Dmdedup: Device Mapper Target for Data Deduplication

Appears in the Proceedings of the 2014 Ottawa Linux Symposium (OLS'14)

Vasily Tarasov
Stony Brook University & IBM Research
tarasov@vasily.name

Deepak Jain
Stony Brook University
dpakjain@gmail.com

Geoff Kuenning
Harvey Mudd College
geoff@cs.hmc.edu

Sonam Mandal
Stony Brook University
somandal@cs.stonybrook.edu

Karthikeyani Palanisami
Stony Brook University
karthikeyani.palanisami@gmail.com

Philip Shilane
EMC
philip.shilane@emc.com

Sagar Trehan
Stony Brook University
sagartrehan@gmail.com

Erez Zadok
Stony Brook University
ezk@fsl.cs.sunysb.edu

Abstract

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×.

1 Introduction

As storage demands continue to grow [2], 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 [18, 27]. In recent years, however, primary datasets have also expanded substantially [14], and researchers have begun to explore *primary* storage deduplication [13, 21].

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 [3, 21, 23]. 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 database) 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). In this paper, we present Dmdedup's design, demonstrate its flexibility, and evaluate its performance, memory usage, and space savings under various workloads.

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 paper, we present our experiences and lessons learned while designing a variety of metadata backends, and include detailed experimental results. We believe that our results and open-source deduplication platform can significantly advance primary deduplication solutions.

2 Design

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

2.1 Classification

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 [15] or WAFL [21]; (2) creating a stackable deduplicating file system either in-kernel [26] or using FUSE [11,20]; or (3) implementing a new deduplicating file system from scratch, such as EMC Data Domain's file system [27]. 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 [19]. 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, GPFS, BTRFS, 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. In the future, we plan to recover some of the lost context using file system and application hints.

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. But it risks negatively impacting the performance of primary workloads. Only a few studies have addressed this issue [21,24]. Dmdedup performs inline deduplication; we discuss its performance in Section 4.

2.2 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 [11, 13, 20] DM operates in the kernel, which improves performance yet does not prohibit communication with user-space daemons when appropriate [19].

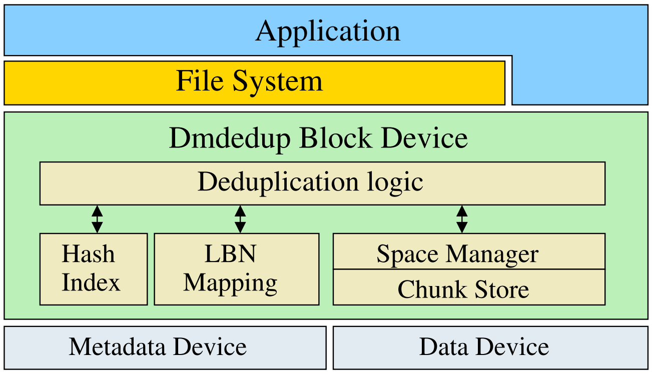


Figure 1: Dmdedup high-level design.

2.3 Dmdedup Components

Figure 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. 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.

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.

Dmdedup main components are (Figure 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 *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.

2.4 Write Request Handling

Figure 2 shows how Dmdedup processes write requests.

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 [14], 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. 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) [16] 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.

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

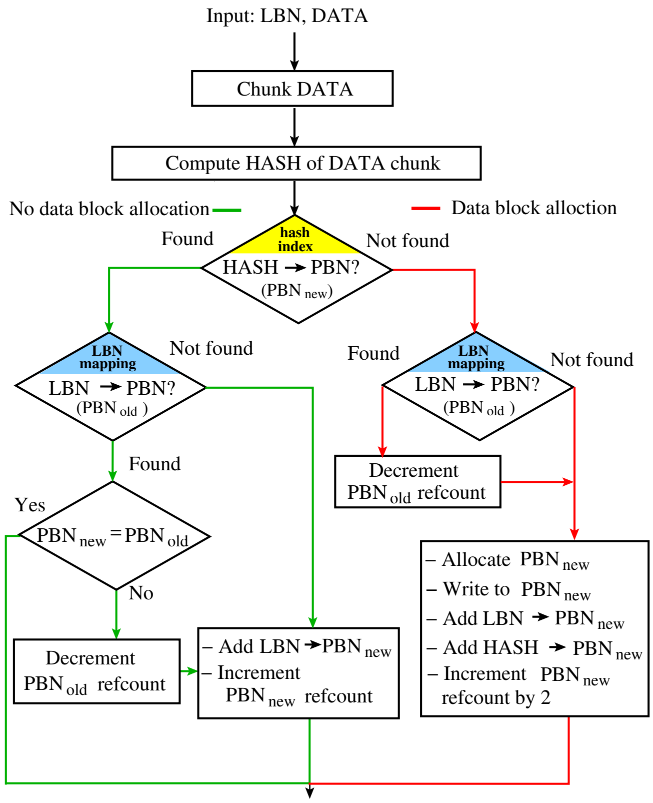


Figure 2: Dmdedup write path. *PBN_{new}* is the PBN found in the hash index: a new physical location for incoming data. *PBN_{old}* is the PBN found in the LBN mapping: the old location of data, before the write ends.

