Selfie: Co-locating Metadata and Data to Enable Fast Virtual Block Devices

Xingbo Wu

Wayne State University

wuxb@wayne.edu

Zili Shao

The Hong Kong Polytechnic University

cszlshao@comp.polyu.edu.hk

Song Jiang

Wayne State University

sjiang@wayne.edu

Abstract

Virtual block devices are widely used to provide block interface to virtual machines (VMs). A virtual block device manages an indirection mapping from the virtual address space presented to a VM, to a storage image hosted on file system or storage volume. This indirection is recorded as metadata on the image, also known as a lookup table, which needs to be immediately updated upon each space allocation on the image for data safety (also known as image growth). This growth is common as VM templates for large-scale deployments and snapshots for fast migration of VMs are heavily used. Though each table update involves only a few bytes of data, it demands a random write of an entire block. Furthermore, data consistency demands correct order of metadata and data writes be enforced, usually by inserting the FLUSH command between them. These metadata operations compromise virtual device's efficiency.

In this paper we introduce Selfie, a virtual disk format, that eliminates frequent metadata writes by embedding metadata into data blocks and makes write of a data block and its associated metadata be completed in one atomic block operation. This is made possible by opportunistically compressing data in a block to make room for the metadata. Experiments show that Selfie gains as much as 5x performance improvements over existing mainstream virtual disks. It delivers near-raw performance with an impressive scalability for concurrent I/O workloads.

Categories and Subject Descriptors   D.4.2 [*OPERATING SYSTEMS*]: Storage Management—Storage hierarchies

1.   Introduction

Virtualization has been an imperative and widely used technique to support isolation and consolidation, and to improve resource and power efficiency in various computing platforms, from cloud systems to personal/mobile computing devices. A virtual storage device, such as a virtual disk, is one of the most essential components in a virtual machine (VM) and its efficiency is critical to the VM's performance, especially for I/O-intensive applications. A virtual disk, emulating functionalities available in a physical block device, is hosted as a disk image on another storage layer, which can be any file or block abstraction, such as a regular file, a disk partition, or a logical volume (QEMU 2015; oVirt 2014).

To enable virtual disk and support such much desired features as thin provisioning, COW (Copy-on-Write), and snapshots, various disk image formats have been proposed and deployed, including QCOW2 for QEMU/KVM (McLoughlin 2008), VirtualBox's VDI (Ellison 2008), VMware's VMDK (VMWare 2007), and Microsoft's VHD (Microsort 2012). Unlike real block device, in addition to storing blocks of data these images also need to record several types of metadata and keep them up to date. The metadata include lookup table, bitmap, and reference counts to enable space allocation and address mapping. For data persistency on newly allocated space on a virtual disk, not only data blocks but also the metadata associated with the blocks must be written onto the disk. Because of frequently taking snapshots and using COW in the VM environment, such writes are common (Lagar-Cavilla et al. 2009). Unfortunately, the extra metadata writes can make virtual disks much slower than physical disks for several reasons.

First, metadata is usually only a couple of bytes, much smaller than a data block (usually 4 KB). Because a virtual disk is still a block device, persisting a few bytes of metadata demands writing an entire block, resulting in a very high write amplification.

Second, metadata are usually clustered together and are separated from the data. When a hard disk is used, the two writes (of metadata and data) for serving one WRITE command to a virtual device incurs disk head seeks between

them, and multiple WRITE commands for writing sequential data blocks lead to random disk writes, potentially slowing down the disk throughput by several folds.

Third, it is often required that writes of metadata and data for a WRITE command are ordered for data consistency on the virtual disk. For example, writing data block(s) must precede writing into lookup table so that unintended data would never be returned upon a READ command. To enforce the ordering, a FLUSH command has to be issued between the writes. With frequent issuance of FLUSH, there would be little room for on-disk schedulers such as NCQ (Native Command Queuing) to optimize the order of disk accesses, leading to much degraded disk efficiency. The performance becomes even worse with multiple concurrent I/O streams.

Last but not least, synchronous writes are common. Application developers and system designers are decreasingly willing to leave their data in the volatile buffer and to take the risk of losing data. For example, in a workload study with Apple software iWork and iLife suites, it is reported that "*...half the tasks synchronize close to 100% of their written data while approximately two-thirds synchronize more than 60%*" (Harter et al. 2011). It is also concluded that "*For file systems and storage, the days of delayed writes (C.Mogul 1994) may be over; new ideas are needed to support applications that desire durability.*" (Harter et al. 2011). Synchronous writes demand immediate persistence of metadata, making the system buffer incapable.

There have been efforts to remove or ameliorate the performance barrier, including QED (QEMU Enhanced Disk format) (Hajnoczi 2010) for removing the ordering requirement when the virtual disk is hosted on a file system, and FVD (Fast Virtual Disk) (Tang 2011) for potentially reducing frequency of updating metadata by adopting a larger allocation and address mapping unit. However, they have not addressed the root cause of the issue, which is metadata updates attached to data block writing, and their effectiveness is often conditional and limited (see Sections 3 and 4 for analysis and comparisons).

In this paper, we aim to fundamentally address the issue by removing the necessity of having separate metadata write(s). This essentially leaves almost only writing of data blocks on the critical path of regular virtual disk operations and allows a virtual disk to behave much like a real disk. While immediately persisting metadata at the time of data write cannot be avoided for a successful service of a synchronous WRITE command, we are challenged with the issue of where to efficiently write the metadata. To this end, we propose to store metadata in their corresponding data block(s). By doing this, the metadata write as well as its induced overhead with write amplification and disk head seeks can be completely eliminated. Furthermore, when a piece of metadata is embedded in a data block, the aforementioned ordering problem is also removed as a block's write can be considered as an atomic operation in a block device. In the meantime, integrity of data in a WRITE command cannot be compromised, and a data block is of fixed size (usually 4 KB) and cannot be expanded to accommodate the metadata. To this end we propose to compress data in the block to make room for the metadata, and believe that data in performance-sensitive scenario is usually compressible for our purpose.

