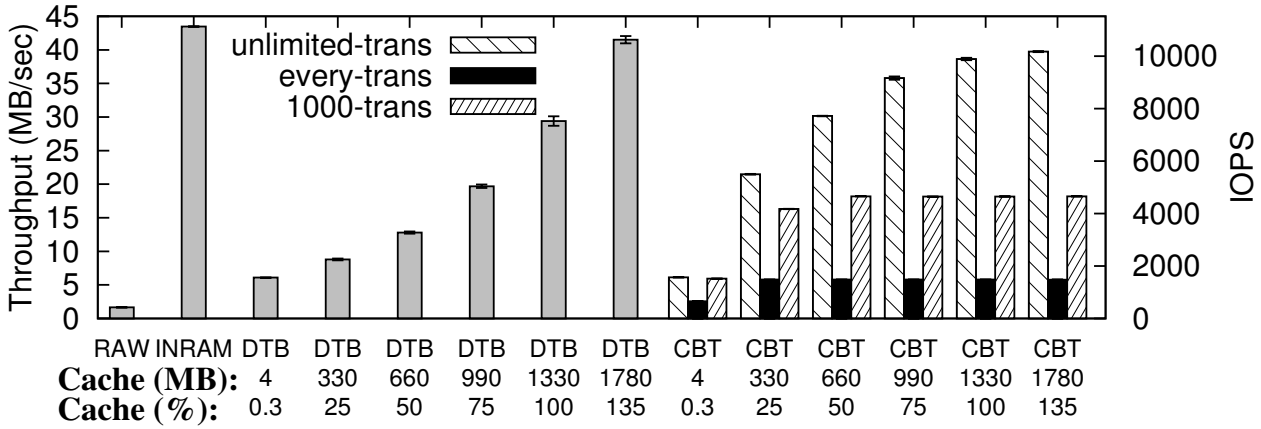
We evaluated four setups: the raw device, and Dmdedup with three backends—INRAM, disk table (DTB), and COW B-Tree (CBT). In all cases Dmdedup's logical size was set to the size of the data device, 146GB, which allowed us to conduct experiments with unique data without running out of physical space. 146GB corresponds to 1330MB of metadata: 880MB of hash→PBN entries (24B each), 300MB of LBN→PBN entries (8B each), and 150MB of reference counters (4B each). We used six different metadata cache sizes: 4MB (0.3% of all metadata), 330MB (25%), 660MB (50%), 990MB (75%), 1330MB (100%), and 1780MB (135%). A 4MB cache corresponds to a case when no significant RAM is available for deduplication metadata; the cache acts as a small write buffer to avoid doing I/O with every metadata update. As described in Section 3.6.2, by default Dmdedup flushes dirty blocks after their number exceeds 75% of the total cache size; we did not change this parameter in our experiments. When the cache size is set to 1780MB (135% of all metadata) the 75% threshold equals to 1335MB, which is greater than the total size of all metadata (1330MB). Thus, even if all metadata is dirty, the 75% threshold cannot be reached and flushing never happens.

4.2 Experiments

To users, Dmdedup appears as a regular block device. Using its basic performance characteristics—sequential and random read/write behavior—one can estimate the performance of a complete system built using Dmd-



(a) Sequential write, 640KB I/O size



(b) Random write, 4KB I/O size

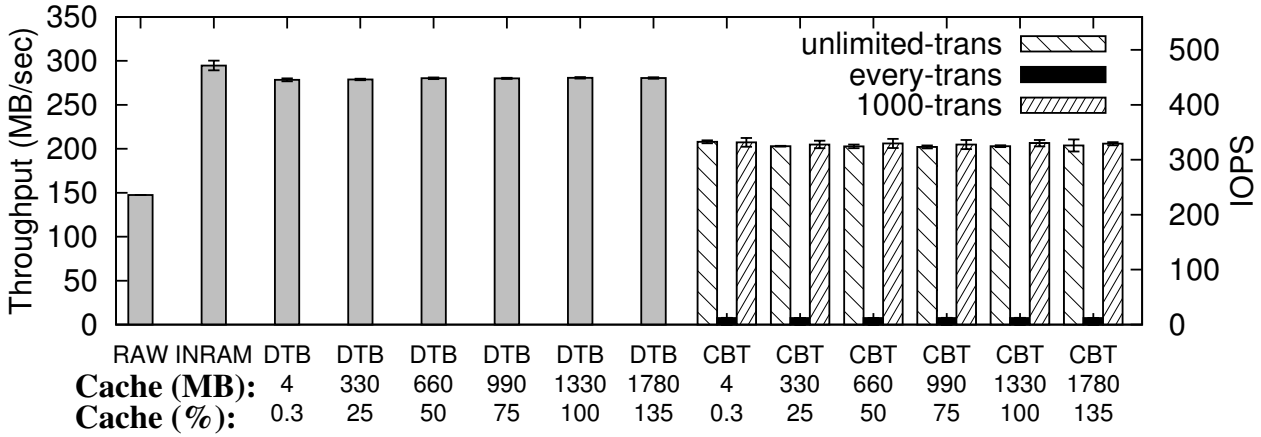
Figure 4.1: *Sequential and random write throughput for the **Unique** dataset. Results are for the raw device and for Dmdedup with different metadata backends and cache sizes. For the CBT backend we varied the transaction size: unlimited, every I/O, and 1,000 writes.*

edup. Thus, we first evaluated Dmdedup's behavior with micro-workloads. To evaluate its performance in real deployments, we then replayed three production traces.

