Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick Lee, John Lui

Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui

Dept of Computer Science and Engineering,
The Chinese University of Hong Kong, Hong Kong
{chng,mcma,tywong,pclee,cslui}@cse.cuhk.edu.hk

Abstract. Deduplication is an approach of avoiding storing data blocks with identical content, and has been shown to effectively reduce the disk space for storing multi-gigabyte virtual machine (VM) images. However, it remains challenging to deploy deduplication in a real system, such as a cloud platform, where VM images are regularly inserted and retrieved. We propose LiveDFS, a live deduplication file system that enables deduplication storage of VM images in an open-source cloud that is deployed under low-cost commodity hardware settings with limited memory footprints. LiveDFS has several distinct features, including spatial locality, prefetching of metadata, and journaling. LiveDFS is POSIX-compliant and is implemented as a Linux kernel-space file system. We deploy our LiveDFS prototype as a storage layer in a cloud platform based on OpenStack, and conduct extensive experiments. Compared to an ordinary file system without deduplication, we show that LiveDFS can save at least 40% of space for storing VM images, while achieving reasonable performance in importing and retrieving VM images. Our work justifies the feasibility of deploying LiveDFS in an open-source cloud.

Keywords: Deduplication, virtual machine image storage, open-source cloud, file system, implementation, experimentation

1 Introduction

Cloud computing makes computing and storage resources available to users on demand. Users can purchase resources from commercial cloud providers (e.g., Amazon EC2 [1]) in a pay-as-you-go manner [2]. On the other hand, commercial clouds may not be suitable for some users, for example, due to security concerns [25]. In particular, from developers' perspectives, commercial clouds are externally owned and it is difficult to validate new methodologies for a cloud without re-engineering the cloud infrastructures. An alternative is to deploy a self-manageable cloud using *open-source* cloud platforms, such as Eucalyptus [18] and OpenStack [22]. Such open-source cloud platforms can be deployed within in-house data centers as private clouds, while providing functionalities similar to commercial clouds. For example, Eucalyptus resembles the functionalities of Amazon EC2. Note that an open-source cloud can be deployed using *low-cost commodity hardware and operating systems* that are commonly available to general users [18].

Similar to commercial clouds, an open-source cloud should provide users with *virtual machines (VMs)*, on which standard applications such as web servers and file storage can be hosted. To make deployment flexible for different needs of users and applications, it is desirable for the open-source cloud to support a variety of versions of VMs for different types of configurations (e.g., 32-bit/64-bit hardware, file systems, operating systems, etc). A major challenge is to scale up the storage of a large number of VM images, each of which is a file that could be of gigabytes. Certainly, increasing the storage capacity for hosting VM images is one option. However, this also implies higher operating costs for deploying an open-source cloud under commodity settings.

One promising technology for improving storage efficiency of VM images is *deduplication*, which eliminates redundant data blocks by creating smaller-size pointers to reference an already stored data block that has identical content. One major application of deduplication is the data backup in Content Addressable Storage (CAS) systems [23], in which each data block is identified by its *fingerprint* computed from a collision-resistant hash of the *content* of the data block. If two data blocks have the same fingerprint, then they are treated as having the same content. Recent studies [13, 11] show that the VM images of different versions of the same Linux distribution generally have a high proportion of identical data blocks (e.g., about 30% of overlap in adjacent Fedora distributions [13]). Hence, deduplication can actually enhance the storage utilization of VM images. On the other hand, to enable deduplication for VM image storage in a cloud, we need to address several deployment issues:

- **Performance of VM operations.** Existing studies mainly focus on the effectiveness of using deduplication to save space for storing VM images, but there remain open issues regarding the deployment of deduplication for VM image storage. In particular, it remains uncertain if deduplication degrades the performance of existing VM operations, such as VM startup.
- **Support of general file system operations.** To make the management of VM images more effective, a deduplication solution should allow general file system operations such as data modification and deletion. For example, if an old VM version is no longer used, then we may want to purge the VM image file from the cloud for better maintenance. However, current deduplication techniques are mainly designed for backup systems, which require data be immutable and impose a write-once policy [23] to prevent data from being modified or deleted.
- **Compatibility with low-cost commodity settings.** Although there have been commercial file systems (e.g., SDFS [20] and ZFS [21]) that support efficient I/O operations while allowing deduplication, they are mainly designed for enterprise servers with a large capacity of main memory. Some deduplication systems [8, 15] use flash memory to relieve the main memory requirement of deduplication. To make deduplication compatible with commodity settings, a deduplication solution should preserve the I/O performance with reasonable memory footprints and standard commodity hardware configurations.

In this paper, we present a live deduplication file system called *LiveDFS*, which enables deduplication storage of VM image files in an open-source cloud. In particular, we target the open-source cloud platforms that are deployed in low-cost commodity hardware and operating systems. LiveDFS supports general file system operations, such

as read, write, delete, while allowing *inline deduplication* (i.e., on-the-fly deduplication is applied to data that is to be written to the disk). LiveDFS consists of several design features that make deduplication efficient and practical.

- **Spatial locality.** LiveDFS stores only partial deduplication metadata (e.g., fingerprints) in memory for indexing, but puts the full metadata on disk. To mitigate the overhead of accessing the metadata on disk, LiveDFS exploits spatial locality by carefully placing the metadata next to their corresponding data blocks with respect to the underlying disk layout.
- **Prefetching of metadata.** LiveDFS prefetches deduplication metadata of the data blocks in the same block group into the page cache (i.e., the disk cache of Linux). This further reduces the seek time of updating both metadata and data blocks on the disk.
- **Journaling.** LiveDFS supports journaling, which keeps track of file system transactions and enables crash recovery of both data blocks and fingerprints. In addition, LiveDFS exploits the underlying journaling design to combine block writes in batch and reduce disk seeks, thereby improving the write performance.

LiveDFS is *POSIX-compliant*, so its above design features are implemented in such a way that is compliant with the Linux file system layout. It is implemented as a Linux kernel-space driver module, which can be loaded to the Linux kernel without the need of modifying and re-compiling the kernel source code. To justify the practicality of LiveDFS, we integrate it into an open-source cloud platform based on OpenStack [22]. Thus, LiveDFS serves as a storage layer between cloud computing nodes and the VM storage backend. We conduct extensive experiments and compare LiveDFS and the Linux Ext3 file system (Ext3FS). We show that LiveDFS saves at least 40% of storage space for VM images compared to Ext3FS. Given that deduplication introduces fragmentation [24], we also evaluate the performance overhead of LiveDFS in inserting and retrieving VM images in a cloud setting. To our knowledge, this is the first work that addresses the practical deployment of live deduplication for VM image storage in an open-source cloud.