First, we need only 10 bytes in a 4 KB block, or 0.25% of a block, to accommodate metadata associated with the block. As long as the data is compressible, the compression ratio would be almost certainly much larger than what is needed. Second, there are many scenarios in which data have not been compressed when it is written to the disk, or data is still compressible at the (virtual) disk. One scenario is that data being actively processed is usually not compressed on the disk. Examples include compiling the Linux kernel and frequently updated data structures such as file system metadata, on-disk B-tree, database index, and log or journal data. Another scenario is that individual data items may have been compressed but blocks filled with multiple such items are often still compressible. Examples include compressed values in individual key-value pairs packed into an SSTable (Chang et al. 2008), and individually compressed memory pages swapped out to the disk as blocks in the virtual memory management (Jennings 2013). In both examples, data items are easier to be located and retrieved if they are individually compressed. Third, users are often less sensitive to the performance of writing large volume of data that is incompressible, such as movie files or archival data which have been compressed as a whole, as the operation often happens in the background. Furthermore, WRITE commands for very large data allow batched metadata writes. Even though metadata writes cannot be removed, the batched writes have been reasonably efficient and their performance is less of concern.

By exploiting opportunity of metadata embedding, we propose a virtual disk image format and its corresponding block device driver, collectively named as *Selfie*, implying the fact that co-locating data and its associated metadata makes the block self-descriptive. In summary, we make the following three contributions in this paper:

- We proposed to embed metadata into data blocks for their temporary storage to enable their almost-zero-cost persistence without requiring any hardware support or changes of device access protocol.

- We applied the idea in the optimization of a virtual disk's operations. We designed Selfie that opportunistically compresses data in a block to make room for storing its associated metadata. By doing this, Selfie has great potential to significantly improve performance without compromising features offered by existing virtual disks.

- We have implemented a Selfie prototype in QEMU and extensively evaluated it against widely used QCOW2 virtual disk image and recently proposed ones such as FVD
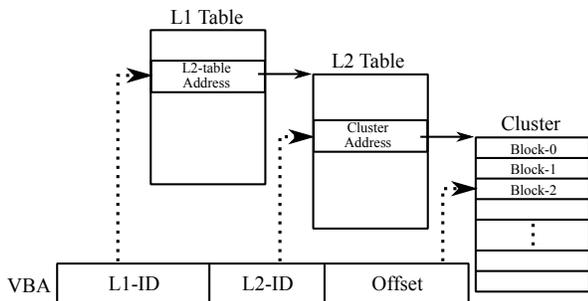
Figure 1: *In-memory lookup table in QCOW2.*

and QED with representative benchmarks. The measurements show that Selfie improves performance, in terms of I/O throughput or execution time, by up to five times.

2. The Design of Selfie

To enable a virtual disk, Selfie provides a disk image of its defined format for storing data and metadata and conducts block address translation to map a virtual disk space to the data space designated in the image. It has an in-memory data structure to support normal address translation and an on-image data structure for re-constructing the in-memory one after a failure.

2.1 Maintaining In-memory Lookup Table

Similar to other virtual disk drivers, such as QCOW2, Selfie maintains an in-memory lookup tree (or table) to translate a virtual block address (VBA), which is in the address space presented to the virtual machine as a block abstraction, to an image block address (IBA), which is actually also a logic block address in the disk image. The disk image can be hosted on a file in the host machine's file system or on a logical block volume. For time- and space-efficient translation from VBA to IBA, the lookup table consists of two levels of sub-tables, namely L1 and L2 tables, as shown in Figure 1. There is only one L1 table, and each entry of the L1 table links to an L2 table. Each entry of an L2 table points to a *cluster*, which is the unit for space allocation on an image and contains multiple blocks (e.g., a 64KB cluster consists of 16 4KB-blocks). Accordingly a VBA consists of three components: L1-ID, L2-ID, and offset. L1-ID and L2-ID indicate specific entries in the corresponding L1 and L2 tables, respectively, and the offset specifies the block in the corresponding cluster, or the IBA a VBA is mapped to. The lookup table is essential for accessing data on a virtual disk. For every write of data whose mapping from VBA to IBA has not been established in the tree, or the corresponding cluster has not been allocated on the image, the disk driver must allocate the cluster and record the mapping to the cluster in the tree. However, it is not sufficient to only keep the mapping information in the volatile memory, as its loss, possibly due to power failure, makes the data inaccessible. In serving synchronous writes, a virtual disk must make the mapping information be immediately and completely persistent on the image, or synchronously maintain an on-image lookup table, as existing virtual disks, such as QCOW2, do.

A unique aspect of Selfie's design is how the mapping information is persisted without synchronously maintaining the on-image lookup table. The idea is to embed the information into the data blocks. To effectively apply the idea, there are two challenges to address during a virtual disk's recovery from a failure. The first one is how to tell the blocks that have been compressed and contain the mapping information, which are named as **Z-block**s, from other blocks that do not contain the information due to their data's incompressibility, which are named as **N-block**s. The second one is how to minimize recovery time with the mapping information distributed across many data blocks.

2.2 Zoning On-image Space

Due to possible co-existence of Z-blocks and N-blocks on a Selfie's image, the two types of blocks cannot be distinguished by themselves. The blocks need to separated into different areas so the type of a block can be be identified from the area's type. Therefore, in addition to the mapping information, another kind of metadata is area type.

A convenient form of area could be cluster, the image allocation unit. That is, clusters can be dedicated to storing either Z-blocks (**Z-clusters**) or N-blocks (**N-clusters**). Actually a lookup table is to map a virtual cluster address, a concatenation of L1-ID and L2-ID as shown in Figure 1, to an on-image cluster address. Using a larger cluster, rather than disk access unit (block), as the unit for space allocation and address tracking is a common practice on various virtual disks, including QCOW2, to retain spatial locality in the block accesses. If we use cluster-sized area, both kinds of metadata are recorded and tracked at the unit of cluster and both need to be made persistent with a cluster allocation in response to a block write. Though the mapping information can be embedded in Z-blocks, the area type has to be recorded outside of data blocks. With such a design, Selfie would fail to improve the disk's efficiency at all.

