RevDedup: A Reverse Deduplication Storage System Optimized for Reads to Latest Backups

Chun-Ho Ng and Patrick P. C. Lee
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Hong Kong
ngch.hk@gmail.com, pclee@cse.cuhk.edu.hk

Abstract

Deduplication is known to effectively eliminate duplicates, yet it introduces fragmentation that degrades read performance. We propose *RevDedup*, a deduplication system that optimizes reads to the latest backups of virtual machine (VM) images using reverse deduplication. In contrast with conventional deduplication that removes duplicates from new data, RevDedup removes duplicates from old data, thereby shifting fragmentation to old data while keeping the layout of new data as sequential as possible. We evaluate our RevDedup prototype using a 12-week span of real-world VM image snapshots of 160 users. We show that RevDedup achieves high deduplication efficiency, high backup throughput, and high read throughput.

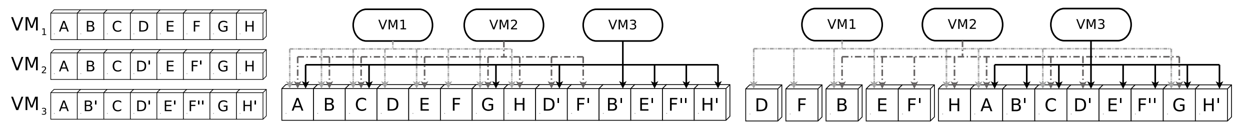
1 Introduction

Many enterprises today run virtual machines (VMs) to reduce hardware footprints. For disaster recovery, conventional approaches schedule backups for each VM disk image and keep different versions of each VM backup. Today's backup solutions mainly build on disk-based storage. While the harddisk cost is low nowadays, in the face of a large volume of VMs and a large volume of versions associated with each VM, scaling up the backup storage for VM images (typically of size several gigabytes each) remains a critical deployment issue.

Deduplication improves storage efficiency by eliminating redundant data. Instead of storing multiple copies of data blocks with identical content, deduplication stores only one copy of identical blocks and refers other blocks to that copy via smaller-size references. Deduplication is mainly studied in backup systems (see §2). It also provides space-efficient VM image storage since VM images have high content similarities [5].

However, deduplication has a drawback of introducing *fragmentation* [6, 8, 10, 14, 15], since some blocks of a file may now refer to other identical blocks of a different file. To illustrate, Figure 1(a) shows three snapshots of a VM, denoted by VM_1, VM_2, and VM_3, which are to be written to disk that initially has no data. Figure 1(b) shows how a conventional deduplication system writes data. First, it writes VM_1 with unique blocks A to H sequentially. For VM_2, which some of the blocks are identical to those of VM_1, the system stores only the references that refer to those identical blocks, and appends the unique blocks D' and F' to the end of the last write position. Similarly, for VM_3, the system only writes the unique blocks B', E', F'', and H' to the end of the last write position. Hence, reading VM_3 is no longer sequential, but requires additional disk seeks to the identical blocks being referenced. This significantly degrades read performance. On the other hand, we believe that achieving high read throughput is necessary in any backup system. For instance, a fast restore operation can minimize the system downtime during disaster recovery [10, 16].

In practice, users are more likely to access more recent data. Our key insight is that traditional deduplication systems check if new blocks can be represented by any already stored blocks with identical contents. Thus, the fragmentation problem of the latest backup is the most severe since its blocks are scattered across all the prior backups. To mitigate fragmentation in newer backups, we propose to do the opposite, and check if any already stored blocks can be represented by the new blocks to be written. We remove any duplicate blocks that are already stored so as to reclaim storage, and refer them to


VM1
VM2
VM3
A B C D E F G H
A B C D' E F' G H
A B' C D' E' F'' G H'
A B C D E F G H D' F' B' E' F'' H'
D F B E F' H A B' C D' E' F'' G H'

(a) Backups (b) Conventional deduplication (c) Reverse deduplication

Figure 1: An example of how conventional deduplication and reverse deduplication place data on disk.

the new blocks. This shifts the fragmentation problem to the older backups, while keeping the storage layout of the newer backups as sequential as possible. We call this *reverse deduplication*, which is the core component of our deduplication design. Figure 1(c) shows the disk layout for the previous example when reverse deduplication is used, such that the blocks of VM_3 (i.e., the latest backup) are sequentially placed on disk. Also, VM_2 is less fragmented than VM_1, in the sense that the blocks of VM_2 are less spread out on disk than those of VM_1. Thus, the newer a backup is, the ~~less~~ fragmentation overhead the backup will experience.