The remainder of the paper proceeds as follows. In Section 2, we present the design of LiveDFS as a deduplication-enabled file system. In Section 3, we explain how LiveDFS is implemented and can be deployed in an open-source cloud based on OpenStack. In Section 4, we present the empirical experimental results. In Section 5, we review the related work in applying deduplication in storage. Finally, in Section 6, we conclude this paper and present future work.

2 LiveDFS Design

LiveDFS is a file system that implements inline deduplication for VM storage and is deployed as a storage layer for an open-source cloud. It is designed for *commodity hardware and operating systems*. For commodity hardware, we consider the native 32-bit/64-bit hardware systems with a few gigabytes of memory. Specifically, we seek to reduce the required memory capacity to reduce the hardware cost. For commodity operating systems, we consider Linux, on which LiveDFS is developed.

We make the following assumptions for LiveDFS design. LiveDFS is deployed in a single storage partition. It only applies deduplication to the stored data within the same partition, but not for the same data stored in different partitions. Nevertheless, it is feasible for a partition to have multiple storage devices, such that deduplication is applied in the file-system level, while data striping is applied in the storage devices and is transparent to the file system. In addition, LiveDFS mainly targets for VM image storage. We do not consider applying deduplication for other types of data objects, which may not have any content similarities for deduplication (e.g., encrypted files).

2.1 Primitives

We design LiveDFS as a branch of the Linux Ext3 file system (Ext3FS) [6]. LiveDFS supports general file system I/O operations such as read, write, and delete. It also supports other standard file system operations for files, directories, and metadata (e.g., changing directories, renaming a file, setting file attributes). Figure 1 depicts the file system layout of LiveDFS, which follows the layout of Ext3FS except that LiveDFS allows block sharing. In the following, we explain the major primitives of LiveDFS as a deduplication-enabled file system.

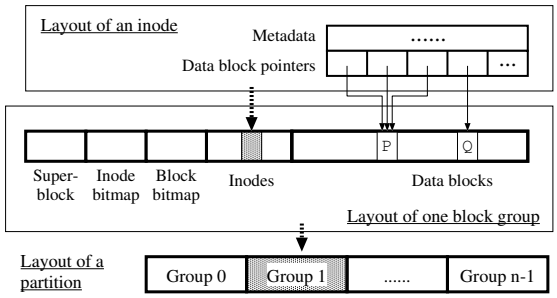


Fig. 1. File system layout of LiveDFS, which is similar to the Ext3FS but supports block sharing.

Typical storage systems organize data into *blocks*, which could be of fixed size or variable size. Most CAS backup systems divide data into variable-size blocks, so as to exploit different granularities of duplicated contents and achieve a higher deduplication rate. The merits of using variable-size blocks in backup systems are studied in [28]. Nevertheless, we choose the fixed-size block implementation in LiveDFS with two reasons. First, most commodity CPUs (e.g., Intel x86) support fixed-size memory pages only. Thus, most mainstream file systems adopt the fixed-size block design to optimize the use of memory pages. Second, [11] shows that for VM image storage, the deduplication efficiencies of using fixed-size and variable-size blocks are similar. Hence, we define a *block* as a fixed-size data unit thereafter.

Similar to Ext3FS, LiveDFS arranges blocks into *block groups* (see Figure 1), each storing the metadata of the blocks within the same group. One advantage of using block groups is to reduce the distance between the metadata and their corresponding blocks on disk, thereby saving the disk seek overhead.

In Ext3FS, each file is allocated an *inode*, which is a data structure that stores the metadata of the file. In particular, an inode holds a set of block pointers, which store the *block numbers* (or addresses) of the blocks associated with the file. LiveDFS also exploits the design of an inode and uses block numbers to refer to blocks. In particular, if two blocks have the same content, then we set their block numbers to be the same (i.e., both of them point to the same block). This realizes the *block-sharing feature* of a deduplication file system. Note that a shared block may be referenced by a single or different files (or inodes).

To enable inline deduplication, LiveDFS introduces two primitives: *fingerprints* and *reference counts*. LiveDFS identifies each block by a fingerprint, which is a hash of the block content. If the fingerprints are collision-resistant cryptographic hash values (e.g., MD5, SHA-1), then it is practical to assume that two blocks having different contents will return two different fingerprints [23]. Since the fingerprint size is much smaller than the block size, we can easily identify duplicate blocks by checking if they have the same fingerprint. Also, to allow block modification and deletion, LiveDFS associates a reference count with each block, such that it keeps the number of block pointers that refer to the block. We increment the reference count of a block if a new block with the same content is written, and decrement it if a block is deleted.

We now describe how LiveDFS performs file system operations. Since LiveDFS is built on Ext3FS, most of its file system operations are the same as those of Ext3FS, except that LiveDFS integrates deduplication into the write operation. LiveDFS implements *copy-on-write* at the block level. If we update a block that is shared by multiple block pointers, then we either allocate a new block from the disk and write the updated content to the disk, or "*deduplicate*" the updated block with an existing block that has the same content. Note that the fingerprints and reference counts of the blocks are updated accordingly. If the reference count is decremented to zero, then LiveDFS deallocates the block as in Ext3FS.

To preserve the performance of file system operations when enabling deduplication, a critical issue is how to maintain the fingerprints and reference counts of all data blocks in a disk partition, such that we can search and update their values efficiently during deduplication. We elaborate this in Section 2.2.

2.2 Deduplication Design

To enable efficient search of fingerprints during deduplication, one option is to keep all fingerprints in main memory, but this significantly increases the memory cost. For example, if we use a block size of 4KB with 16-byte MD5 fingerprints, then 1TB of data will require 4GB of fingerprint space. As a result, we use an implementation that is similar to approaches in [24, 28] to manage the fingerprints. We introduce two components: a set of *fingerprint stores* and a *fingerprint filter*. We keep the full fingerprints in a set of fingerprint stores, which reside on disk. On the other hand, we use the fingerprint filter, which resides in main memory, to speed up the search of fingerprints. Specifically, we differentiate our work from [24, 28] by carefully placing fingerprints on disk based on the file system layout to further reduce the disk seek overhead during deduplication, as elaborated below.