To this end, Selfie adopts larger area unit, called zone. As an example, in our prototype implementation, the block size is 4KB, cluster size is 64KB (the same as that of QCOW2's default cluster size), and the zone size is 64MB. Without losing generality we assume these size numbers in the following description. All clusters in a zone are of the same type, either Z-cluster or N-cluster. Accordingly the zone is named as **Z-zone** or **N-zone**, respectively. The Selfie image format is illustrated in Figure 2. The zone type is recorded at a statically allocated space, called *zone bitmap*, in the image. In the bitmap, each zone's type is recorded. An allocated zone can be either an N-zone or a Z-zone. For a Z-zone, mapping information embedded in it may have been committed to the lookup table on the image and it is a committed Z-zone. If not yet, it is an uncommitted Z-zone. The zone bitmap is rarely updated as allocation of new zones is much less fre-
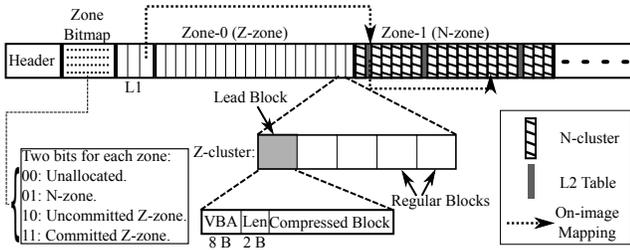
Figure 2: *Selfie's image format.*

(a) *Allocate-write into blocks of a cluster, including lead block.*

(b) *Allocate-write into blocks of a cluster, excluding lead block.*

(c) *Overwrite into blocks of a cluster, including lead block.*

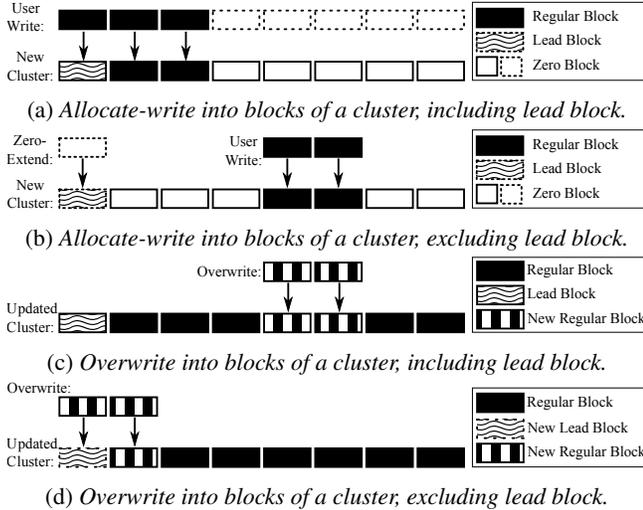(d) *Overwrite into blocks of a cluster, excluding lead block.*

Figure 3: *Scenarios of writes on Z-zone.*

quent than write requests and cluster allocations. The basic metadata of an image is stored at the beginning of the image, including the image size and unit size of clusters and zones.

2.3 Writing Blocks into Zones

An image grows with data writes to unallocated addresses. This growth can be allocation of new clusters in a currently non-full zone, which can be either Z-zone or N-zone depending on the data's compressibility. If such a non-full zone doesn't exist, the image grows by allocating a new zone. By checking in-memory lookup table Selfie knows whether the corresponding cluster has been allocated. If not yet, a cluster is allocated to accommodate the write. A Z-cluster or an N-cluster will be allocated depending on whether address mapping information (the metadata) can be embedded into the cluster.

Because the mapping is created at the cluster level, one cluster only needs to record its corresponding virtual cluster address (the L1-ID and L2-ID concatenation shown in Figure 1), which is 8 bytes long, as the mapping information. Selfie designates the first block (the lead block) in a Z-cluster to store the information at its beginning, followed by two bytes recording length of the compressed data in the block used for decompression, as shown in Figure 2. The following blocks in the Z-cluster store data without being

compressed just like regular blocks. Therefore, for a WRITE command writing one or a sequence of blocks into a cluster, Selfie only needs to determine whether the data to be stored in the cluster's lead block can be compressed to a size at least 10 bytes smaller. If this is true (see Figure 3a) or the WRITE command does not write to the lead block (see Figure 3b), a Z-cluster is allocated in the Z-zone for receiving the data and the mapping information about the cluster is embedded in the lead block. Note that a common practice in virtual disks, such as QCOW2, QED, and FVD, is to make every block in a cluster be written right after the cluster is allocated, either by the data in the WRITE command or by data from the backing image where this image is derived (if the backing image does not exist or cover the address space, the block is reset with 0.). This is to make sure that future READ commands do not return unexpected data.

If the WRITE command writes to non-lead blocks in an allocated Z-cluster, Selfie simply stores the data into the blocks without any additional operations (see Figure 3c). If there are data to be written on the lead block, Selfie checks the compressibility of the block of data (see Figure 3d). If they can be compressed to a size 10 bytes smaller than the block size, the compressed data are written into the lead block probably with its "Len" field in Figure 2 updated. Otherwise, the entire cluster of data is considered as incompressible and an N-cluster in a non-full N-zone will be allocated to hold the data. The relocation invalidates the space in the Z-zone and future Z-zone allocations can reuse this space. After a failure Selfie will not be confused on where the cluster is stored, as the cluster in the N-zone is immediately linked in the on-image lookup table. There is no need to relocate an N-cluster back into a Z-zone.

If a WRITE command writes to an unallocated cluster and its lead block cannot be compressed, an N-cluster is allocated in the N-zone. Just like Zone 1 shown in Figure 2, an N-zone may contain both N-clusters and L2-table blocks. When a new cluster is allocated, the on-image lookup is updated, similarly as existing virtual disks do. If an L2-table block, which is the counterpart of an in-memory L2 table, exists on the image to cover virtual address of the new cluster, the L2-table block is updated to record this cluster's address. If such an L2-table block does not exist, it is allocated to record the cluster's address. The L1 table is updated to record the address of the new L2-table block. In QCOW2 a particular order for these two updates has to be enforced (updating L1 table after updating L2 table) to ensure the L1 table does not point to an uninitialized L2 table. Otherwise, after a failure between the two updates, a READ command may access and return data in a random data block. As a performance optimization, Selfie does not enforce this order by removing a FLUSH command between them. This does not violate the image's correctness because Selfie always clears a zone with zeros immediately after it is allocated. In this way, even when L2 table is not yet updated to
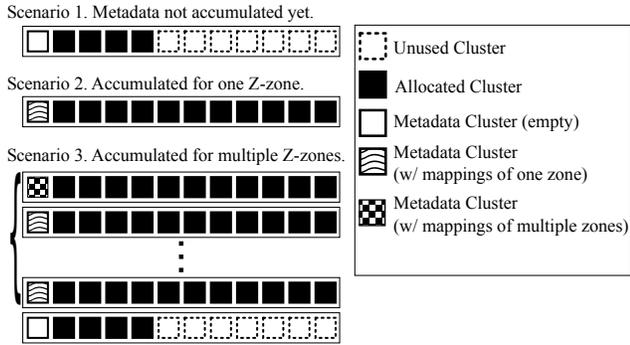
Figure 4: *Accumulating mapping info into metadata clusters.*

point to the right cluster, it does not point to any meaningful cluster. This zone clearing operation is also necessary for Z-zones, as it allows identification of unallocated clusters. To minimize possible impact of the zone clearing operation on front-end application, Selfie maintains a background thread to pre-initialize zones ahead of use during detected I/O-quite periods.