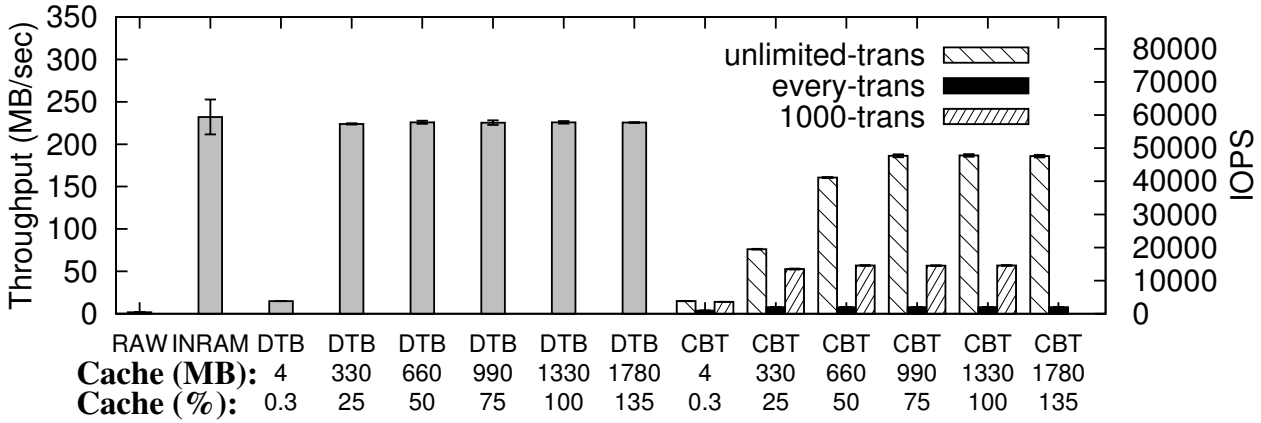
4.2.0.1 Micro-workloads

Unlike traditional (non-deduplicating) storage, a deduplication system's performance is sensitive to data content. For deduplication systems, a key content characteristic is its deduplication ratio, defined as the number of logical blocks divided by the number of blocks physically stored by the system [12]. Figures 4.1, 4.2, and 4.3 depict write throughput for our *Unique*, *All-duplicates*, and *Linux-kernels* datasets, respectively. Each dataset represents a different point along the content-redundancy continuum. *Unique* contains random data obtained from Linux's */dev/urandom* device. *All-duplicates* consists of a random 4KB block repeated for 146GB. Finally, *Linux-kernels* contains the sources of 40 Linux kernels (more details on the format follow).

We experimented with two types of micro-workloads: large sequential writes (subfigures (a) in Figures 4.1–4.3) and small random writes (subfigures (b) in Figures 4.1–4.3). I/O sizes were 640KB and 4KB for sequential and random writes, respectively. We chose these combinations of I/O sizes and access patterns because real applications tend either to use a large I/O size for sequential writes, or to write small objects



(a) Sequential write, 640KB I/O size



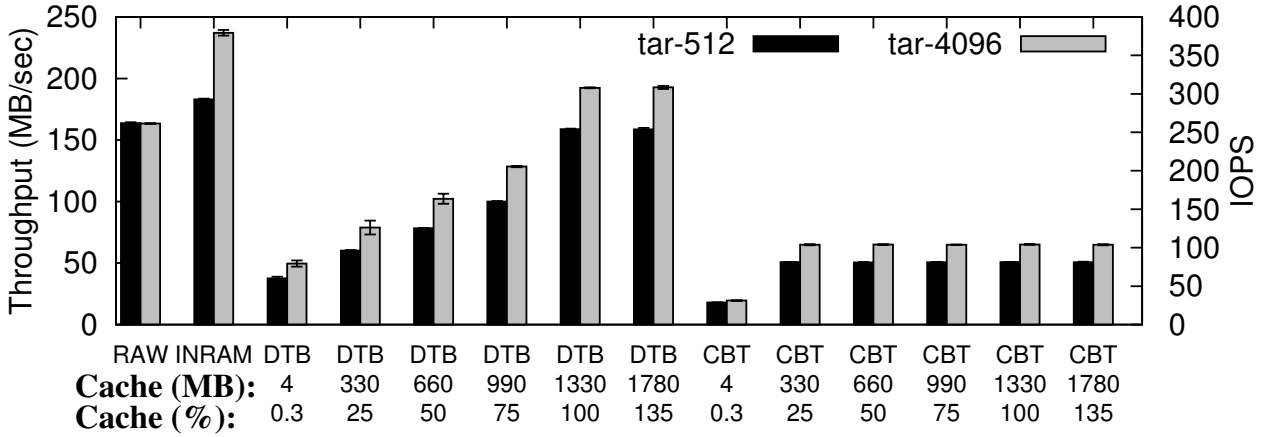
(b) Random write, 4KB I/O size

Figure 4.2: *Sequential and random write throughput for the **All-duplicates** dataset. Results are for the raw device and for Dmdedup with different backends and cache sizes. For the CBT backend, we varied the transaction size: unlimited, every I/O, and 1,000 writes.*

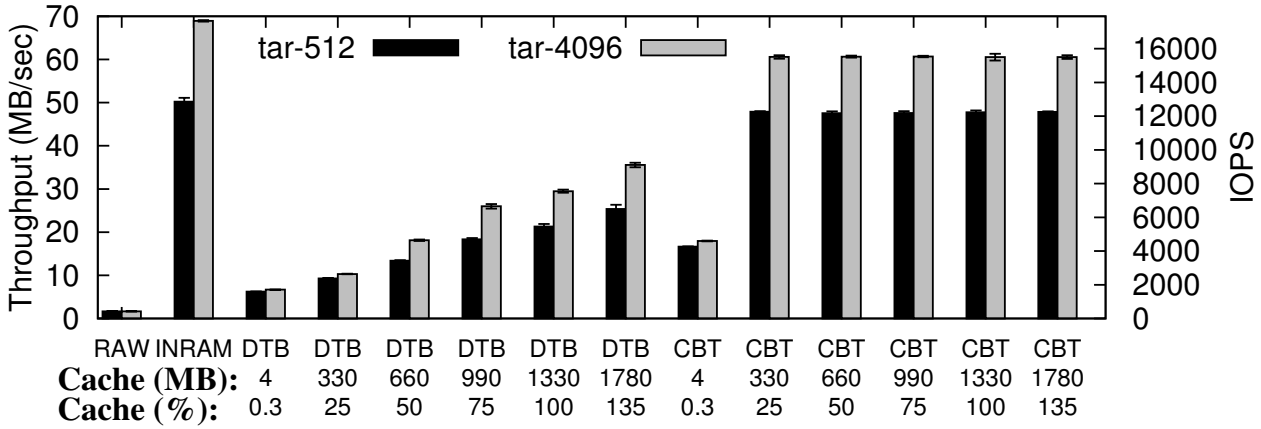
randomly (e.g., databases) [9]. 4KB is the minimal block size for modern file systems. Dell's PERC 6/i controller does not support I/O sizes larger than 320KB, so large requests are split by the driver; the 640KB size lets us account for the impact of splitting. We started all experiments with an empty Dmdedup device and ran them until the device became full (for the Unique and All-kernels datasets) or until the dataset was exhausted (for Linux-kernels).

Unique (Figure 4.1) Unique data produces the lowest deduplication ratio (1.0, excluding metadata space). In this case, the system performs at its worst because the deduplication steps need to be executed as usual (e.g., index insert), yet all data is still written to the data device. For the sequential workload (Figure 4.1(a)), the INRAM backend performs as well as the raw device—147MB/sec, matching the disk's specification. This demonstrates that the CPU and RAM in our system are fast enough to do deduplication without any visible performance impact. However, it is important to note that CPU utilization was as high as 65% in this experiment.

For the DTB backend, the 4MB cache produces 34MB/sec throughput—a 75% decrease compared to the raw device. Metadata updates are clearly a bottleneck here. Larger caches improve DTB's performance



(a) Sequential write, 640KB I/O size)



(b) Random write, 4KB I/O size

Figure 4.3: *Sequential and random write throughput for the **Linux-kernels** dataset. Results are for the raw device and for Dmdedup with different metadata backends and cache sizes. For the CBT backend, the transaction size was set to 1,000 writes.*

roughly linearly up to 147MB/sec. Interestingly, the difference between DTB-75% and DTB-100% is significantly more than between other sizes because Dmdedup with 100% cache does not need to perform any metadata reads (though metadata writes are still required). With a 135% cache size, even metadata writes are avoided, and hence DTB achieves INRAM's performance.