In this paper, we propose *RevDedup*, a deduplication system for VM image backup storage. RevDedup exploits content similarities of VM images using a hybrid of inline and out-of-order deduplication approaches. It applies coarse-grained global deduplication (inline) to different VMs and removes any duplicates on the write path, and further applies fine-grained reverse deduplication (out-of-order) to different backup versions of the same VM and removes any duplicates from old backup versions. We propose a configurable, threshold-based block removal mechanism that combines *hole-punching* [9] to remove duplicate blocks of old backup versions and *segment compaction* to compact data segments without duplicate blocks to reclaim contiguous space.

We implement RevDedup and experiment the prototype on a RAID disk array using real-life VM image snapshots for 160 users over a 12-week span. We show that RevDedup achieves (i) high deduplication efficiency with around 97% of saving, (ii) high write throughput at 4-7GB/s, and (iii) high read throughput for the latest backup at 1.2-1.7GB/s. We also show that conventional deduplication experiences throughput drop when retrieving newer backups.

2 Related Work

Deduplication is first studied in Venti [13] for data backup. To minimize memory usage for indexing, DDFS [16] and Foundation [14] use the Bloom filter, while other studies [1, 4, 7] exploit workload characteristics. The above studies aim to achieve high write throughput, low memory usage, and high storage efficiency in deduplication, but put limited emphasis on read performance. Kaczmarczyk *et al.* [6] also propose to improve read performance for latest backups in deduplication as in our work. Their system selectively rewrites deduplicated data to disk to mitigate fragmentation, and removes old rewritten blocks in the background. However, they consider deduplication for different versions of a single backup only, while we enable global deduplication across multiple VMs. Nam *et al.* [10] propose to measure the fragmentation impact given the input workload and activates selective deduplication on demand. Lillibridge *et al.* [8] use the container capping and forward assembly area techniques to improve the restore performance. Unlike the previous studies [6, 8, 10] that aim to remove duplicates from new data, we use a completely different design by removing duplicates from old data to maintain high deduplication efficiency and inherently making older (newer) backups more (less) fragmented. iDedup [15] also aims to optimize read performance in deduplication, but it targets primary storage.

3 RevDedup

RevDedup is a deduplication system for backing up VM images. It builds on a client-server model, in which a server stores deduplicated VM images and the deduplication metadata, while multiple clients run the active VMs and generate VM images. RevDedup considers a single snapshot of a VM image as a backup. We call different snapshots that belong to the same VM to be *versions*.

We assume that RevDedup applies fixed-size chunking to backup streams, i.e., we divide data into fixed-size units each identified by a fingerprint, and determine if the unit can be deduplicated. Fixed-size chunking shows significant storage savings for VM images [5], while having smaller chunking overhead than variable-size chunking.

3.1 Coarse-Grained Global Deduplication

RevDedup first applies coarse-grained global deduplication to the already stored VM snapshots. By coarse-grained, we mean that RevDedup applies deduplication to large fixed-size units called *segments*, each of which has a size of several megabytes. By global, we mean that we apply deduplication to all versions and eliminate duplicate segments that appear (i) in the same version, (ii) in different versions of the same VM, or (iii) in different versions of different VMs. Each segment is identified by a fingerprint that is generated from the cryptographic hash of the segment content.

With a large segment size, the disk seek time of locating segments only forms a small portion of the total time of reading all segment contents of a VM image. Thus, we mitigate fragmentation by amortizing disk seeks over large-size segments [7, 15]. Evaluations on our real-world dataset (see §4) show that coarse-grained global deduplication itself still achieves high deduplication efficiency. Nevertheless, we point out that it cannot maintain the same level of deduplication efficiency as in existing fine-grained deduplication approaches, as shown in §3.2.

3.2 Fine-Grained Reverse Deduplication

RevDedup also applies more fine-grained deduplication on a sub-segment level to further eliminate duplicates. We define smaller fixed-size sub-segments called *blocks*, each of which has a size of several kilobytes (e.g., using the native file system block size 4KB). Like segments, each block is identified by a fingerprint given by the cryptographic hash of the block content.