2.4 Quickly Recovering from a Failure

Like other virtual disk images, Selfie also maintains an on-image lookup table consisting of one statically allocated L1 table and multiple L2 tables dynamically allocated in the N-zones. The difference is that Selfie only synchronously updates the on-image table for N-clusters. Information on the updates about Z-clusters is initially stored within the Z-clusters and actual execution of the updates is delayed possibly until recovery upon a failure.

To recover from a failure, Selfie scans all Z-clusters in the Z-zones to collect embedded mapping information, or virtual cluster addresses, and commit them to the in-memory lookup table and to the on-image lookup table. The Z-zones can then be marked as committed in the zone bitmap, as shown in Figure 2. However, with a large image such a commitment may take a long time.

To address this issue, Selfie incrementally collects mapping information dispersed over many clusters and places it into much fewer dedicated clusters reserved for this purpose, named as *metadata clusters*. As shown in Figure 4, the first cluster in a Z-zone is reserved as a metadata cluster, which is empty when the zone is not full. When it is full, the virtual cluster addresses embedded in all the Z-clusters in the zone are collectively written to the metadata cluster. Note that Selfie does not need to read the addresses from the image, as they are also available in the memory. To further gather the mapping information together, for every eight consecutive Z-zones Selfie writes their mapping information into the metadata cluster of the first of the eight zones, as shown in Figure 4. A 64KB metadata cluster has the exact space to hold the eight zones' mapping information. Because each metadata zone is cleared to be all zeros when a

Z-zone is allocated and zones are allocated contiguously and never de-allocated, Selfie can unambiguously know in a specific metadata cluster which zone(s)' mapping information are stored and accordingly which Z-clusters can be skipped during a recovery. In this way, Selfie can reach all mapping information much faster and quickly complete the recovery. Also, if Selfie detects a long quiet period, it can opportunistically commit mapping information with minimal impact on front-end applications.

3. Performance Evaluation

To evaluate Selfie's performance, we implemented a prototype to reveal insights of its performance behaviors. Selfie is implemented on QEMU as a block device driver. In particular, it implements the QEMU's co-routine driver API, including `bdrv_co_readv()` and `bdrv_co_writev()`. QEMU is configured to use Linux native asynchronous I/O. In the Selfie's prototype, we choose a block size of 4 KB, a cluster size of 64 KB, a zone size of 64 MB, and an L2-table block (in N-zones) of 4 KB. We use LZ4 as the compression algorithm (Collet 2015). In Section 3.1 all data are assumed to be compressible. Section 3.2 examines the impact of data compressibility.

In the evaluation we compare Selfie with QCOW2, FVD, and QED. QCOW2 is one of the most popular virtual drivers. As mentioned, its image format is much like that of Selfie that has only N-zones. Both incrementally grow their images with allocation of new clusters. FVD adopts a large cluster (1 MB) so that it allows only one level of lookup table and reduces frequency of table updating. However, for small and non-sequential writes such a large allocation unit can lead to storage wastage[1]. To address this issue, FVD has to rely on the host file or storage system to support sparse image, where allocation of physical storage space happens upon data writing, instead of upon cluster allocation. By doing this, FVD has to introduce new metadata, or a bitmap for tracking which blocks in a cluster contain data, and the bitmap often needs to be made persistent upon data writes. QED is an enhanced QCOW2 driver. By assuming hosting file system would return zeros (for holes in a sparse file) or report error (for space beyond the end of file) when a not-yet-updated L2 table is used for reading data, QED allows out-of-order updates of data and metadata (L1 and L2 tables). Both QCOW2 and QED use 64 KB cluster size. To help reveal potential overhead with a virtual disk, we set up a new comparison target dubbed "**HOST**", where a real disk on the hosting server replaces virtual disk.

In the experiments both host and guest systems use the Linux 3.16.2 kernel. The guest system runs on top of QEMU 2.1.0 with KVM enabled and uses *virtio* to access virtual

---

[1] Selfie does not have the storage wastage issue with its use of large zones because it is at the cluster level, rather than at the zone level, that address mapping information is maintained. So a zone can accommodate data whose virtual addresses are non-contiguous.
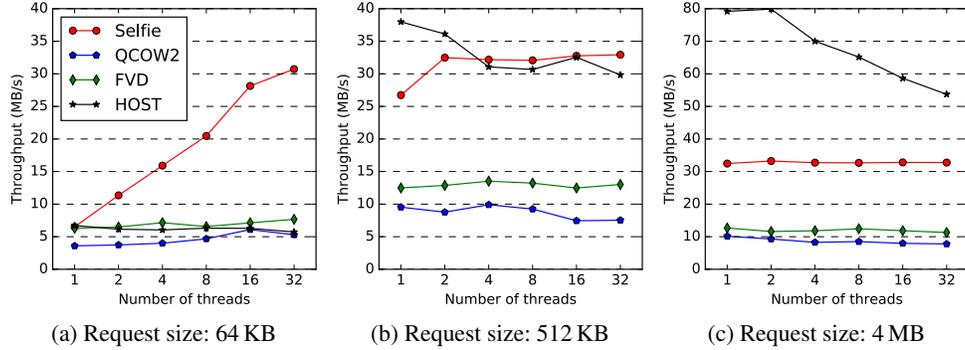
(a) Request size: 64 KB   (b) Request size: 512 KB   (c) Request size: 4 MB

Figure 5: *Throughput of* `fio` *with the virtual image hosed on the logical volume and each thread making* **sequential** *accesses.*

disk. The host system uses CFQ as its I/O scheduler. The system is a Dell CS23-SH server with two Intel Xeon L5410 4-core processors, 64GB DDR2-ECC memory, and two Western Digital hard drives (WD3200AAJS), one for hosting operating system and the other for storing user data and servicing I/O accesses. In the experiments with SSD, the second hard disk is replaced with a Samsung 840 EVO 1-TB SSD. Peak throughputs of the disk and the SSD are around 100 MB/s and 250 MB/s, respectively.