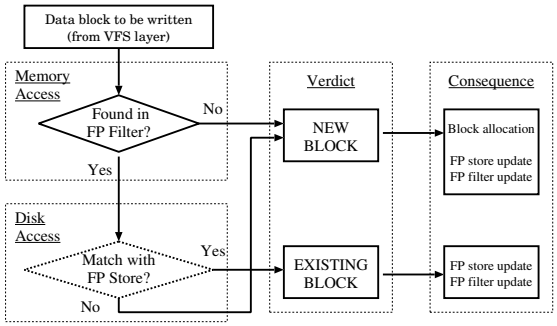


Fig. 2. Every block written to LiveDFS goes through the same decision process.

Overview. The main challenge of LiveDFS is about the writing of data blocks. Figure 2 shows how LiveDFS handles that issue. Whenever a data block arrives at LiveDFS (and we name such a block an incoming block thereafter), its fingerprint is generated. We use the fingerprint of the incoming block to determine if the incoming block is unique.

The first checkpoint is the fingerprint filter ("`FP filter`" in Figure 2). The fingerprint filter is a memory-based filter that aims to determine if the incoming block can be deduplicated. If the incoming block is new to the file system, then it can be directly written to the disk. The design of the fingerprint filter will be detailed in later discussion.

Recalling that the fingerprint filter does not store any complete fingerprints, so the next step is to access the corresponding fingerprint store ("`FP store`" in Figure 2) on disk in order to confirm if the incoming block can actually be deduplicated. If the target fingerprint store does not contain the fingerprint of the incoming block, then it implies that the fingerprint filter gives a false-positive result and that the incoming block is unique; otherwise, the block is not unique and can be deduplicated.

In the following, we first elaborate the design of a fingerprint store, followed by that of the fingerprint filter.

Fingerprint store. LiveDFS exploits *spatial locality* for storing fingerprints with respect to the disk layout of the file system. Our insight is that LiveDFS follows Ext3FS and organizes blocks into block groups, each keeping the metadata of the blocks within the same group. In LiveDFS, each block group is allocated a *fingerprint store*, which is an array of pairs of fingerprints and reference counts, such that each array entry is indexed by the block number (i.e., block address) of the respective disk block in the same block group. Thus, each block group in the disk partition has its corresponding fingerprint store. Figure 3 shows how LiveDFS deploys a fingerprint store in a block group. We place the fingerprint store at the front of the data block region in each block group. When LiveDFS writes a new block, we write the content to the disk and update the corresponding fingerprint and reference count for the block. A key observation is that all updates are localized within the same block group, so the disk seek overhead is minimized.

To quantify the storage overhead of a fingerprint store, we consider the case where each fingerprint is a 16-byte MD5 hash and each reference count is of 4 bytes. Note

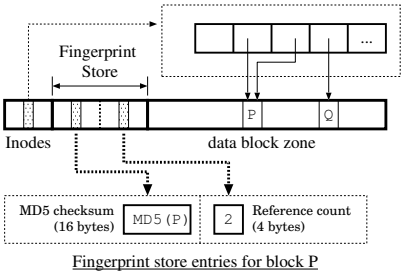


Fig. 3. Deployment of a fingerprint store in a block group.

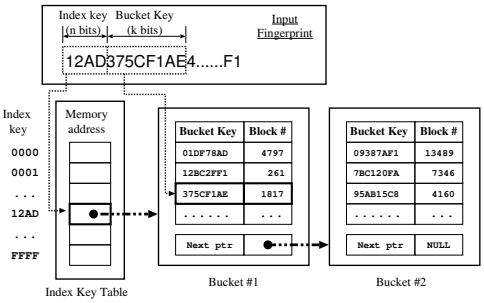


Fig. 4. Design of the fingerprint filter.

that in Ext3FS, the default block size is 4KB and default block group size is 128MB, so there are 32,768 blocks per block group. The fingerprint store will consume 655,360 bytes, or equivalently 160 blocks. This only accounts for 0.49% of the block group size. Hence, the storage overhead of a fingerprint store is limited.

Fingerprint filter. The *fingerprint filter* is an in-memory indexing structure that aims to speed up the search of fingerprints on disk. While there are many possible designs of indexing techniques (see Section 5), we explore a simple design that suffices for our requirements and is shown to work well according to our evaluation (see Section 4).

Figure 4 illustrates the design of the fingerprint filter. The fingerprint filter is a two-level filter. The first-level filter maps the first n prefix bits of a fingerprint called the *index key*, while the second-level filter maps the next k bits called the *bucket key*. We elaborate how n and k can be chosen in later discussion.

We initialize the fingerprint filter when LiveDFS is first mounted. The mounting procedure first allocates the *index key table*, which is an array of 2^n entries, each of which is a memory address that points to the head of a chain of buckets. Then LiveDFS will read through all fingerprint stores in different block groups, and retrieve the index key, the bucket key, and the block number associated with each disk block. We construct a *bucket entry* for each tuple (bucket key, block number) and store the entry in the correct bucket according to the index key. If a bucket is full of entries, then we create a new bucket, so a chain of buckets may be created. We sort the bucket entries in each bucket by the bucket keys, in order for us to efficiently search for a bucket entry using binary search. We emphasize that the initialization of the fingerprint filter is only a one-time process, which is performed when the file system is mounted. We evaluate the mount time using our experimental testbed (see Section 4). We find that the mount time is within 3 minutes for a 512GB harddisk partition that is fully filled with data. In general, the mount time varies linearly with the amount of data, and hence the number of fingerprints, being stored on disk.

We query the fingerprint filter whenever there is a new block to be written to the disk. If the fingerprint of the new block matches one of the "$n + k$"-bit prefixes stored in the filter, then the filter will return the corresponding block number. Note that there may be more than one fingerprint sharing the same "$n + k$" prefix bits, so the filter may return more than one block number. To eliminate false positives, LiveDFS will look up

the corresponding fingerprint store based on each returned block number, and verify if the new block to be written matches the full fingerprint in the fingerprint store. Note that LiveDFS updates the fingerprint filter every time a block is written, modified, or deleted.

Performance impact of parameters. We now elaborate how we set the parameters for the fingerprint filter, which depends on the index key length n and the bucket key length k. These parameters in turn determine the trade-offs of different metrics: (i) the total memory size of the filter, (ii) the false positive rate of each fingerprint filter lookup, and (iii) the bucket chain length. Instead of deriving the optimal choices of the parameters, we consider a special setting which performs well in practice based on our experiments (see Section 4).

In the following, we consider a 1TB partition with of block size 4KB (i.e., 2^{28} blocks in total). We employ the 16-byte MD5 hash algorithm to produce fingerprints. The system under our consideration adopts the 32-bit address space. We consider a special case where n = 19 and k = 24. Note that the following analysis can still apply to other parameter settings as well, e.g., 64-bit addressing.