3.2.1 Reverse Deduplication on Unique Segments

We first consider how reverse deduplication operates on different versions of a single VM, assuming that all segments are unique and there is no global deduplication across segments. Note that different unique segments may still share identical blocks.

Figure 2 shows how reverse deduplication works with three versions VM_1, VM_2, and VM_3. Each version contains a number of block pointers, each of which holds either a *direct reference* to the physical block content of a segment, or an *indirect reference* to a block pointer of a future version. An indirect reference indicates that the block can be accessed through some future version. In RevDedup, any latest version of a VM must have all block pointers set to direct references.

Suppose that the system has already stored a version VM_1, and now a new version VM_2 of the same VM is submitted to the system. We compare VM_1 and VM_2 by loading all their block fingerprints from disk (see §3.3). If a matched block is found in both VM_1 and VM_2, we remove the respective block of VM_1, and update that block with an indirect reference that refers to the identical block of VM_2. Now if we write another version VM_3 of the same VM, we compare its blocks with those of VM_2, and remove any duplicate blocks of VM_2 as above. Some blocks of VM_1 now refer to those of VM_3. To access those blocks of VM_1, we follow the references from VM_1 to VM_2, and then from VM_2 to VM_3.

In general, when writing the ith version VM_i, we compare the block fingerprints of VM_i with those of the pre-

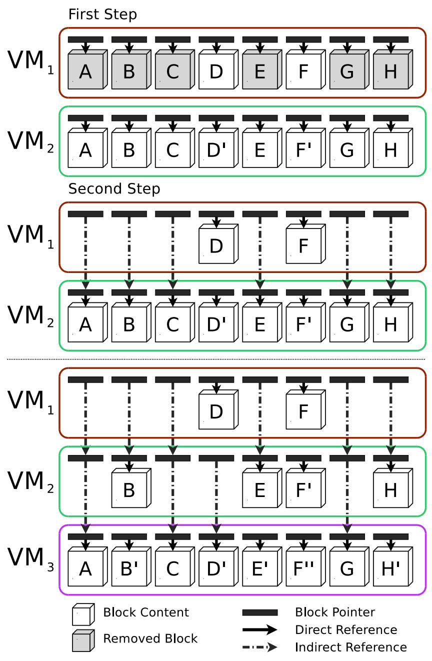


Figure 2: An example of reverse deduplication for multiple versions of the same VM.

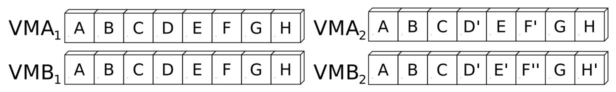
vious version VM_{i-1}. We remove any duplicate blocks of VM_{i-1} and update the block pointers to refer to the identical blocks of VM_i. To simplify the deduplication process, one key assumption we make is that we only compare the two most recent versions VM_i and VM_{i-1}. Hence, we may miss the deduplication with the redundant blocks of earlier versions (i.e., VM_{i-2}, VM_{i-3}, $\cdots$, etc.). Nevertheless, the analysis of our real-world dataset (see §4) indicates that such misses are unlikely and only contribute 0.6% of additional space.

When reading VM_i, we either follow the direct reference to access the physical block, or a chain of indirect references to future versions (i.e., VM_{i+1}, VM_{i+2}, $\cdots$, etc.) until a direct reference is hit.

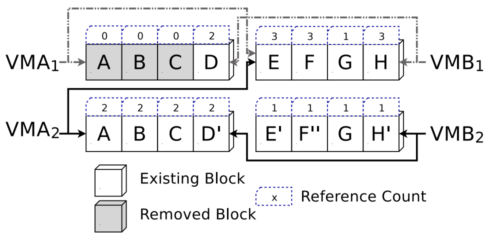
3.2.2 Reverse Deduplication on Shared Segments

When segment-level global deduplication is in effect, we cannot directly remove a block whose associated segment is shared by other versions or within the same version. RevDedup uses *reference counting* to decide if a block can be safely removed. We associate each block with a reference count, which indicates the number of direct references that currently refer to the block among all versions of the same VM or different VMs.

Figure 3 shows an example of how reference counts are used. Suppose that two VMs, namely VMA and VMB, are stored. Let the segment size be four blocks. The first versions VMA_1 and VMB_1 have the same set of blocks. For the second versions, VMA_2 has new blocks



(a) VM versions



(b) Reference counts

Figure 3: An example that shows how reference counts are assigned when reverse deduplication is applied to shared segments.