There are two commonly used methods for hosting a virtual disk image, either using a logical volume supporting on-demand space allocation, or using a file, where the image is actually a file in the host server's file system and the file system translates an address on the image file to an address on the physical disk. We experiment with both methods, where Linux LVM is used to manage the logical volume and the file system can be either Ext4 or Btrfs. We use micro-benchmark `fio` (fio 2014) to generate I/O workloads, and applications, such as `dd`, `Postmark`, and Linux kernel build. The benchmarks access the guest's virtual disk via a raw partition (`/dev/vdb`) to avoid interference from guest's file system.

3.1 Performance with Micro-benchmark

For experiments with the `fio` micro-benchmark, the virtual disk image is first hosted on a logical volume. In each experiment, we change number of threads that concurrently send synchronous write requests to the disk at their highest rate. Each thread issues a FLUSH after each write to ensure that data are safely saved on the disk. Each thread writes to its own 8GB virtual disk space starting at offset (thread-id × 8 GB). Here we do not include QED, as it functions correctly only when a virtual disk is hosted on a regular file.

Figure 5 shows the throughput with threads making sequential accesses using various write request sizes. As shown in Figure 5a, without request concurrency (one thread) and with small request (64 KB) QCOW2's throughput (3.5 MB/s) is substantially lower than that of Selfie, FVD, and HOST (around 6 MB/s). While QCOW2 needs to update its metadata (L2-table block) every 64 KB data write, FVD updates its metadata (L1 table) every 1 MB data write,

and Selfie and HOST do not have any metadata to update. However, the throughput difference is relatively small, because frequent FLUSHes between small writes have already made the physical disk very inefficient.

There are two factors whose changes may cause significant increase of Selfie's throughout. One is to increase request concurrency by having more threads. As seen in Figure 5a, its throughput increases almost linearly. In contrast, HOST has little improvement. Higher concurrency is not translated into better performance because the concurrent requests from different threads are essentially random ones, and the disk head thrashes between the 8 GB disk regions accessed by different threads. For QCOW2 and FVD, both need to write metadata with data writes. For correctness not only writes of data and metadata from one thread but also updates of metadata about writes from different threads need to be serialized. This mostly cancels the benefit of high concurrency. In contrast, Selfie takes the best of both worlds. Like HOST, Selfie writes only data blocks without paying the cost of writing metadata and enforcing write ordering. Meanwhile, like QCOW2 and FVD the image is written in the request arrival order, turning random writes from different threads on the virtual disk into sequential ones.

With larger request sizes (512 KB and 4 MB), as shown in Figures 5b and 5c HOST's throughout significantly increases as large requests greatly reduce the impact of random access. Selfie's throughput also increases with larger requests. However, with large requests it almost does not increase with the thread count because Selfie has reached to its highest possible throughput (around 33 MB/s) at small thread count. Note that by clearing any newly allocated zones with zeros, Selfies essentially doubles amount of data written to a disk [2]. Also its throughput is limited by FLUSHes between requests. Without the additional write, HOST's throughput reaches its peak value at around 80 MB/s.

Figure 6 shows throughput of random accesses, which are evenly distributed over a 300 GB virtual disk space. As expected, QCOW2 and Selfie's throughput does not decrease when the I/O pattern changes from sequential access

---

[2] The Selfie's zone pre-clearing cost can be mostly removed on SSDs supporting the TRIM command.
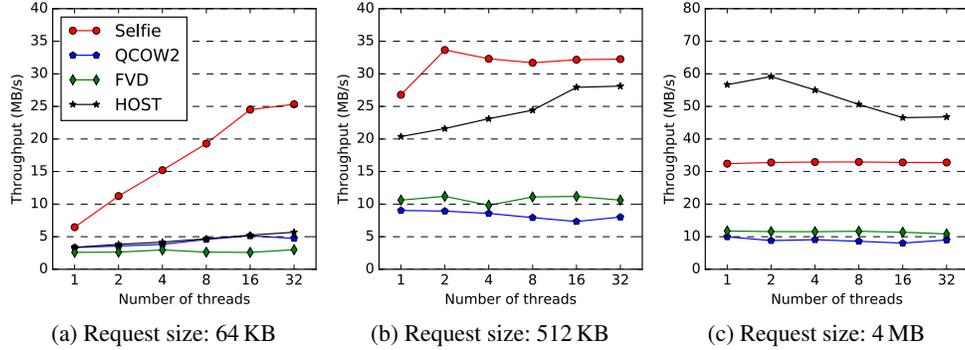
(a) Request size: 64 KB (b) Request size: 512 KB (c) Request size: 4 MB

Figure 6: *Throughput of `fio` with the virtual image hosed on the logical volume and each thread making* **random** *accesses.*

(Figure 5) to random access (Figure 6). They always make any writes to newly allocated space sequential. In contrast, HOST's throughput decreases significantly as the access randomness directly results in more frequent disk head seeks. FVD's throughput with 4 KB and 512 KB requests decreases slightly. FVD's allocation unit is 1 MB. When the random requests are smaller than 1 MB, writes to different 1 MB clusters are still random, reducing its throughout.

We re-do the experiments shown in Figure 5 with the image hosted by a file on Ext4 file system. Because QED functions with file-system supported image, we include it here. The results are presented in Figure 7. Comparing the two figures, we observe that Selfie mostly keeps its throughput on the logical volume. However, both QCOW2 and FVD have substantial throughput decreases. This difference is caused by their different methods of growing virtual disk images. Selfie grows its image by one zone (64 MB) at a time, while QCOW2 and FVD do by a cluster of much smaller sizes (4 KB and 1 MB, respectively) at a time. Frequently growing a file and updating the file size increase the burden on the file system by adding more file-system metadata operations such as journaling and metadata writes. This issue does not disappear with large requests because the allocation unit does not change with request size. In contrast, by using a large allocation unit, Selfies effectively allows file system to amortize cost of the metadata operations. QED's throughout is higher than that of QCOW2 and FVD. However, it is far lower than that of Selfie. QED's performance advantage lies on its elimination of write ordering in the metadata writes. However, it still needs synchronously write metadata, while Selfie completely removes metadata writing from the critical path of write. We also experiment with random requests on the file-system hosted images, and have similar observations.

3.2 Performance with Other Applications

We now experiment with other benchmarks including some commonly used applications. They access virtual disk via the guest file system and disk images are supported by a logical volume on the host system, unless stated otherwise.

The first application is Linux command **dd** that runs on the guest system to read a 8 GB file from a virtual disk and then write it back as a new file. In the experiment we would like to answer three questions. First, does Selfie affect the read performance? Second, does Selfie still have performance advantage with asynchronous write? Third, how does Selfie perform with writes of incompressible data?