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} [9]. 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× [12], 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 2. 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 [13]. 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 [17]. 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.

2.5 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.

2.6 Metadata Backends

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**.

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. During the course of the project, several junior programmers were asked to develop experimental backends for Dmdedup; anecdotally, they were able to accomplish their task in a short time and without changing the API.

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.

2.6.1 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.

2.6.2 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.

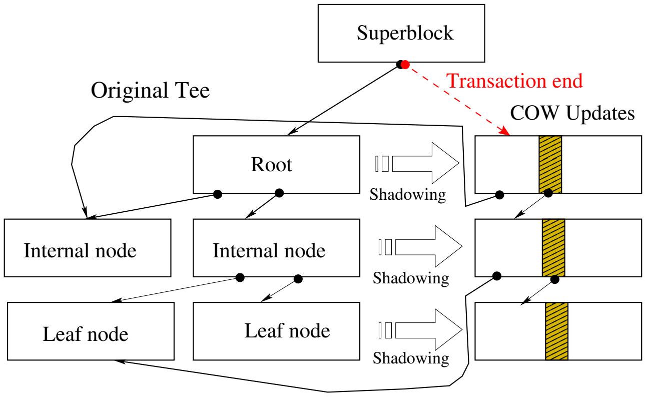


Figure 3: Copy-on-Write (COW) B-trees.

2.6.3 CBT Backend

Unlike the INRAM and DTB backends, CBT provides true transactionality. It uses Linux's on-disk Copy-On-Write (COW) B-tree implementation [6, 22] to organize its key-value stores (see Figure 3). 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 predefined 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 [25], 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 Implementation

The latest Dmdedup code has been tested against Linux 3.14 but we performed experimental evaluation on version 3.10.9. 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.

4 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 [11]—an alternative, popular deduplication system.

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 2.6.2, 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 Dmdedup

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

4.2.1 Micro-workloads

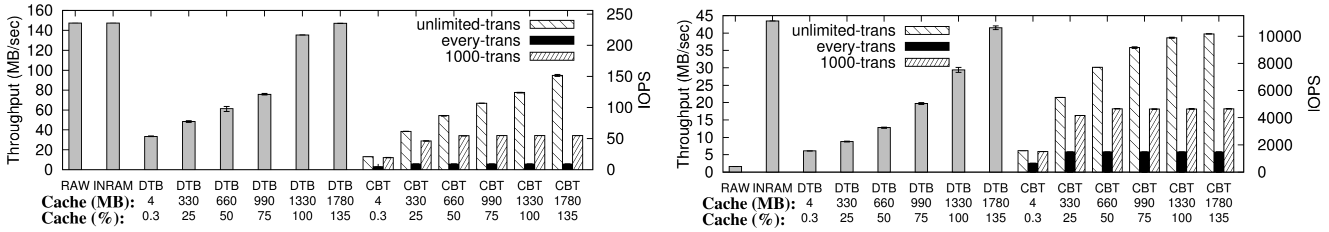
Unlike traditional (non-deduplicating) storage, a deduplication system's performance is sensitive to data content. For deduplication systems, a key content characteristic is its deduplication ratio, defined as the number of logical blocks divided by the number of blocks physically stored by the system [7]. Figures 4, 5, and 6 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 4–6) and small random writes (subfigures (b) in Figures 4–6). 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 randomly (e.g., databases) [5]. 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). 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(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 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.

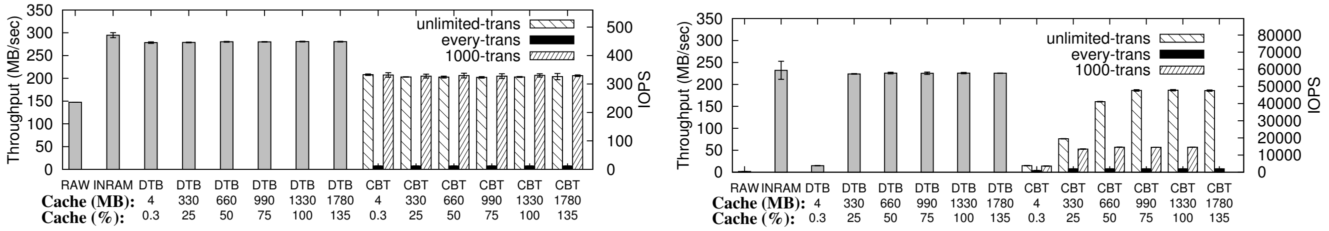
The CBT backend behaves similarly to DTB but its performance is always lower—between 3–95MB/sec depending on the cache and transaction size. The reduced performance is caused by an increased number of I/O operations, 40% more than DTB. The transaction size significantly impacts CBT's performance; an 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