The CBT backend behaves similarly to DTB but its performance is always lower—between 3–95MB/sec depending on the cache and transaction size. The reduced performance is caused by an increased number of I/O operations, 40% more than DTB. The transaction size significantly impacts CBT's performance; an unlimited transaction size performs best because metadata flushes occur only when 75% of the cache is dirty. CBT with a transaction flush after every write has the worst performance (3–6MB/sec) because every write to Dmdedup causes an average of 14 writes to the metadata device: 4 to update the hash index, 3 to update the LBN mapping, 5 to allocate new blocks on the data and metadata devices, and 1 to commit the superblock. If a transaction is committed only after 1,000 writes, Dmdedup's throughput is 13–34MB/sec—between that of unlimited and single-write transactions. Note that in this case, for all cache sizes over 25%, performance does not depend on the cache size but only on the fact that Dmdedup flushes metadata after every 1,000 writes.

Trace	Duration (days)	Written (GB)	Written unique by content (GB)	Written unique by offset (GB)	Read (GB)	Dedup ratio	Dedup block size (B)	Ranges accessed (GB)
Web	21	42.64	22.45	0.95	11.89	2.33	4,096	19.07
Mail	20	1,483.41	110.39	8.28	188.94	10.29	4,096	278.87
Homes	21	65.27	16.83	4.95	15.46	3.43	512	542.32

Table 4.1: *Summary of FIU trace characteristics*

For random-write workloads (Figure 4.1(b)) the raw device achieves 420 IOPS. Dmdedup performs significantly better than the raw device—between 670 and 11,100 IOPS (in 4KB units)—because it makes random writes sequential. Sequential allocation of new blocks is a common strategy in deduplication systems [33, 64, 71], an aspect that is often overlooked when discussing deduplication's performance impact. We believe that write sequentialization makes primary storage deduplication significantly more practical than commonly perceived.

All-duplicates (Figure 4.2) This dataset is on the other end of the deduplication ratio range: all writes contain exactly the same data. Thus, after the first write, nothing needs to be written to the data device. As a result, Dmdedup outperforms the raw device by 1.4–2× for the sequential workload in all configurations except CBT with per-write transactions (Figure 4.2(a)). In the latter case, Dmdedup's throughput falls to 12MB/sec due to the many (ten) metadata writes induced by each user write. Write amplification is lower for All-duplicates than for Unique because the hash index is not updated in the former case. The throughput does not depend on the cache size here because the hash index contains only one entry and fits even in the 4MB cache. The LBN mapping is accessed sequentially, so a single I/O brings in many cache entries that are immediately reaccessed.

For random workloads (Figure 4.2(b)), Dmdedup improves performance even further: 2–140× compared to the raw device. In fact, random writes to a disk drive are so slow that deduplicating them boosts overall performance. But unlike the sequential case, the DTB backend with a 4MB cache performs poorly for random writes because LBNs are accessed randomly and 4MB is not enough to hold the entire LBN mapping. For all other cache sizes, the LBN mapping fits in RAM and performance was thus not impacted by the cache size.

The CBT backend caches two copies of the tree in RAM: original and modified. This is the reason why its performance depends on the cache size in the graph.

Linux kernels (Figure 4.3) This dataset contains the source code of 40 Linux kernels from version 2.6.0 to 2.6.39, archived in a single tarball. We first used an unmodified `tar`, which aligns files on 512B boundaries (*tar-512*). In this case, the tarball size was 11GB and the deduplication ratio was 1.18. We then modified `tar` to align files on 4KB boundaries (*tar-4096*). In this case, the tarball size was 16GB and the deduplication ratio was 1.88. Dmdedup uses 4KB chunking, which is why aligning files on 4KB boundaries increases the deduplication ratio. One can see that although tar-4096 produces a larger logical tarball, its physical size ($16\text{GB}/1.88 = 8.5\text{GB}$) is actually smaller than the tarball produced by tar-512 ($11\text{GB}/1.18 = 9.3\text{GB}$).

For sequential writes (Figure 4.3(a)), the INRAM backend outperforms the raw device by 11% and 45% for tar-512 and tar-4096, respectively. This demonstrates that storing data in a deduplication-friendly format (tar-4096) benefits performance in addition to reducing storage requirements. This observation remains true for other backends. (For CBT we show results for the 1000-write transaction configuration.) Note that for random writes (Figure 4.3(b)), the CBT backend outperforms DTB. Unlike a hash table, the B-tree scales well with the size of the dataset. As a result, for both 11GB and 16GB tarballs, B-trees fit in RAM, while the on-disk hash table is accessed randomly and cannot be cached as efficiently.

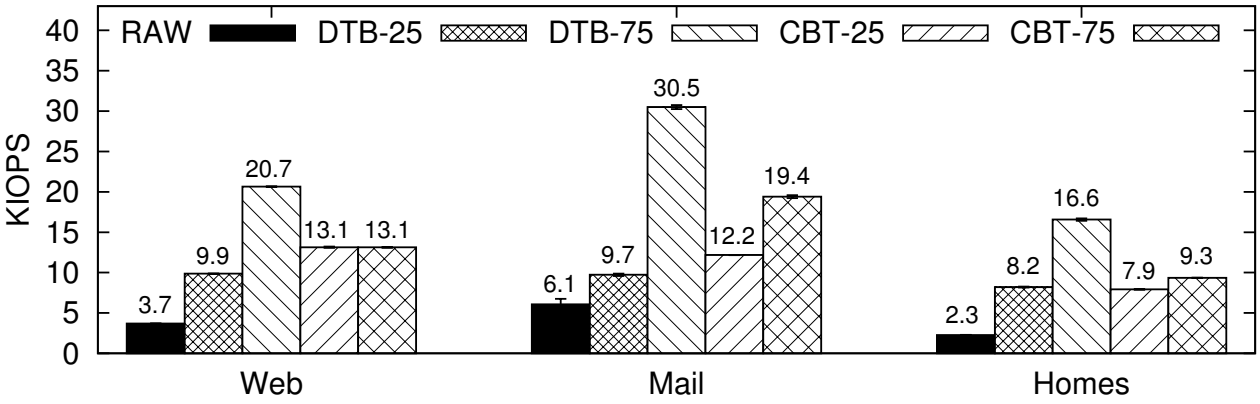


Figure 4.4: *Raw device and Dmdedup throughput for FIU's Web, Mail, and Homes traces. The CBT backend was setup with 1000-writes transaction sizes.*

4.2.0.2 Trace Replay

To evaluate Dmdedup's performance under realistic workloads, we used three production traces from Florida International University (FIU): Web, Mail, and Homes [2, 23]. Each trace was collected in a significantly different environment. The Web trace originates from two departments' Web servers; Homes from a file server that stores the home directories of a small research group; and Mail from a departmental mail server. Table 4.1 presents relevant characteristics of these traces.