As shown in Figure 8, the read performance is little affected by Selfie, which is expected as Selfie maintains a data layout on the image similar to that of QCOW2. To answer the second question, we issue a "sync" only at the end of `dd`'s execution to write the dirty data back, which consumes most of the program's write time. As the server has 64 GB memory, almost all the written data stay in memory before the sync. As shown, compared to QCOW2 and FVD, Selfie reduces the execution time by 77% and 66%, respectively, when all data can be compressed. Without frequent FLUSHes between requests in this asynchronous write, QCOW2 and FVD's performance has little improvement (compare Figures 5c and 8), as FLUSHes between data and metadata writes within individual requests remain.

To answer the third question, we make 50% or 100% of data blocks incompressible and re-do the experiment, and the write time is increased by 75% and 153%, respectively. These results are expected as uncompressed data are written to N-zones with additional metadata writes. However, even without any compressible blocks, Selfie still provides execution times 19% and 43% smaller than those of FVD and QCOW2, respectively. This is because Selfie allows out-of-order update of data and metadata by using pre-zeroed zones, an advantage shared by QED. We further run an experiment, in which the 100% incompressible file overwrites the compressible file currently on the Selfie image, to evaluate the impact of operations for relocating data from Z-zones to N-zones. The measurement shows the write time has little difference from that presented in Figure 8 for writing a 100% incompressible file as a new file. This is because the relocations do not involve any read operations and any additional writes to invalidate the original data in the Z-zones.

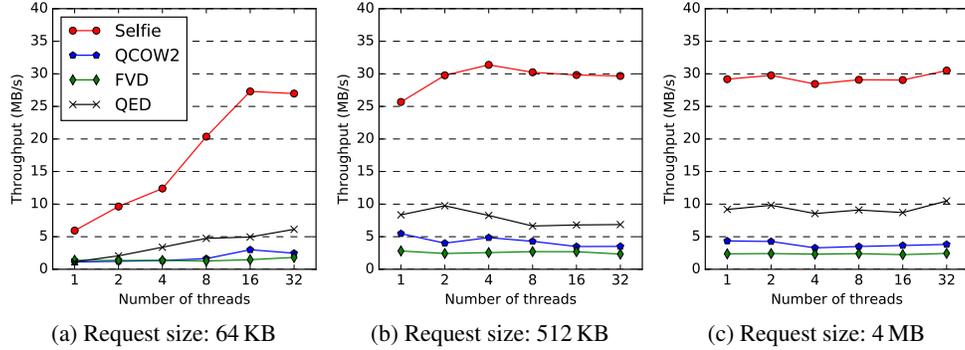To determine the impact of compression cost on the performance of `dd` with 100% compressible data, we simply by-

(a) Request size: 64 KB   (b) Request size: 512 KB   (c) Request size: 4 MB

Figure 7: *Throughput of* `fio` *with the virtual image hosed on the Ext4 file system and each thread making* **sequential** *accesses.*
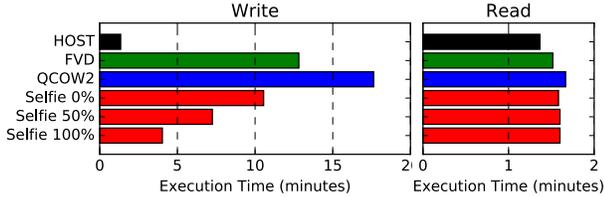

Figure 8: *Execution time of* `dd` *for reading an 8 GB file from a virtual disk to memory, and then writing it back to the virtual disk. A* `sync` *is issued after the writing. For Selfie there are three cases: 0%, 50%, or 100% of blocks are compressible.*

Table 1: *Breakdown of execution time of Kernel Build.*

	mkfs	copy	make
Selfie	4.062s	4.682s	22m4.452s
QCOW2	31.584s	12.935s	31m30.410s
FVD	14.714s	7.469s	34m56.730s
Host	3.961s	0.831s	20m41.463s

pass the (de)compression operation in Selfie without altering the I/O sequence. We found that there is almost no change of its read and write times with this bypassing. This is not surprising as LZ4 can do compression at 340 MB/s and decompression at 610 MB/s, much higher than disk's throughput. In addition, because only the lead block in a Z-cluster needs to be (de)compressed, merely 1/16 of the data are involved. The (de)compression cost is negligible to Selfie.

The second benchmark application is **kernel build**, which prepares and compiles Linux kernel's source code package, with substantial computation time. It consists of three phases. It first uses `mkfs.ext4` to create a clean file system, then uses `cp` to copy a compressed Linux-3.16.2 tarball from a file system on another disk to the new file system, and finally uses `makepkg` to automatically build the kernel (tpowa 2015). The `makepkg` phase entails unpacking the source code, applying patches, making the binary, and creating a installation package. We use "`-j8`" make flag for multi-threaded compiling.

As listed in Table 1, Selfie runs much faster than other two drivers (QCOW2 and FVD) in all three phases. Except in the file copy phases, Selfie's time is close to that of HOST

(directly running on the host system). The copied file is a 77 MB compressed tarball, which Selfie has to store them in N-zones and incur the cost for updating metadata. However, in the kernel make phase, the generated object and binary files are compressible, and Selfie makes I/O operations much more efficient. Even though this phase is mostly CPU-intensive (e.g., the 'depmod' operation keeps CPU fully busy for around two minutes for generating module dependence), Selfie still reduces this phase's execution time by 30% and 37% over QCOW2 and FVD, respectively.

The third application is **Postmark**, an I/O-intensive benchmark simulating the behavior of mail servers. It consists of three phases. Here, it first creates 100,000 files, then conducts mixed operations, including file creation, deletion, read, and append, where 500000 files are involved. Finally all files are deleted. We adopt the benchmark's default file size distribution between 500 B to 9.77 KB. In the execution about 2/3 of the disk writes are overwrites, which do not cause metadata writes on the image. In addition, 17% of the first-time writes are served in N-zones in Selfie. Figure 9(a) shows the execution times. Even though with significant portion of I/O accesses whose performance Selfie cannot improve, such as reads and overwrite, and I/O accesses for which Selfie cannot remove metadata writes, such as writes to N-zones, Selfie retains its performance advantage over FVD and QCOW2 (2.1X and 2.6X, respectively), and provides a performance closer to that of HOST.

For this benchmark, we also replace the hard disk with an SSD and re-do the experiment. As shown in Figure 9(b), Selfie's performance improvement over FVD and QCOW2 is reduced to about 1.4X. For SSD, Selfie's benefit on reduced writes and FLUSHes is retained. While reducing writes helps with SSD's durability, reducing FLUSHes contributes most of the performance improvement.