We first evaluate the memory size of the fingerprint filter. For each data block, there is a 7-byte bucket entry, which stores a bucket key of k = 24 bits and a block number of 32 bits. Since there are a maximum of 2^{28} blocks, we need 1.792GB of memory for all bucket entries. Also, each index key entry is a 32-bit memory address that points to the head of a bucket chain, so we need $2^{19} \times 4$ = 2MB of memory for all index key entries. If we take into account internal fragmentation, then the fingerprint filter needs at most 2GB of memory, which is available for today's commodity configurations.

We emphasize that the memory footprint of LiveDFS is significantly less than those of ZFS [21] and SDFS [20], which assume 64GB and 8GB of memory per 1TB of data with block size 4KB, respectively. Although ZFS and SDFS use longer hashes (SHA-256 and Tiger, respectively) for fingerprints, the reason why they use significantly more memory is that they load the full fingerprints into memory. On the contrary, LiveDFS only loads the fingerprint prefixes into memory, and organizes the full fingerprints near their corresponding data blocks on disk so as to mitigate the disk seek overhead.

We next consider the false positive rate for each fingerprint filter lookup. Recall that there are 2^{28} data blocks. For a given fingerprint x, the probability ϵ that there exists another fingerprint (or data block) that has the same $n + k = 43$ prefix bits as x is: $1 - \left(1 - 2^{-43}\right)^{2^{28}-1} \approx 2^{-15}$. That is, on average every one out of 32,678 blocks will have a fingerprint mapped to more than one block number.

We also evaluate the bucket chain length. If the fingerprint value is uniformly distributed, then the average number of buckets associated with each index key is $2^{28}/2^{19}$ = 512 (for n = 19). Each bucket entry is of 7 bytes. If each bucket is stored as a memory page of size 4KB, then it can hold around 585 bucket entries, and the average bucket length is less than one bucket.

2.3 Prefetching of Fingerprint Store

Whenever LiveDFS writes a block, the fingerprint and the reference count of that block has to be updated. As a result, LiveDFS has to access the corresponding fingerprint store

on disk every time LiveDFS is writing a data block. We improve the performance of our deduplication design by extending the notion of spatial locality for caching fingerprints in memory. Our key observation is that a VM image file generally consists of a large stream of data blocks. If a data block is unique and cannot be deduplicated with existing blocks, then it will be written to the disk following the previously written block. Thus, the data blocks of the same block group are likely to be accessed at about the same time.

In order to further reduce the number of disk seeks, LiveDFS implements a *fingerprint prefetching mechanism*. When LiveDFS is about to access the fingerprint store of a block group, instead of accessing only the target block (which contains the target fingerprint), LiveDFS prefetches the entire fingerprint store of the corresponding block group and store it into the *page cache*, the disk cache of the Linux kernel. Therefore, subsequent writes in the same block group can directly update the fingerprint store in the page cache. The I/O scheduler of the Linux kernel will later flush the page cache into the disk. This further reduces the disk seeks involved. We point out that the consistency of the data content between the page cache and the disk is protected by the underlying journaling mechanism (see Section 2.4). Note that the idea is also used in [24, 28], except that we apply the idea in accordance with the Linux file system design.

The following calculations show that the overhead of our fingerprint prefetching mechanism is small. Suppose that LiveDFS uses MD5 hashes as the fingerprints of data blocks. Then a fingerprint store in a block group consumes 160 blocks (or 640KB space). Today an ordinary 7200-RPM hard disk typically has a data rate of 100MB/s, so it will consume only about 6-7ms for prefetching a whole fingerprint store into memory (i.e., the page cache). This time value is close to the average time of a random disk seek, which can be around 8-10ms [26].

2.4 Journaling

LiveDFS supports *journaling*, a feature that keeps track of file system transactions in a *journal* so that the file system can be recovered to a stable state when the machine fails, e.g., power outage. LiveDFS extends the journaling design in Ext3FS. In particular, we treat every write to a fingerprint store as the file system metadata and have the journal process modifications to the fingerprints and reference counts.

Figure 5 shows the pseudo-code of how LiveDFS updates a fingerprint store for a data block `P` to be written to the file system, while a similar set of procedures are taken when we delete a data block from the file system. Our goal is to update the fingerprint and the reference count associated with the block `P`. First, we obtain the handle that refers to the journal and perform the updates of metadata (e.g., indirect blocks, inodes) for `P` as in Ext3FS (Lines 1-2). If `P` is a new block that cannot be deduplicated with any existing disk block, then LiveDFS first loads the corresponding block that stores `P`'s fingerprint into memory (Line 4), and notify the journal that the fingerprint block is about to be updated via the function `ext3_journal_get_write_access()` (Line 5). After the fingerprint block is updated, we notify the journal that the modification is finished via the function `ext3_journal_dirty_metadata()` (Lines 6-7). Then, we update `P`'s reference count similarly (Lines 9-12). When LiveDFS releases the journal handle (Line 13), the journal will update the fingerprint store on disk *atomically*.

```
function LiveDFS Fingerprint Store Update Block
Input: data block P to be written to the file system
 1: handle = ext3_journal_start()
 2: Perform metadata writes via the journal handle
 3: if P cannot be deduplicated with an existing block then
 4:     Load a fingerprint block fp into memory
 5:     ext3_journal_get_write_access(handle, fp)
 6:     Update fp with the fingerprint of P
 7:     ext3_journal_dirty_metadata(handle, fp)
 8: end if
 9: Load P's reference count cp into memory
10: ext3_journal_get_write_access(handle, cp)
11: Increment the reference count cp by one
12: ext3_journal_dirty_metadata(handle, cp)
13: ext3_journal_stop(handle)
```

Fig. 5. Pseudo-code of how LiveDFS updates the fingerprint store through the journaling system.

Not only can the journal improve the file system reliability, but it can also enhance the write performance. The journal defers the disk updates of each write request and combines multiple disk writes in batch. This reduces the disk seeks and improve the write performance. We demonstrate this improvement in Section 4.

3 LiveDFS Implementation and Deployment

LiveDFS is a kernel-space file system running atop Linux. We implemented LiveDFS as a kernel driver module for the Linux kernel 2.6.32, and it can be loaded to the kernel *without* requiring any modification or recompilation of the kernel source code. The deduplication logic is implemented by extending the virtual file system (VFS) address space operations. In particular, we perform fingerprint computation and determine if a block can be deduplicated in the function `writepage()` (in Linux source tree: `fs/ext3/inode.c`), which is called by a kernel thread and will flush dirty pages to the disk.