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

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

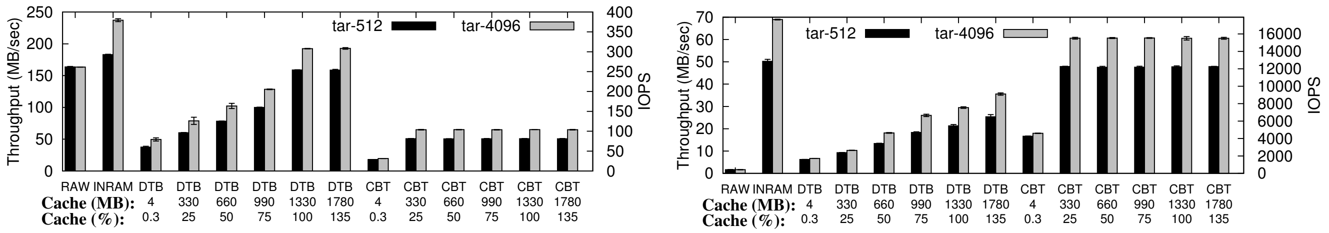
Figure 4: Sequential and random write throughput for the **Unique** dataset. Results are for the raw device and for Dmdedup with different metadata backends and cache sizes. For the CBT backend we varied the transaction size: unlimited, every I/O, and 1,000 writes.



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

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

Figure 5: 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.



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

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

Figure 6: 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.

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.

For random-write workloads (Figure 4(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 [13,23,27], 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 5). 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 5(a)). In the latter case, Dmdedup's throughput falls to 12MB/sec due to the many (ten) metadata writes induced by each user

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 1: *Summary of FIU trace characteristics*

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 5(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 6). 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 (16GB/1.88 = 8.5GB) is actually smaller than the tarball produced by tar-512 (11GB/1.18 = 9.3GB).

For sequential writes (Figure 6(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 6(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.

4.2.2 Trace Replay

To evaluate Dmdedup's performance under realistic workloads, we used three production traces from Florida International University (FIU): Web, Mail, and Homes [1, 10]. 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 1 presents relevant characteristics of these traces.

FIU's traces contain data hashes for every request. We applied a patch from Koller and Rangaswami [10] 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

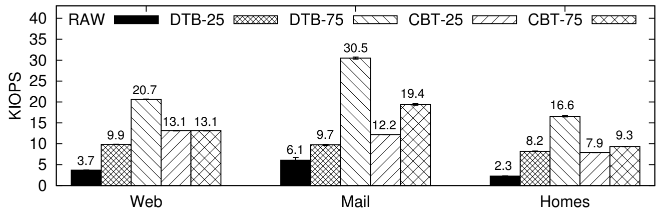


Figure 7: Raw device and Dmdedup throughput for FIU's Web, Mail, and Homes traces. The CBT backend was setup with 1000-writes transaction sizes.

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 7 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.

4.2.3 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 [11], SDFS [20], and ZFS [4]. 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.

Table 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.

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

Table 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.*

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 [19]. 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.

5 Related Work

Many previous deduplication studies have focused on backup workloads [18, 27]. 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 [15, 21] and a few have designed deduplicating file systems from scratch [4]. Although there are advantages to deduplicating inside a file system, doing so is less flexible for users and researchers because they are limited to 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 [11,20]; however, FUSE's high overhead makes this approach impractical for production environments [19]. To address performance problem, El-Shimi et al. built an in-kernel deduplication file system as a filter driver for Windows [8] at the price of extra development complexity.

The systems most similar to ours are Dedupv1 [13] and DBLK [23], 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 [4,13]. Others were intended for specific experiments and lack experimental flexibility [24]. High-quality production systems have been developed by industry [3, 21] 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.

6 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 paper 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.

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.

Acknowledgments. This work was made possible in part thanks to NSF awards CNS-1305360, IIS-1251137, and CNS-1302246, as well as an EMC Faculty award. We also thank Teo Asinari, Mandar Joshi, Atul Karmarkar, Gary Lent, Amar Mudrankit, Eric Mueller, Meg O'Keefe, and Ujwala Tulshigiri for their valuable contributions to this projects. We appreciate Raju Rangaswami's and Ricardo Koller's help on understanding and replaying FIU traces.

References

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

[2] 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.

[3] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the ATC Conference*, 2000.

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

[5] 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)*, 2011.

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

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

[8] 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 ATC Conference*, 2012.

[9] 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.

[10] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of the FAST Conference*, 2010.

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

[12] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *Proceedings of the SYSTOR Conference*, 2012.

[13] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Proceedings of the MSST Conference*, 2010.

[14] D. Meyer and W. Bolosky. A Study of Practical Deduplication. In *Proceedings of the FAST Conference*, 2011.

[15] 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.

[16] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the ACM Symposium on Operating System Principles*, 2001.

[17] 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 ATC Conference*, 2006.

[18] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the FAST Conference*, 2002.

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

[20] Opendedup - SDFS, January 2012. www.opendedup.org.

[21] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the FAST Conference*, 2012.

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

[23] Y. Tsuchiya and T. Watanabe. DBLK: Deduplication for Primary Block Storage. In *Proceedings of the MSST Conference*, 2011.

[24] 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.

[25] F. Xie, M. Condict, and S. Shete. Estimating Duplication by Content-based Sampling. In *Proceedings of the ATC Conference*, 2013.

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

[27] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the FAST Conference*, 2008.