Furthermore, to understand the impact of a log-structured file system on the virtual disks' performance, we replace Ext4 with Btrfs as the host filesystem. Figures 9(c) and (d) show Postmark's execution times with Ext4 and Btrfs. It is obvious that the performance is significantly worse on Btrfs than that on Ext4, which is consistent to the suggestion provided on the official KVM website (KVM 2014).
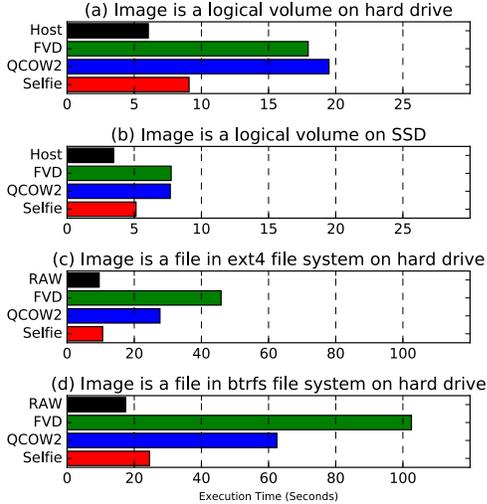
(a) Image is a logical volume on hard drive
(b) Image is a logical volume on SSD
(c) Image is a file in ext4 file system on hard drive
(d) Image is a file in btrfs file system on hard drive

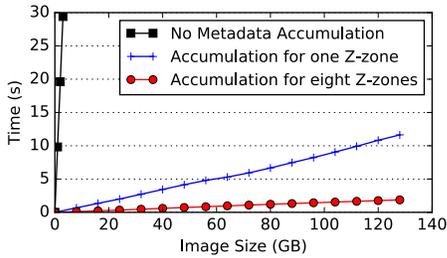Figure 9: *Execution times of Postmark*

Figure 10: *Recovery time for a Selfie image of different size.*

3.3 Recovery Time

In this section we measure the time of recovering a Selfie's image, that was created by running dd with 100% compressible data blocks, after a failure. Assuming there are no Z-zone committing operations before the failure, the recovery time is expected to be proportional to the image size. To measure the time period it takes for the service to become available, we include the time for reading mapping information from all of the Z-zones and committing them to the in-memory lookup table in the recovery time. Figure 10 shows the time with different image size in three cases: metadata are spread over individual Z clusters, one Z-zone's metadata are collected into a metadata cluster, or multiple Z-zones' (up to eight) metadata are collected into a metadata cluster. In each case, the time is indeed proportional to the image size. The time saving due to use of the metadata clustering is significant. Recovering a 128 GB image without using metadata cluster takes about 21 minutes. By using a metadata cluster to collect metadata for one Z-zone, the time is reduced to 11.6 seconds. If we further allow multiple Z-zones' metadata to be packed into one metadata cluster, the time is reduced to mere 1.9 seconds.

4. Related Work

Selfie applies the compression technique to address the performance issue with metadata updates in virtual disks. Be-

low we will describe related work on the compression technique, the efforts on the issue with metadata in virtual disks, and finally the efforts on reducing metadata overhead in general.

4.1 Data compression

Compression is a widely used technique to reduce data size for more compact storage. It may also help with I/O performance. One example is to compress process memory and write-back data in memory so that more data can be held in memory for virtual memory or file data to reduce I/O access (Jennings 2013; VMWare 2011). With data compression before they are written to the disk, smaller amount of data is written and read for higher effective I/O throughput. Some file systems support transparent data compression for space and I/O efficiency, such as NTFS (btdebug 2008), Btrfs (Btrfs 2014), and ZFS (OpenZFS 2015). Their approach is to compress multiple contiguous data pages (usually 64 KB or larger) into fewer ones and probably pad the last page with zeros for alignment. Selfie looks for compression opportunities at the block device level. Even for the data compressed by an operating system, the padded pages can be still available for further compression in Selfie.

Like Selfie, ZBD also applies compression at the block level between OS and the storage (Klonatos et al. 2012; Makatos et al. 2010). Using block-level compression, ZBD can increase effective storage capacity by up to 2X. The work also shows that most of its workload can be compressed. This is consistent to what's reported on data stored in a production deduplication file system, where data that have been deduplicated can be further compressed at a ratio of about 2 (Zhu et al. 2008; Wallace et al. 2012). MC is a technique for improving compression ratio by rearranging data blocks in a large data file (Lin et al. 2014). The work shows that many full system backups and VM images can be compressed by at least 2x to 3x. These results suggest Selfie's potential for large performance gains.

One limitation with the use of compression in existing storage systems is that benefit of compression is proportional to the compression ratio. However, Selfie does not rely on high compression ratio. Actually, the fast LZ4 algorithm with moderate compression ratio is sufficient to meet Selfie's needs. Moreover, Selfie applies compression on only 1/16 of the blocks to remove metadata operations.

By applying compression, existent systems, such as ZBD, have to introduce additional metadata attached to the compressed data. The additional cost for accessing the metadata may offset their benefits. Experiments about ZBD show its performance does not increase in most of its test cases, and sometimes it can be worse than that of a system without using compression. In contrast, Selfie efficiently uses large zones to recognize compressed and uncompressed data. Rather than adding new metadata, Selfie removes metadata overhead out of a system's critical path.

4.2 Virtual Disk Drivers

Virtual disks have been a focus for improvement, and many efforts have been made to reduce unnecessary writes and exploit I/O parallelism. For example, to support snapshots internal within an image, QCOW2 uses a metadata (reference count) to track the number of references for clusters in multiple snapshots. Immediate updating of this metadata is expensive. So it is removed from critical path by employing after-failure scanning of all snapshots to recover any out-of-date counts (Hajnoczi 2012). In this sense, Selfie similarly uses scanning to recover dispersed mapping information. To speed up the process, Selfie pre-collects the information. The challenge for QCOW2 is that it cannot directly use the approach to avoid immediately updating of mapping information, while it is a unique contribution for Selfie to embed the information within data blocks to address the issue.

The necessity for immediately mapping metadata is a major performance concern demanding optimization. It usually means frequent and in-order additional writes, leading to more head seeks and reduced concurrency. There are optimizations attempting to ameliorate the issue. FVD uses a larger mapping unit and reduces the lookup table to one level to reduce the frequency of metadata update (Tang 2011). Because FVD assumes an effective mapping provided by the host storage between the address in its mapping unit and physical disk, it pushes the burden of managing the mapping information to the next lower level. Moreover, using a large mapping unit weakens the advantage of thin provisioning.