FIU's traces contain data hashes for every request. We applied a patch from Koller and Rangaswami [23] to Linux's *btreplay* utility so that it generates unique write content corresponding to the hashes in the traces, and used that version to drive our experiments. Some of the reads in the traces access LBNs that were not written during the tracing period. When serving such reads, Dmdedup would normally generate zeroes without performing any I/O; to ensure fairness of comparison to the raw device we pre-populated those LBNs with appropriate data so that I/Os happen for every read.

The Mail and Homes traces access offsets larger than our data device's size (146GB). Because of the indirection inherent to deduplication, Dmdedup can support a logical size larger than the physical device, as long as the deduplication ratio is high enough. But replaying the trace against a raw device (for comparison) is not possible. Therefore, we created a small device-mapper target that maintains an LBN→PBN mapping in RAM and allocates blocks sequentially, the same as Dmdedup. We set the sizes of both targets to the maximum offset accessed in the trace. Sequential allocation favors the raw device, but even with this optimization, Dmdedup significantly outperforms the raw device.

We replayed all traces with unlimited acceleration. Figure 4.4 presents the results. Dmdedup performs 1.6–7.2× better than a raw device due to the high redundancy in the traces. Note that write sequentialization can harm read performance by randomizing the read layout. Even so, in the FIU traces (as in many real workloads) the number of writes is higher than the number of reads due to large file system buffer caches. As a result, Dmdedup's overall performance remains high.

4.2.0.3 Performance Comparison to Lessfs

Dmdedup is a practical solution for real storage needs. To our knowledge there are only three other deduplication systems that are free, open-source, and widely used: Lessfs [27], SDFS [53], and ZFS [7]. We omit research prototypes from this list because they are usually unsuitable for production. Both Lessfs and SDFS are implemented in user space, and their designs have much in common. SDFS is implemented using Java, which can add high overhead. Lessfs and Dmdedup are implemented in C, so their performance is more comparable.

	Ext4	Dm dedup 4KB	Lessfs		
			BDB 4KB	BDB 128KB	HamsterDB 128KB TransOFF
Time (sec)	649	521	1,825	1,413	814

Table 4.2: *Time to extract 40 Linux kernels on Ext4, Dmdedup, and Lessfs with different backends and chunk sizes. We turned transactions off in HamsterDB for better performance.*

Table 4.2 presents the time needed to extract a tarball of 40 Linux kernels (uncompressed) to a newly created file system. We experimented with plain Ext4, Dmdedup with Ext4 on top, and Lessfs deployed above Ext4. When setting up Lessfs, we followed the best practices described in its documentation. We experimented with BerkleyDB and HamsterDB backends with transactions on and off, and used 4KB and 128KB chunk sizes. The deduplication ratio was 2.4–2.7 in these configurations. We set the Lessfs and Dmdedup cache sizes to 150MB, which was calculated for our dataset using the db_stat tool from Lessfs. We configured Dmdedup to use the CBT backend because it guarantees transactionality, similar to the databases used as Lessfs backends.

Dmdedup improves plain Ext4 performance by 20% because it eliminates duplicates. Lessfs with the BDB backend and a 4KB chunk size performs 3.5× slower than Dmdedup. Increasing the chunk size to 128KB improves Lessfs's performance, but it is still 2.7× slower than Dmdedup with 4KB chunks. We achieved the highest Lessfs performance when using the HamsterDB backend with 128KB and disabling transactions. However, in this case we sacrificed both deduplication ratio and transactionality. Even then, Dmdedup performs 1.6× faster than Lessfs while providing transactionality and a high deduplication ratio. The main reason for poor Lessfs performance is its high CPU utilization—about 87% during the run. This is caused by FUSE, which adds significant overhead and causes many context switches [47]. To conclude, Dmdedup performs significantly better than other popular, open-source solutions from the same functionality class.

Unlike Dmdedup and Lessfs, ZFS falls into a different class of products because it does not add deduplication to existing file systems. In addition, ZFS deduplication logic was designed to work efficiently when all deduplication metadata fits in the cache. When we limited the ZFS cache size to 1GB, it took over two hours for `tar` to extract the tarball. However, when we made the cache size unlimited, ZFS was almost twice as fast as Dmdedup+Ext4. Because of its complexity, ZFS is hard to set up in a way that provides a fair comparison to Dmdedup; we plan to explore this in the future.

Chapter 5

Hints: Design and Implementation

The block layer has a simple interface to communicate with the upper layers. Due to the simplicity of this interface, data context is lost by the time control reaches the block layer. Deduplication has its own overheads in terms of CPU usage and metadata tracking. Thus it makes sense to incur this overhead only where it will be beneficial. It is also useful to try and reduce the overhead of expensive deduplication operations where possible.

To allow the block layer to be aware of context, we designed a system that lets hints flow from higher to lower layers in the storage stack. Applications and file systems can then communicate important information about their data to lower layers. The red arrows in Figure 5.1 show how hints are passed to the block layer. We have implemented two important hints: NODEDUP and PREFETCH.

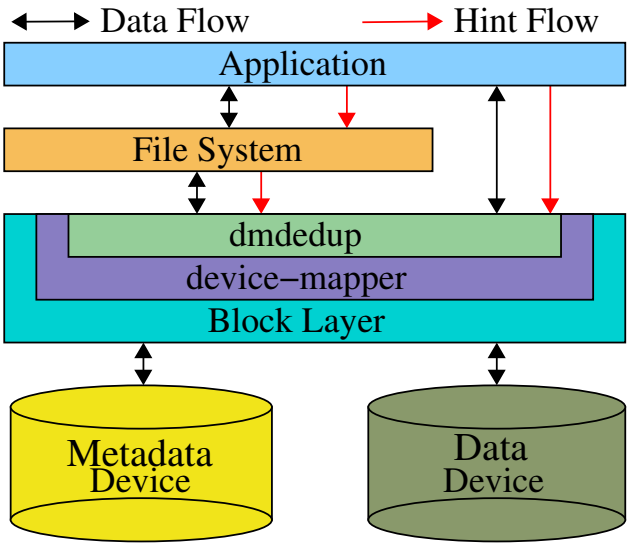


Figure 5.1: *Flow of hints across the storage layers.*

We used our own open-source block-layer deduplication system, *dmdedup* [60], to develop and evaluate the benefits of hints.

Nodedup Since deduplication uses computational resources and may increase latency, it should only be performed when there is a potential benefit. The NODEDUP hint instructs the block layer not to deduplicate a particular chunk (block) on writes. It has three use cases: **(1)** unique data: there is no point in wasting

resources on deduplicating data that is unlikely to have duplicates, such as file metadata; **(2)** short-lived data: deduplicating temporary files may not be beneficial since they will soon be discarded; and **(3)** reliability: maintaining multiple copies of certain blocks may be necessary, such as with the superblock replicas in many file systems.