Since LiveDFS is POSIX-compliant, it can be seamlessly integrated into an open-source cloud platform that runs atop Linux. In this work, we integrate LiveDFS into OpenStack [22], an open-source cloud platform backed by Rackspace and NASA, such that LiveDFS serves as a storage layer for hosting VM images with deduplication. We point out that Eucalyptus [18] has a similar architecture as OpenStack, so we expect that the deployment of LiveDFS in Eucalyptus follows a similar approach.

OpenStack overview. OpenStack is built on three sub-projects *Compute* (named *Nova*), *Object Storage* (named *Swift*), and *Image Service* (named *Glance*). Figure 6 shows a simplified view of an OpenStack cloud, which consists of Nova and Glance only. Nova defines an architecture that uses several controller services that coordinate the VM instances running on different Compute nodes. Glance is a VM image management system that is responsible for registering, searching, and retrieving VM images. It provides

APIs for accessing a storage backend, which could be Object Storage (Swift), Amazon S3, or a local server on which Glance is deployed. Note that OpenStack uses the `euca2ools` command-line tool provided by Eucalyptus to add and delete VM images.

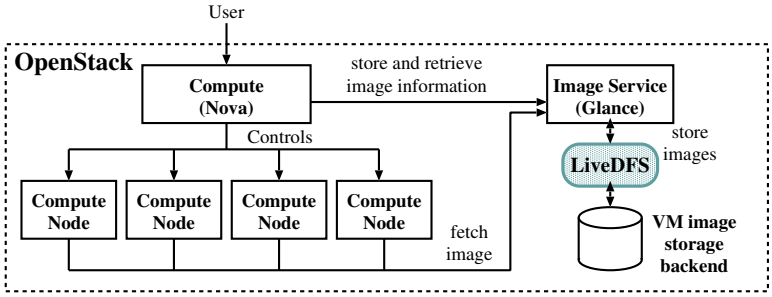


Fig. 6. LiveDFS deployment in an OpenStack cloud.

LiveDFS deployment. Figure 6 shows how LiveDFS is deployed in an OpenStack cloud. LiveDFS serves as a storage layer between Glance and the VM image storage backend. Administrators can upload VM images through Glance, and the images will be stored in the LiveDFS partition. When a user wants to start a VM instance, the cloud controller service of Nova will assign the VM instance to run on one of the Compute nodes based on the current resource usage. Then the assigned Compute node will fetch the VM image from Glance, which then retrieves the VM image via LiveDFS.

4 Experiments

In this section, we empirically evaluate our LiveDFS prototype. We first measure the I/O throughput performance of LiveDFS as a disk-based file system. We then evaluate the deployment of LiveDFS in OpenStack-based cloud platform. We justify the performance overhead of LiveDFS that we observe. We compare LiveDFS with Ext3FS, which does not support deduplication.

4.1 I/O Throughput

We measure the file system performance of different I/O operations using synthetic workload based on LiveDFS and Ext3FS. In LiveDFS, we assume that the index key length n is 19 bits and the bucket key length k is 24 bits (see Section 2.2). Note that LiveDFS is built on different design components, including (i) spatial locality, in which we allocate fingerprint stores in different block groups (see Section 2.2), (ii) prefetching of a fingerprint store (see Section 2.3), and (iii) journaling (see Section 2.4). We evaluate different LiveDFS variants that include different combinations of the design components, as shown in Table 1, to see the performance impact of each component. When spatial locality is disabled, we simply place all fingerprint stores at the end of the

disk partition; when prefetching is disabled, we bypass the step of prefetching a fingerprint store into the page cache; when journaling is disabled, we use alternative calls to directly write fingerprints and reference counts to the fingerprint stores on disk.

	Spatial locality	Prefetching	Journaling
LiveDFS-J	×	×	√
LiveDFS-S	√	×	×
LiveDFS-SJ	√	×	√
LiveDFS-all	√	√	√

Table 1. Different LiveDFS variants evaluated in Section 4.1 (√ = enabled, × = disabled).

Experimental testbed. Our experiments are conducted on a Dell Optiplex 980 machine with an Intel Core i5 760 CPU at 2.8GHz and 8GB DDR-III RAM. We equip the machine with two harddisks: a 1TB harddisk of Western Digital WD1002FAEX 7200RPM SATA for our benchmarking, and a 250GB harddisk for hosting our benchmarking tools and the operating system. Our operating system is Ubuntu 10.04.2 server 64-bit edition with Linux kernel 2.6.32.

Evaluation methodology. We use Linux basic system calls `read()` and `write()` to measure the I/O performance. Each experimental result is averaged over 10 runs. In each run, we use the system call `gettimeofday()` to obtain the duration of an operation, and then compute the throughput. At the beginning of each run, we clear the kernel page cache using the command `echo 3 > /proc/sys/vm/drop_caches` so that we can accurately evaluate the performance due to disk accesses.

Experiment A1: Sequential write. We first evaluate the sequential write performance of LiveDFS by writing a 16GB file with all unique blocks (i.e., all blocks cannot be deduplicated with others). The file size is larger than the 8GB RAM in our test machine, so that not all requests are kept in the kernel buffer cache.

Figure 7 shows the throughput of different LiveDFS variants and Ext3FS. First, considering LiveDFS-J and LiveDFS-SJ, we observe that LiveDFS-SJ has a higher throughput than LiveDFS-J by 16.1MB/s (or 37%). Thus, spatial locality by itself can improve the throughput, by putting the fingerprint stores close to their blocks on disk.

We note that LiveDFS-S (without journaling) has the lowest throughput among all variants. Specifically, LiveDFS-SJ increases the throughput of LiveDFS-S by 34.6MB/s (or 78.6%). Since journaling combines write requests and flushes them to the disk in batch (see Section 2.4), journaling can effectively minimize the disk accesses in addition to providing robustness against system crashes.

We observe that LiveDFS-all further improves the throughput of LiveDFS-SJ via prefetching (by 13.8MB/s, or 18%). Now, comparing LiveDFS-all and Ext3FS, we observe that LiveDFS-all is slightly less than Ext3FS's throughput by 3.7MB/s (or 3.8%), mainly due to the increase in the block group accesses. Because we introduce a fingerprint store to each block group, LiveDFS has fewer data blocks per block group than Ext3FS. However, we show that each design component of LiveDFS can reduce disk accesses and increase the throughput of LiveDFS close to that of Ext3FS.