QED takes a different approach. Instead of reducing metadata access, QED improves the I/O concurrency for both data and metadata by relaxing the order requirement among the writes (Hajnoczi 2010). To make the relaxation possible, QED requires its image be mapped to a regular file. However, disk partitions and logical volumes are important abstractions to support virtual disk images. QED has limited uses with this restriction.

There are virtual disk drivers from major companies, such as VMDK from VMWare (VMWare 2007), VHDX from Microsoft (Microsort 2012), and VDI from VirtualBox (Ellison 2008). They use image layouts for on-demand allocation similar to that of QCOW2, and share the performance issue on inefficient metadata updating with QCOW2.

4.3 Updating Metadata in File System

Like a virtual disk image, a file system consists of data and metadata including inode and dentry. When they are committed to the disk, these data and metadata must be consistent. To guarantee the consistency, their writes must follow a predefined order. However, doing this would reduce the request concurrency and limit the effectiveness of the I/O scheduler for higher disk efficiency. Chidambaram et al. eliminate the ordering in serving requests by introducing back-pointers (Chidambaram et al. 2012). This method requires use of SCSI drive for writing the back-pointer in

the drive's 8-byte checksum area attached to each 512-byte sector. ReconFS adopts a similar approach for flash disks by temporarily writing metadata into the OOB (out-of-band) area in SSDs (Lu et al. 2014), which enables lazy metadata update. It relies on SSD to expose the internal area to users by providing additional API functions. OptFS optimizes journaling commitment by introducing Asynchronous Durability Notification command to a disk interface, removing expensive disk flushes (Chidambaram et al. 2013).

A critical issue with these strategies is that their real-world applications depend on particular hardware supports and the required supports are either only available on less-popular disks or not available yet, and their benefits cannot be materialized until the hardware vendors are willing to provide the supports. Even for devices providing extra space for each block, the space may not be available for storing the metadata. For example, the 8-byte checksum area in SCSI disks may have been occupied for end-to-end data protection. Even for relatively large OOB area in the SSD, the ECC scheme may leave little or no space for other metadata (STLinux 2014). In contrast, Selfie does not have such limitation at all. It is readily usable on commodity block devices, and the devices do not have to be aware of the optimization.

5. Conclusions

In this paper we propose Selfie, a virtual disk driver, that removes all overheads associated with metadata updating out of the critical path, including data seeks for extra writes and flushes for enforcing correct request service order. It opportunistically takes advantage of data block's compressibility and embeds metadata into data blocks. Compared with other similar efforts, Selfie addresses the root cause of the inefficiency issue—existence of the metadata (look-up table), and removes the need of immediately updating the table on the disk.

We have implemented a Selfie disk driver on QEMU. Experiments show that Selfie provides a much better performance than other drivers (QCOW2, QED, and FVD). Specifically, its throughput is higher than that of the other drivers by 2 to 8 times for synchronous writes, and by 1.5 to 4 times with macro-benchmarks (dd, postmark, and kernel build). It takes less than two seconds to recover or reboot from a large Selfie image (128 GB). A notable advantage of the technique is that it does not require any hardware support and is ready for use on any stock block devices. Without much effort, this technique can be extended into other system or application scenarios, such as file systems and databases where metadata causes substantial performance drag.

6. Acknowledgments

References

btdebug. Understanding ntfs compression. `http://goo.gl/P9FfRk`, 2008.

Btrfs. Compression - btrfs wiki. `http://goo.gl/1RJRpC`, 2014.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. . URL `http://doi.acm.org/10.1145/1365815.1365816`.

V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2208461.2208470`.

V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL `http://doi.acm.org/10.1145/2517349.2522726`.

J. C.Mogul. A better update policy. In *Proceedings of 1994 Summer USENIX Conference*, 1994.

Y. Collet. lz4. `http://cyan4973.github.io/lz4/`, 2015.

T. Ellison. All about vdis. `http://goo.gl/wgmtZN`, 2008.

fio. fio. `http://freecode.com/projects/fio`, 2014.

S. Hajnoczi. qed: Add qemu enhanced disk format. `http://goo.gl/WrRHtE`, 2010.

S. Hajnoczi. qcow2: implement lazy refcounts. `http://goo.gl/32U8UC`, 2012.

T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *ACM Symposium on Operating System Principles(SOSP)*, 2011.

S. Jennings. Transparent memory compression in linux. `http://goo.gl/xAqSsP`, 2013.

Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas. Transparent online storage compression at the block-level. *Trans. Storage*, 8(2):5:1–5:33, May 2012. ISSN 1553-3077. . URL `http://doi.acm.org/10.1145/2180905.2180906`.

KVM. Tuning kvm. `http://www.linux-kvm.org/page/Tuning_KVM`, 2014.

H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. . URL `http://doi.acm.org/10.1145/1519065.1519067`.

X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 257–271, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-08-9. URL `http://dl.acm.org/citation.cfm?id=2591305.2591330`.

Y. Lu, J. Shu, and W. Wang. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 75–88, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-08-9. URL `http://dl.acm.org/citation.cfm?id=2591305.2591313`.

T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 1–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. . URL `http://doi.acm.org/10.1145/1755913.1755915`.

M. McLoughlin. Qcow2 the qcow2 image format. `http://goo.gl/Nj6vO3`, 2008.

Microsort. Vhdx format specification v1.00. `http://goo.gl/PmBJjC`, 2012.

OpenZFS. Zfs features. `http://open-zfs.org/wiki/Features`, 2015.

oVirt. Vdsm disk images (ovirt). `http://wiki.ovirt.org/Vdsm_Disk_Images`, 2014.

QEMU. Qemu/images. `http://en.wikibooks.org/wiki/QEMU/Images`, 2015.

STLinux. Data storage on nand flash. `http://www.stlinux.com/howto/NAND/data`, 2014.

C. Tang. Fvd: A high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2002181.2002199`.

tpowa. linux-3.16.2 pkgbuild. `http://goo.gl/ORXBgU`, 2015.

VMWare. Virtual disk format 1.1. `http://goo.gl/f4R1Ho`, 2007.

VMWare. Understanding memory resource management in vmware vsphere 5.0. `http://goo.gl/X8ZyDn`, 2011.

G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2208461.2208465`.

B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1364813.1364831`.