Prefetch One of the most time-consuming operations in a deduplication system is hash lookup, because it often requires extra I/O operations. Worse, by their very nature, hashes are randomly distributed. Hence, looking up a hash often requires random I/O, which is the slowest operation in most storage systems.

The PREFETCH hint is used to inform the deduplication system of I/O operations that are likely to generate further duplicates (e.g., during a file copy) so that their hashes can be prefetched and cached to minimize random accesses. This hint can be set on the read path for applications that expect to access the same data again.

5.1 Implementation

To add support for hints, we modified various parts of the storage stack. The generic changes to support propagation of hints from higher levels to the block layer modified about 77 lines of code in the kernel. Effort to add application level hint support to other file systems like Btrfs are also currently ongoing. We also modified the OPEN system call to take two new flags, O_NODEDUP and O_PREFETCH. User-space applications can use these flags to pass hints to the underlying deduplication block device. If the block layer does not support the flags, they are ignored. Applications that require redundancy, have a small number of duplicates, or create many temporary files can pass the O_NODEDUP hint when `open`ing for write. Similarly, applications that are aware of popular data blocks, or that know some data will be accessed again can pass O_PREFETCH `open`ing for read. Hashes of the blocks being read can then be prefetched, so that on a later write they can be found in the cache rather than looking them up at that point.

We modified *dmdedup* to support the NODEDUP and PREFETCH hints by adding and modifying about 741 LoC. In *dmdedup*, if a request has the NODEDUP flag set, we skip lookups and updates in the hash index. Instead, we add an entry only to the LBN→PBN mapping. The read path needs no changes to support NODEDUP.

On the read path in *dmdedup*, the LBN→PBN map is consulted to find whether the given location is known, but no hash calculation is normally necessary because a previous write would have already added it to the hash→PBN map. If a request has the PREFETCH hint set on the read path then *dmdedup* hashes the data after it is read and puts the corresponding hash→PBN tuple in a prefetch cache. At write time, our code saves execution time by checking the cache before searching the metadata backend. When a hash is found in the prefetch cache, it is evicted, since after the copy there is little reason to believe that the hash will be used again soon. If the user recopies the file, the data must be reread and thus the copying application will have the opportunity to set the PREFETCH hint again.

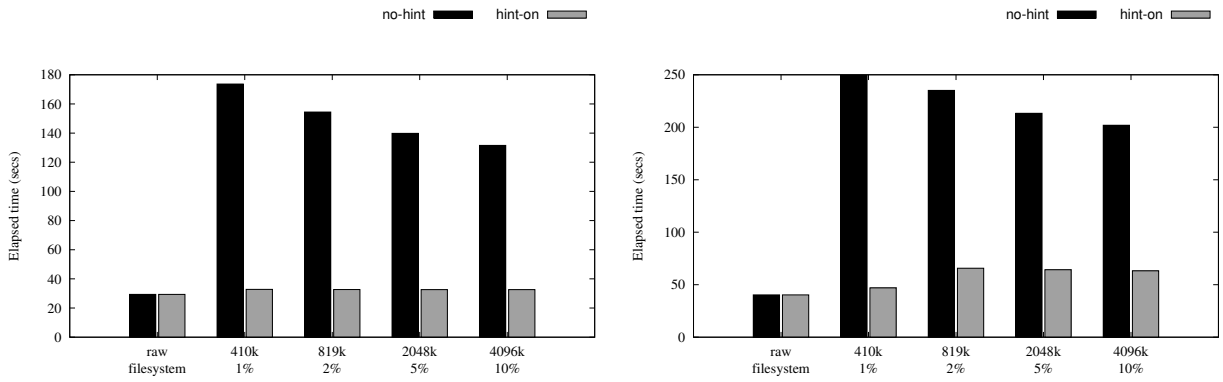
We also modified some specific file systems to pass the NODEDUP hint for their metadata. Experimentally we found that all file-system metadata deduplicates poorly. For Linux's Nilfs2, we changed about 371 kernel LoC to mark its metadata with hints and propagate them from the upper levels to the block layer. Similar changes to Ext4 changed 16 lines of code; in Ext3 we modified 6 lines (which also added support for Ext2). The Ext2/3/4 changes were small because we were able to leverage the (newer) REQ_META flag being set on the file system metadata to decide whether to deduplicate based on data type. The rest of the metadata-related hints are inferred; we identify journal writes from the process name, *jbd2*.

When we began work on hints for *dmdedup*—in kernel version 3.1—we also needed to patch Ext2/3/4 to add flags to mark metadata, which required changing about 300 LoC for each file system. But these changes were unnecessary beginning in Linux 3.10, because Linux developers had recognized the utility of passing

hints to block layers and had begun to mark their metadata with a REQ_META flag. We also ported *dmdedup* along with our changes for hints to Linux kernel 3.17. In recent kernels, REQ_META is also used by the BCACHE device mapper target (block device as cache) to disable metadata read-ahead.

Chapter 6

Hints: Evaluation



(a) Ext4

(b) Nilfs2

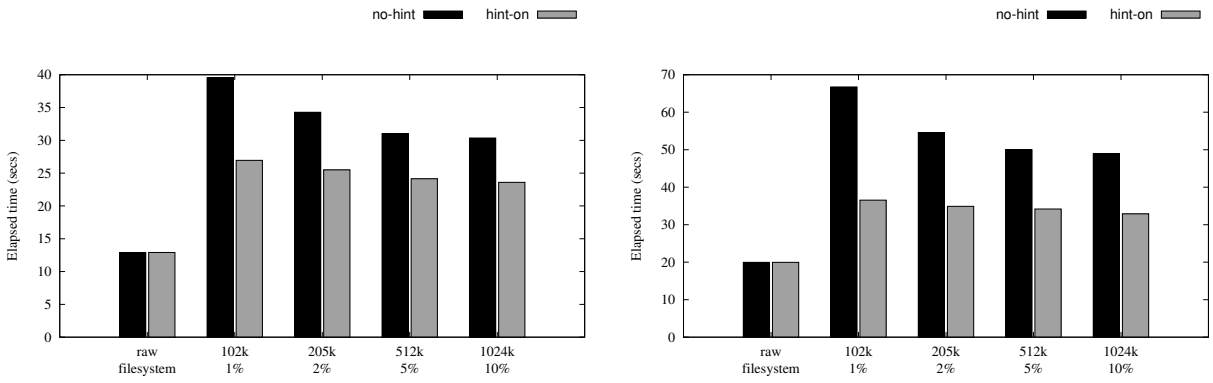
Figure 6.1: *Performance of using* `dd` *to create a 4GB file with unique content, both with and without the* NODEDUP *hint, for different file systems on Machine M1. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by the workload. Metadata was flushed after every 1,000 writes. Lower is better.*

6.1 Experimental Setup

In our experiments we used two systems of different ages and capabilities. Our newer machine (designated M1 from here on) is a Dell PowerEdge R710, equipped with an Intel Xeon E5540 2.4GHz 4-core CPU and 24GB of RAM. The older one (designated M2) has an Intel Core^{TM} 2 Q9400 2.66GHz Quad CPU and 4GB of RAM. Both machines ran Ubuntu Linux 14.04 x86_64, upgraded to a Linux 3.17.0 kernel.