Experiment A2: Sequential read. In this experiment, we evaluate the sequential read performance of LiveDFS by reading the stored 16GB file created in Experiment A1. Figure 8 shows the results. LiveDFS and Ext3FS. We observe that all LiveDFS variants

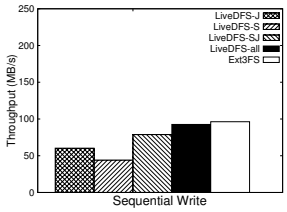


Fig. 7. Throughput of sequential write.

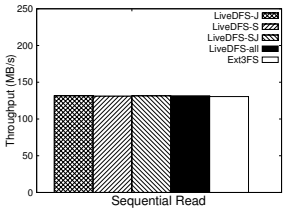


Fig. 8. Throughput of sequential read.

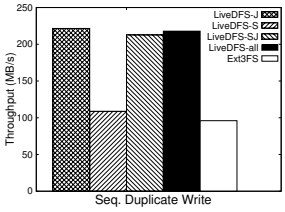


Fig. 9. Throughput of sequential duplicate write.

have almost the same throughput as Ext3FS, mainly because LiveDFS uses the same execution path as that of Ext3FS for reading data blocks.

Experiment A3: Sequential duplicated write. In this experiment, we write another 16GB file that has the identical content to the one in Experiment A1. Figure 9 shows the results. We observe that all LiveDFS variants (except LiveDFS-S without journaling) significantly boost the throughput of Ext3FS by around 120MB/s. The reason is that LiveDFS only needs to read the fingerprints of data blocks and update the reference counts, without re-writing the same data blocks (with larger size) to the disk. We note that the write-combining feature of journaling plays a significant role in boosting the throughput of LiveDFS, as LiveDFS-S does not achieve the same improvement.

Experiment A4: Crash recovery with journaling. In this experiment, we consider how deduplication affects the time needed for a file system to recover from a crash using journaling. We set the journal commit interval to be five seconds when the file system is first mounted. We then write a 16GB file with unique blocks sequentially into LiveDFS, and unplug the power cable in the middle of writing. The file system is therefore inconsistent. After rebooting the machine, we measure the time needed for our file system check tool `fsck`, which is modified from the Ext2FS utility `e2fsprogs` to recover the file system.

We observe that LiveDFS ensures the file system correctness using journaling. However, it generally uses a longer recovery time than Ext3FS. On average (over 10 runs), LiveDFS uses 14.94s to recover, while Ext3FS uses less than 1s. The main reason is that LiveDFS needs to ensure that the fingerprints in the fingerprint store match the new data blocks being written since the last journal commit interval. Such additional fingerprint checks introduce overhead. Since system crashes are infrequent, we expect that the recovery time is acceptable, as long as the file system correctness is preserved.

4.2 OpenStack Deployment

We now evaluate LiveDFS when it is deployed in an OpenStack-based cloud. Our goal is to justify the practicality of deploying LiveDFS in a real-life open-source cloud for VM image storage with deduplication. Specifically, we aim to confirm that LiveDFS achieves the expected storage savings as observed in prior studies [13, 11], while achieving reasonable I/O performance of accessing VM images.

System configuration. Our experiments are conducted in an OpenStack cloud platform consisting of three machines: a Nova cloud controller, a Glance server, and a Compute

node. The Nova cloud controller is equipped with an Intel Core 2 Duo E7400 2.8GHz CPU, while both the Glance server and the compute node are equipped with an Intel Core 2 Quad Q9400 2.66GHz CPU. All machines are equipped with 4GB DDR-II RAM, as well as two harddisks: a 1TB 7200RPM harddisk for storing VM images and a 250GB harddisk for hosting the operating system and required software. All machines use Ubuntu 10.04.2 server 64-bit edition as the operating system. Furthermore, all three machines are inter-connected by a Gigabit Ethernet switch, so we expect that the network transmission overhead has limited performance impact on our experiments.

All VM images are locally stored in the Glance server. We deploy either LiveDFS or Ext3FS within the Glance server, which can then access VM images via the deployed file system using standard Linux file system calls. We also deploy Kernel-based Virtual Machine (KVM) as the default hypervisor in the compute node. By default, we assume that LiveDFS enables all design components (i.e., spatial locality, prefetching, and journaling).

Our cloud testbed consists of only one Compute node, assuming that in most situations there is at most one Compute node that retrieves a VM image at a time. We also evaluate the scenario when a Compute node retrieves multiple VM images simultaneously (see Experiment B3).

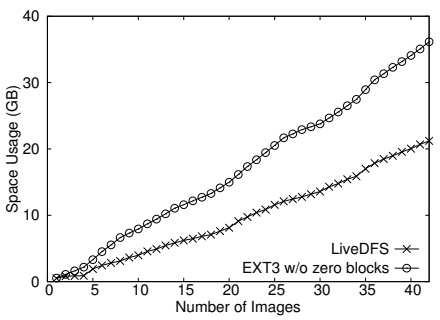
Dataset. We use deployable VM images to drive our experiments. We have created 42 VM images in Amazon Machine Image (AMI) format. Table 2 lists all the VM images. The operating systems of the VM images include ArchLinux, CentOS, Debian, Fedora, OpenSUSE, and Ubuntu. We prepare images of both x86 and x64 architectures for each distribution, using the recommended configuration for a basic server. Networked installation is chosen so as to ensure that all installed software packages are up-to-date. Each VM image is configured to have size 2GB. Finally, each VM image is created as a single monolithic flat file.

Distribution	Version (x86 & x64)	Total
ArchLinux	2009.08, 2010.05	4
CentOS	5.5, 5.6	4
Debian	5.0.8, 6.0.1	4
Fedora	11, 12, 13, 14	8
OpenSUSE	11.1, 11.2, 11.3, 11.4	8
Ubuntu	6.06, 8.04, 9.04, 9.10, 10.04, 10.10, 11.04	14

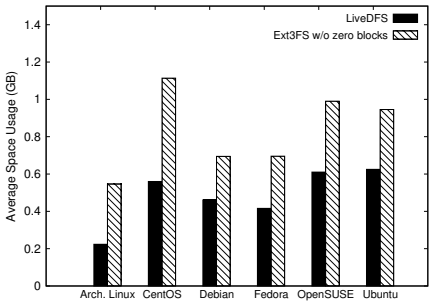
Table 2. The collection of virtual machine images involved in the experiments.

Experiment B1: Storage efficiency. We first validate that LiveDFS can save space for storing VM images. Note that each VM image typically has a large number of zero-filed blocks [11]. One main source of zero-filled blocks, according to our created VM images, is due to the unused space of the VM. To reflect the true saving achieved by deduplication, we exclude counting the zero-filled blocks in our evaluation.