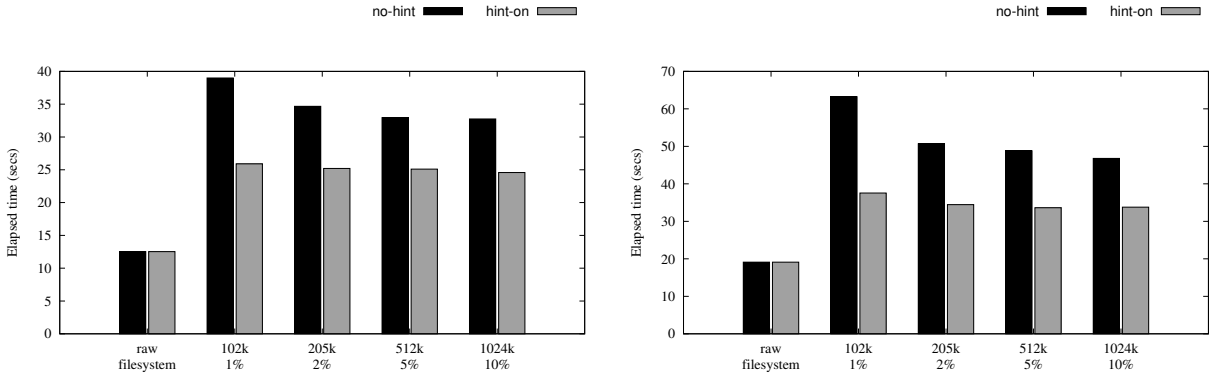
On M1, we used an Intel DC S3700 series 200GB SSD as the *dmdedup* metadata device and a Seagate Savvio 15K.2 146GB disk drive for the data. Both drives were connected to their host using Dell's PERC 6/i Integrated controller. On M2, we used the same disk drive for the data, but for the metadata we used an Intel SSD 320 series of size 300GB with a 3GB/s SATA controller. Although both SSDs are large, in all our experiments we used 1.5GB or less for *dmdedup* metadata.

We ran all experiments at least three times and ensured that standard deviations were less than 5% of the mean. To ensure that all dirty data reached stable media in the micro-workload experiments, we called `sync` at the end of each run and then unmounted the file system; our time measurements include these two steps.



(a) Ext4 (b) Nilfs2

Figure 6.2: *Performance of using* `dd` *to copy a 1GB file, both with and without the* PREFETCH *hint, for different file systems on Machine M1. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by workload. Metadata was flushed after every 1,000 writes. Lower is better.*



(a) Ext4 (b) Nilfs2

Figure 6.3: *Performance of using* `dd` *to copy a 1GB file with deduplication ratio of 2:1, both with and without the* PREFETCH *hint, for different file systems on Machine M1. The X axis lists the metadata cache size used by* dmdedup*, in both absolute values and as a percentage of the total metadata required by workload. Metadata was flushed after every 1,000 writes. Lower is better.*

For all experiments we used *dmdedup*'s *cowbtree* transactional metadata backend as it helps avoid inconsistent metadata state on a crash. We used this backend because it is is the safest and most consistent. *Cowbtree* allows users to specify different metadata cache sizes; we took the total amount of deduplication metadata expected for each experiment, and calculated cache sizes of 1%, 2%, 5%, and 10% of the expectation. These ratios are typical in real deduplication systems. *Dmdedup* also allows users to specify the granularity at which they want to flush metadata. We ran all experiments with two settings: flush metadata on every write, or flush after every 1,000 writes. In our results we focus on the latter case because it is a more realistic setting. Flushing after every write is like using the O_SYNC flag for every operation and is uncommon in real systems; we used that setting to achieve a worst-case estimate. *Dmdedup* also flushes its

metadata when it receives any flush request from the above layers. Thus our granularity of flushing matches that of the file system or application above and ensures that the semantics of the data reaching the disk safely is the same as that of a normal block device without *dmdedup*.

To show the benefits of the NODEDUP hint, we modified `dd` to use the O_NODEDUP open flag when opening files for write. We then used `dd` to write unique data to the storage system, which allowed us to assess the performance of systems that primarily write unique data and to show the benefit of not wasting deduplication effort on such systems.

To show the impact of the PREFETCH hint, we modified `dd` to use the O_PREFETCH open flag on the read path so that writes could benefit from caching hashes. The PREFETCH hint is set on reads to indicate that the same data will be written soon.

We also modified *Filebench* to generate data in the form of a given duplicate distribution instead of arbitrary data. We then used the `Fileserver` workload to generate unique data and test three cases: using no hints, hinting NODEDUP for all file metadata writes, and using the NODEDUP hint for both data and file metadata writes.

6.2 Experiments

We evaluated the NODEDUP and PREFETCH hints for four file systems: Ext2, Ext3, Ext4, and Nilfs2. Ext2 is a traditional FFS-like file system that updates metadata in place; Ext3 adds journaling support; and Ext4 further adds extent support. Nilfs2 is a log-structured file system: it sequentializes all writes and has a garbage-collection phase to remove redundant blocks. We show results only for Ext4 and Nilfs2, because we obtained similar results from the other file systems.

NODEDUP hint To show the effectiveness of application-layer hints, we added the NODEDUP hint as an open flag on `dd`'s write path. We then created a 4GB file with unique data, testing with the hint both on and off. This experiment shows the benefit of the NODEDUP hint on a system where unique data is being written (i.e., where deduplication is not useful), or where reliability considerations trump deduplicating. This hint may not be as helpful in workloads that produce many duplicates. Figure 6.1 shows the benefit of the NODEDUP hint for Ext4 and Nilfs2 when metadata was flushed every 1,000 writes on M1; results for other file systems were similar. We found that the NODEDUP hint decreased unique-data write times by 2.2–5.4× on M1, and by 3.4–6.6× on M2. The reason M2 benefited more from this hint is that it has slower hardware, so avoiding the slower I/O has a larger relative impact.