Figure 10(a) shows the cumulative space usage of storing the VM images using LiveDFS and Ext3FS. Overall, LiveDFS saves at least 40% of space over Ext3FS (for non-zero-filled blocks). If we count the zero-filled blocks as well, then LiveDFS still



(a) Cumulative space usage



(b) Average space usage of a VM image for different distributions

Fig. 10. Space usage of VM images (excluding zero-filled blocks).

uses around 21GB of space (as all zero-filled blocks can be denoted by a single block), while Ext3FS consumes 84GB of space for the 42 2-GB VM images. In this case, we can even achieve 75% of saving.

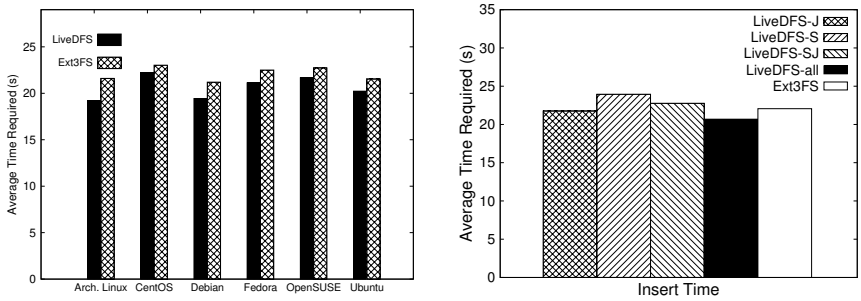
Figure 10(b) shows the average space usage of a VM image for each Linux distribution. The space savings range from 33% to 60%. It shows that different versions of VM images of the same Linux distribution have a high proportion of identical data blocks that can be deduplicated. Therefore, our LiveDFS implementation conforms to the observations that deduplication is effective in improving the storage efficiency of VM images [13, 11]. We do not further investigate the deduplication effectiveness of VM images, which has been well studied in [13, 11].

Experiment B2: Time for inserting VM images. In this experiment, we evaluate the time needed for inserting VM images into our Glance server. We execute the commands `euca-bundle-image`, `euca-upload-bundle`, and `euca-register`[1] to insert the VM images from the cloud controller to the Glance server (over the Gigabit Ethernet switch). We repeat the test five times and obtain the average. Our goal is to measure the practical performance of writing VM images using LiveDFS.

Figure 11(a) shows the average insert time for individual distributions (over five runs). Overall, LiveDFS consumes less time than Ext3FS in inserting VM images. The reason is that LiveDFS does not write the blocks that can be deduplicated to the disk, but instead it only updates the smaller-size reference counts. Figure 11(b) shows the average insert time for all 42 VM images using different LiveDFS variants as defined in Section 4.1. Although this time the differences among the different LiveDFS variants are not as significant as seen in Section 4.1, we observe that enabling all design components (i.e., LiveDFS-all) still gives the least insert time.

Experiment B3: Time for VM startup. We now evaluate the time needed to launch a single or multiple VM instances. We assume that all VM images have been inserted into the file system. We then execute the `euca-run-instances` command in the Nova cloud controller to start a VM instance in the Compute node, which fetches the corre-

[1] Those `euca-*` commands come with `euca2ools`, the command-line tool of Eucalyptus for VM image management.



(a) Individual distribution (b) Different LiveDFS variants

Fig. 11. Average time required for inserting a VM image to the Glance server.

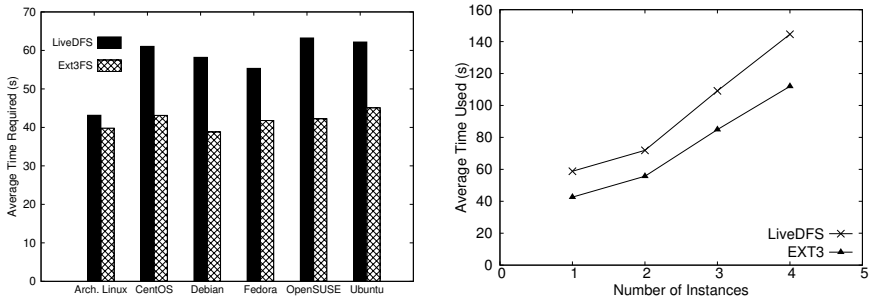
sponding VM image from the Glance server. We measure the time from the command being issued until the time when the `euca-run-instances` command invokes the KVM hypervisor (i.e., when the VM starts running). Our goal is to measure the practical performance of reading VM images using LiveDFS.

Figure 12(a) shows the time needed to launch a single VM instance for different distributions in the Compute node. We observe that LiveDFS has lower throughput performance than Ext3FS. The main reason is that deduplication introduces *fragmentation* [24]. That is, in deduplication, some blocks of a file may be deduplicated with the blocks of a different file, so the actual block allocation of a file on disk is no longer in the sequential order as seen in the ordinary file system without deduplication. Fragmentation is an inherent problem in deduplication, as it remains an open issue to be solved [24]. To see how fragmentation affects the practical performance, we note that in LiveDFS, its increase in VM startup time ranges from 3s to 21s (or 8% to 50%) for a VM image of size 2GB. Currently, the cloud nodes are connected over a Gigabit Ethernet switch. We expect that the VM startup penalty will become less significant if we deploy the cloud in a network with less available bandwidth (e.g., the cloud nodes are connected by multiple switches that are shared by many users), as the network transmission overhead will dominate in such a setting.

We now consider the case when we launch multiple VM instances in parallel. In the Compute node, we issue multiple VM startup commands simultaneously, and measure the time for all VM instances to start running. Figure 12(b) shows the time required for starting one to four VM instances in the Compute node. The overhead of LiveDFS remains consistent (at around 30%), regardless of the number of VM instances being launched. The observation conforms to that of launching a single VM instance.

5 Related Work

Existing deduplication techniques are mainly designed for backup systems. On the other hand, LiveDFS applies deduplication in a different design space and is designed for VM image storage in a cloud platform. Also, LiveDFS seeks to be POSIX-compliant, so its implementation details take into account the Linux file system layout and are different



(a) Startup time for a VM instance for different distributions.

(b) Startup time for multiple VM instances using the Ubuntu distribution.

Fig. 12. Average time for VM startup.

from existing deduplication systems. In this section, we review the related work in deduplication in more detail.

Backup systems. Venti [23] is the first work of content addressable storage (CAS) for data backup. Foundation [24] is another CAS system built upon Venti, and improves Venti with the new compare-by-value mode. It uses the Bloom filter as an in-memory indexing structure to identify new or existing fingerprints. LiveDFS uses the fingerprint filter as an in-memory indexing structure. However, the fingerprint filter not only identifies the existence of fingerprints as in the Bloom filter, but it also specifies where the fingerprint is stored on disk. Data Domain [28] also uses the Bloom filter [4] for in-memory indexing, and uses locality preserved caching to reduce random I/Os for duplicated data. Sparse Indexing [14] trades deduplication opportunities for reduced memory usage in backup systems by only deduplicating data with a few of the most similar previous copies in different backup streams. Bimodal Content Defined Chunking [12] reduces the memory overhead of chunk indexing by using chunks with different granularities in different regions of a backup. Note that the above systems do not consider the scenario where data can be modified or deleted, while LiveDFS addresses this by associating a reference count with each data block in its deduplication design.

Usage of flash memory. Dedupv1 [15] and ChunkStash [8] use flash memory and solid state drives (SSDs) to relieve the memory constraints of deduplication. They exploit the fast random I/O feature of flash memory by putting the fingerprints into the flash rather than in main memory. They show that they achieve competent I/O performance while requiring a small memory capacity. On the other hand, LiveDFS targets a commodity server that is not necessarily equipped with SSDs.

Scalable storage. Extreme Binning [3], HydraFS [27] and DeDe [7] are scalable storage systems that support data deduplication. Extreme Binning exploits *file similarity* rather than chunk locality. HydraFS is a file-system built atop HYDRAstor [9]. DeDe is a storage system that performs *out-of-order* deduplication. Their deployment platforms are based on a distributed environment, while LiveDFS is designed to be deployed on a single commodity server.

Deduplication-enabled file systems. ZFS by Sun Microsystems[21] and SDFS by Opendedup [20] are file systems supporting inline deduplication. However, as stated

in Section 2.2, they need a large memory capacity for enabling deduplication, mainly because they assume that the entire set of fingerprints is loaded into memory. In terms of deployment, ZFS and SDFS are mainly designed for enterprise-scale systems, while LiveDFS is designed as a kernel-space file system for commodity servers. Btrfs [5] is an open-source file system developed by Oracle that supports deduplication. While Btrfs is a Linux kernel module like LiveDFS, it only supports offline deduplication [19] instead of inline at the time of this writing.

VM image storage. It has been shown [13, 11] that deduplication can significantly save the storage space for VM images. However, there remain open issues of deploying deduplication in a VM storage system. Nath *et al.* [17] evaluate a deployed CAS system for storing VM images. They mainly focus on storage efficiency and network load, but do not evaluate the read/write throughput of accessing VM images. Liguori *et al.* [13] deploy a CAS system based on Venti [23] for storing VM images, but its read/write throughput is limited by the Venti implementation. Lithium [10] is a cloud-based VM image storage system that aims for fault tolerance, but it does not consider deduplication. To our knowledge, LiveDFS is the first practical system that deploys deduplication for VM image storage in a real cloud platform.

6 Conclusions and Future Work

We propose LiveDFS, a live deduplication file-system that is designed for VM image storage in an open-source cloud with commodity configurations. LiveDFS respects the file system design layout in Linux and allows general I/O operations such as read, write, modify, and delete, while enabling inline deduplication. To support inline deduplication, LiveDFS exploits spatial locality to reduce the disk access overhead for looking up fingerprints that are stored on disk. It also supports journaling for crash recovery. LiveDFS is implemented as a Linux kernel driver module that can be deployed without the need of modifying the kernel source. We integrate LiveDFS into a cloud platform based on OpenStack and evaluate the deployment. We show that LiveDFS saves at least 40% of storage space for different distributions of VM images, while its performance overhead in read/write throughput is minimal overall. Our work demonstrates the feasibility of deploying deduplication into VM image storage in an open-source cloud.

In this work, we mainly focus on deduplication on a single storage partition. Since a cloud platform is typically a distributed system, we plan to extend LiveDFS in a distributed setting (e.g., see [16]). One challenging issue is to balance the trade-off between storage efficiency and fault tolerance. On one hand, deduplication reduces the storage space by removing redundant data copies; on the other hand, it sacrifices fault tolerance with the elimination of redundancy. We pose this issue as future work.

The source code of LiveDFS is published for academic use at

http://ansrlab.cse.cuhk.edu.hk/software/livedfs.

Acknowledgments

This work was supported in part by grant ITS/297/09 from the Innovation and Technology Fund of the HKSAR.

References

1. Amazon EC2. http://aws.amazon.com/ec2.
2. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Comm. of the ACM*, 53(4):50–58, Apr 2010.
3. D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. IEEE MASCOTS*, pages 1–9. IEEE, 2009.
4. B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.
5. Btrfs. http://btrfs.wiki.kernel.org.
6. M. Cao, T. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proc. of the Ottawa Linux Symposium (OLS)*, 2005.
7. A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proc. USENIX ATC*, 2009.
8. B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proc. USENIX ATC*, 2010.
9. C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proc. USENIX FAST*, 2009.
10. J. G. Hansen and E. Jul. Lithium: Virtual Machine Storage for the Cloud. In *Proc. of ACM SOCC*, 2010.
11. K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. ACM SYSTOR*, 2009.
12. E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proc. USENIX FAST*, page 18. USENIX Association, 2010.
13. A. Liguori and E. Van Hensbergen. Experiences with Content Addressable Storage and Virtual Disks. In *WIOV'08*, 2008.
14. M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. USENIX FAST*, 2009.
15. D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proc. IEEE MSST*, 2010.
16. A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of ACM SOSP*, 2001.
17. P. Nath, M. A. Kozuch, D. R. 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 *Proc. USENIX ATC*, 2006.
18. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud Computing System. In *Proc. of IEEE CCGrid*, 2009.
19. Offline Deduplication for Btrfs. http://www.spinics.net/lists/linux-btrfs/msg07818.html.
20. OpenDedup. NBU for vSphere. http://code.google.com/p/opendedup/downloads/detail?name=SDFS%20Architecture.pdf, December 2010.
21. OpenSolaris. ZFS Dedup FAQ (Community Group zfs.dedup) - XWiki. http://hub.opensolaris.org/bin/view/Community+Group+zfs/dedup, December 2010.

22. OpenStack. http://www.openstack.org.
23. S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, 2002.
24. S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. USENIX ATC*, 2008.
25. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. of ACM CCS*, 2009.
26. Seagate. 7200-RPM Drive Specification Comparison. http://www.seagate.com/docs/pdf/whitepaper/mb578_7200_drive_specification_comparison.pdf.
27. C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *Proc. of USENIX FAST*, 2010.
28. B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. USENIX FAST*, 2008.