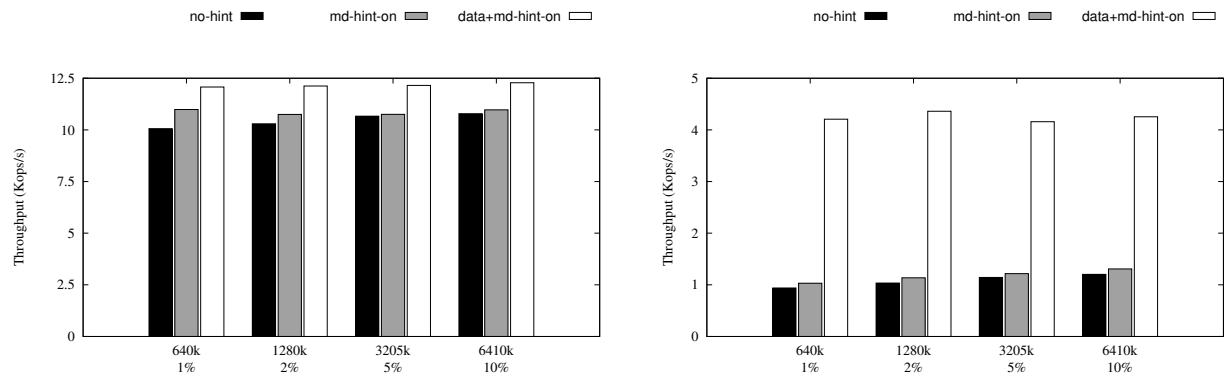
Flushing *dmdedup's* metadata after every write reduced the benefit of the NODEDUP hint, since the I/O overhead is high, but we still observed improvements of 1.5–1.7× for M1, and 1.2–1.7× for M2.

PREFETCH hint To evaluate the PREFETCH hint we repeatedly copied a 1GB file with unique content within a single file system. We used unique content so that we could measure the worst-case performance where no deduplication can happen, and to ensure that the prefetch cache was heavily used. For all four file systems, the results were similar because most file systems manage single-file data-intensive workloads similarly. Figure 6.2 shows results for Ext4 and Nilfs2 on M1. When flushing *dmdedup*'s metadata every 1,000 writes, the reduction in copy time compared to the no-hint configuration was 1.2–1.8× on M1, and 2.9–4.6× on M2.

When we flushed the metadata after every write, the copy times ranged from 10% *worse* to 10% better on M1. On M2, copy times improved by 1.2–1.5×. The improvement from hints was less significant here because the overhead of flushing was higher than the benefit obtained from prefetching the hashes.

We also ran the above experiment with a 1GB file having a dedup ratio of 2:1 (one duplicate block for each unique block) and obtained similar results as the unique file experiments. The reduction in copy time

compared to the no-hint configuration was 1.1–1.6× for flushing *dmdedup*'s metadata every 1,000 writes.



(a) Ext4

(b) Nilfs2

Figure 6.4: *Throughput obtained using* Filebench's `Fileserver` *workload modified to write all-unique content, for different file systems on M1. Throughput is shown with the* NODEDUP *hint off (no-hint); with the hint on for file system metadata only (md-hint-on); and with the hint on for both metadata and data (data+md-hint on). The X axis lists the* dmdedup *metadata cache size, in both absolute values and as a percentage of an estimate of the total metadata required by the workload. Metadata was flushed after every 1,000 writes. Higher is better.*

Macro Workload We ran *Filebench's* `Fileserver` workload to assess the benefit of setting the NODEDUP hint for: (1) file metadata writes, where we mark the metadata blocks and the journal writes with this hint, and (2) file data writes along with the metadata writes. We used a unique-write workload to show the benefits of applying the NODEDUP hint for applications writing unique data. Figure 6.4 shows the maximal benefit of setting the NODEDUP hint on for file metadata writes alone, and for data and metadata writes. We ran Filebench on M1, with the writes being flushed after 1,000 writes, and had Filebench generate all-unique data. When setting the NODEDUP hint only for metadata writes, we saw an increase in throughput by 1–10%. When we set the hint for both data and metadata writes, we saw an improvement in throughput of 1.1–1.2× for Ext4, and 3.5–4.4× for Nilfs2.

We also ran a similar experiment (not shown for brevity) where Filebench generated data with a dedup ratio of 4:1 (3 duplicate blocks for every unique block). We set the NODEDUP hint on for metadata writes only (because Filebench generated unique data on a per-write basis whereas our hint works on a per-file basis), and compared this to the case where the NODEDUP hint was off. On M1, we saw a modest improvement in throughput ranging from 3–6% for Ext4, and 6–10% for Nilfs2.

Chapter 7

Related Work

Many previous deduplication studies have focused on backup workloads [44, 71]. Several studies have also characterized primary datasets from the deduplication perspective and showed that typical user data contains a lot of duplicates [13, 31, 35]. So, although Dmdedup can be used for backups, it is intended as a general-purpose deduplication system. Thus, in this section we focus on primary-storage deduplication.

Several studies have incorporated deduplication into existing file systems [38, 57] and a few have designed deduplicating file systems from scratch [7]. Although there are advantages to deduplicating inside a file system, doing so is less flexible for users and researchers because they are limited by that system's architecture. Dmdedup does not have this limitation: it can be used with any file system. FUSE-based deduplication file systems are another popular design option [27, 53]; however, FUSE's high overhead makes this approach impractical for production environments [47]. To address performance problem, El-Shimi et al. built an in-kernel deduplication file system as a filter driver for Windows [13] at the price of extra development complexity.

The systems most similar to ours are Dedupv1 [33] and DBLK [64], both of which deduplicate at the block level. Each is implemented as a user-space iSCSI target, so their performance suffers from additional data copies and context switches. DBLK is not publicly available.

It is often difficult to experiment with existing research systems. Many are raw prototypes that are unsuitable for extended evaluations, and only a few have made their source code available [7, 33]. Others were intended for specific experiments and lack experimental flexibility [67]. High-quality production systems have been developed by industry [6, 57] but it is hard to compare against unpublished, proprietary industry products. In contrast, Dmdedup has been designed for experimentation, including a modular design that makes it easy to try out different backends.

The semantic divide between the block layer and file systems has been addressed previously [3,10,54–56] and has received growing attention because of the widespread use of virtualization and the cloud, which places storage further away from applications [20, 21, 32, 48, 50, 61].

An important approach to secondary storage is ADMAD, which performs application-aware file chunking before deduplicating a backup [30]. This is similar to our hints interface, which can be easily extended to pass application-aware chunk-boundary information.

Many researchers have proposed techniques to prefetch fingerprints and accelerate deduplication filtering [28, 68, 71]. While these techniques could be added to *dmdedup* in the future, our current focus is on providing semantic hints from higher layers, which we believe is an effective complementary method for accelerating performance. In addition, some of these past techniques rely on workload-specific data patterns (e.g., backups) that might not be beneficial in general-purpose in-line primary-storage deduplication systems.

Some have studied memory deduplication in virtualized environments [36,37], which is helpful in closing the semantic gap caused by multiple virtualization layers. There, memory is scanned by the host OS to identify and merge duplicate memory pages. Such scanning is expensive, misses short-lived pages, and is

slow to identify longer-lived duplicates. These studies found that pages in the guest's unified buffer cache are good sharing candidates, so marking requests from the guest OS with a dedup hint can help to quickly identify potential duplicates. This approach is specific to memory deduplication and may not apply to storage systems where we identify duplicates even before writing to the disk.

Lastly, others have demonstrated a loss of potential deduplication opportunities caused by intermixing metadata and data [29], showing that having hints to avoid unnecessary deduplication might be beneficial.

Chapter 8

Conclusions and Future Work

Primary-storage deduplication is a rapidly developing field. To test new ideas, previous researchers had to either build their own deduplication systems or use closed-source ones, which hampered progress and made it difficult to fairly compare deduplication techniques. We have designed and implemented Dmdedup, a versatile and practical deduplication block device that can be used by regular users and researchers. We developed an efficient API between Dmdedup and its metadata backends to allow exploration of different metadata management policies. We designed and implemented three backends (INRAM, DTB, and CBT) for this report and evaluated their performance and resource use. Extensive testing demonstrates that Dmdedup is stable and functional, making evaluations using Dmdedup more realistic than experiments with simpler prototypes. Thanks to reduced I/O loads, Dmdedup improves system throughput by 1.5–6× under many realistic workloads.

Deduplication at the block layer has two main advantages: (1) allowing any file system and application to benefit from deduplication, and (2) ease of implementation [60]. Unfortunately, application and file system context is lost at the block layer, which can harm deduplication's effectiveness. However, by adding simple yet powerful hints, we were able to provide the missing semantics to the block layer, allowing the dedup system to improve performance and possibly also reliability.

Our experiments show that adding the NODEDUP hint to applications like `dd` can improve performance by up to 5.3× when copying unique data, since we avoid the overhead of deduplication for data that is unlikely to have duplicates. This hint can be extended to other applications, such as those that compress or encrypt, and even those that create many temporary files. Adding the PREFETCH hint to applications like `dd` improved copying time by as much as 1.8× because we cache the hashes and do not need to access the metadata device to fetch them on the write path. Adding hints to macro workloads like *Filebench's* `Fileserver` workload improved throughput by as much as 4.4×.

Future work. Dmdedup provides a sound basis for rapid development of deduplication techniques. By publicly releasing our code we hope to spur further research in the systems community. We plan to develop further metadata backends and cross-compare them with each other. For example, we are considering creating a log-structured backend.

Compression and variable-length chunking can improve overall space savings but require more complex data management, which might decrease system performance. Therefore, one might explore a combination of on-line and off-line deduplication.

Aggressive deduplication might reduce reliability in some circumstances; for example, if all copies of an FFS-like file system's super-block were deduplicated into a single one. We plan to explore techniques to adapt the level of deduplication based on the data's importance. We also plan to work on automatic scaling of the hash index and LBN mapping relative to the size of metadata and data devices.

Also, due of the success of our initial experiments, we intend to add hint support for other file systems, such as Btrfs and XFS. We also plan to implement other hints, discussed in Section 2.2.1, to provide richer context to the block layer.

Acknowledgements

Bibliography

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. Klein. The design of similarity based deduplication system. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[2] Storage Networking Industry Association. IOTTA trace repository. http://iotta.snia.org, February 2007. Cited December 12, 2011.

[3] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A non-invasive exclusive caching mechanism for RAIDs. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 176–187, Washington, DC, USA, 2004. IEEE Computer Society.

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

[5] 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.

[6] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur. Single instance storage in Windows 2000. In *Proceedings of USENIX Annual Technical Conference*, 2000.

[7] J. Bonwick. ZFS deduplication, November 2009. http://blogs.oracle.com/bonwick/entry/zfs_dedup.

[8] Bzip2 data compression utility. http://bzip.org/.

[9] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press.

[10] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[11] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[12] M. Dutch. Understanding data deduplication ratios. Technical report, SNIA Data Management Forum, 2008.

[13] A. El-Shimi, R. Kalach, A. Kumar, A. Oltean, J. Li, and S. Sengupta. Primary data deduplication—large scale study and system design. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[14] Ext4. http://ext4.wiki.kernel.org/.

[15] Filebench, 2016. http://filebench.sf.net.

[16] John Gantz and David Reinsel. Extracting value from chaos. IDC 1142, June 2011.

[17] J. Gray and C. V. Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, December 2005.

[18] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press.

[19] Microsoft Exchange Server JetStress 2010. www.microsoft.com/en-us/download/details.aspx?id=4167.

[20] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007.

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[22] A. Katiyar and J. Weissman. ViDeDup: An application-aware framework for video de-duplication. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[23] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, 2010.

[24] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 105–118, Boston, MA, June 2001. USENIX Association.

[25] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 291–303, June 1994.

[26] D. Lelewer and D. Hirschberg. Data Compression. In *ACM Computing Surveys (CSUR)*, pages 261–296. ACM, 1987.

[27] Lessfs, January 2012. www.lessfs.com.

[28] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[29] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. Metadata considered harmful ... to deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[30] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D.-S. Wang. ADMAD: Application-driven metadata aware de-duplication archival storage system. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.

[31] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for data reduction in primary storage: A practical analysis. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR)*, 2012.

[32] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance oriented I/O deduplication for primary storage systems in the cloud. In *28th International IEEE Parallel and Distributed Processing Symposium*, 2014.

[33] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proceedings of the MSST Conference*, 2010.

[34] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[35] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[36] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*, London, UK, March 2012.

[37] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, San Jose, CA, 2013. USENIX.

[38] A. More, Z. Shaikh, and V. Salve. DEXT3: Block level inline deduplication for EXT3 file system. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2012.

[39] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 2001.

[40] P. Nath, M. Kozuch, D. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[41] Nimble's Hybrid Storage Architecture. www.nimblestorage.com/products/architecture.php.

[42] Oracle. Btrfs, 2008. http://oss.oracle.com/projects/btrfs/.

[43] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[44] 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)*, 2002.

[45] 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.

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

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

[48] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Seattle, WA, March 2008. ACM.

[49] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[50] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. de Lara, V. Bala, and M. Satyanarayanan. Privacy-sensitive VM retrospection. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[51] David Rosenthal. Deduplicating devices considered harmful. *Queue*, 9(5):30:30–30:31, May 2011.

[52] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[53] Opendedup, January 2012. www.opendedup.org.

[54] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[55] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[56] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[57] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, 2012.

[58] 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.

[59] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[60] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, and S. Trehan. Dmdedup: Device-mapper deduplication target. In *Proceedings of the Linux Symposium*, pages 83–95, Ottawa, Canada, July 2014.

[61] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, June 2013.

[62] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[63] Joe Thornber. *Persistent-data library*, 2011. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/persistent-data.txt.

[64] Y. Tsuchiya and T. Watanabe. DBLK: Deduplication for primary block storage. In *Proceedings of the MSST Conference*, 2011.

[65] WD Blue. http://www.wd.com/en/products/products.aspx?id=800#tab11.

[66] Fusion Drive. http://en.wikipedia.org/wiki/Fusion_Drive.

[67] A. Wildani, E. Miller, and O. Rodeh. HANDS: A heuristically arranged non-backup in-line deduplication system. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.

[68] 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*, 2011.

[69] F. Xie, M. Condict, and S. Shete. Estimating duplication by content-based sampling. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[70] E. Zadok. Writing Stackable File Systems. *Linux Journal*, 05(109):22–25, May 2003.

[71] 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)*, 2008.