D′ and F′, while VMB_2 has new blocks D′, E′, F″, and H′. We see that any blocks with zero reference counts (in the segment ABCD) can be safely removed.

With reference counting, we now describe the complete reverse deduplication design. When a client writes the *i*th version VM_i of a VM, the server first applies global deduplication with the segments of other VMs. For each segment of VM_i, if it is unique, then the reference counts of all associated blocks are initialized to one; if the segment can be deduplicated with some existing segment, then the reference counts of all associated blocks of the existing segment are incremented by one. Next, the server loads all the block fingerprints of VM_{i-1} (the previous version) and VM_i into memory. It applies reverse deduplication and compares the block fingerprints of VM_{i-1} and VM_i. If a block of VM_{i-1} can be deduplicated with some block of VM_i, then the block of VM_{i-1} will have its reference count decremented by one and its direct reference updated to an indirect reference that refers to the block of VM_i. If the reference count reaches zero, it implies that the block (of VM_{i-1}) is not pointed by any direct references, but instead can be represented by the same block of future versions. It can thus be safely removed.

3.2.3 Removal of Duplicate Blocks

RevDedup builds on two approaches to remove duplicate blocks from segments, namely *block punching* and *segment compaction*.

Block punching leverages the hole-punching mechanism available in Linux Ext4 and XFS file systems [9], where we can issue in user space the system call `fallocate(FALLOC_FL_PUNCH_HOLE)` to a file region. Any file system block covered by the hole-punched region will be deallocated and have its space released. The respective block mappings of the file will be updated in the file system. Block punching involves file system metadata operations and is expected to incur small overhead. However, block punching has a drawback of incurring disk fragmentation *(note that it is different from the fragmentation problem in deduplication we discussed)*, as non-contiguous free blocks appear across disk.

Segment compaction is to compact a segment that excludes the removed blocks. It operates by copying all blocks of a segment, except those that are to be removed, sequentially into a different segment. The original segment will be deleted and have its space released, and the new segment is kept instead. Segment compaction mitigates disk fragmentation as it copies all remaining blocks in sequence. However, it has large I/O overhead since it reads and writes the actual data content of the non-removed blocks.

Hence, we propose a threshold-based block removal mechanism, which uses a pre-defined threshold (called the *rebuild threshold*) to determine how to rebuild a segment excluding removed blocks. If the fraction of blocks to be removed from a segment is smaller than the rebuild threshold, then block punching will be used; otherwise, segment compaction will be used. The rebuild threshold is configured to trade between disk fragmentation and segment copying time.

3.3 Indexing

We now describe how RevDedup performs indexing and identifies duplicates. Currently we use 20-byte SHA-1 for both segment and block fingerprints. For global deduplication, the server holds an in-memory index that keeps track of the fingerprints and other metadata of all segments. We argue that the index has low memory usage when using large-size segments. For example, suppose that the segment size is 8MB. For every petabyte of storage, we index 128 million entries. If each entry has size 32 bytes, which we believe suffice to store the fingerprint and other metadata for each segment, then the index has a total size of 4GB.

For reverse deduplication, we associate each segment with a metadata file that keeps the block fingerprints and reference counts of all blocks of the segment. All metadata files are stored on disk. When storing a version, RevDedup builds the index on the fly by loading the metadata files of all associated segments into memory. To quantify the memory usage, we consider the following parameters used in our evaluation: a 7.6GB VM image, 4KB blocks, and 32-byte block-level index entries. Since reverse deduplication operates by comparing two versions of VM images (see §3.2.2), the total memory usage is up to $2\times 7.6\text{GB} \div 4\text{KB} \times 32$ bytes $= 121.6$MB.

3.4 Implementation

Our RevDedup implementation builds on the client-server model as shown in Figure 4. RevDedup uses client-side deduplication to reduce the client-server communication overhead. When a client is about to submit a version of a VM to the server, it first divides the VM image snapshot into different segments and computes both segment-level and block-level fingerprints for the version. Next, the client queries the server, using the segment fingerprints, whether the segments are already stored in the server. If any segment has already been stored, then the client discards the upload of that segment. The client then uploads the unique segments to the server (e.g., via RESTful APIs). It also sends the metadata information, including all segment and block fingerprints for the whole VM image and the information of the version (e.g., the VM that it belongs, the size of the image, etc.). Note that we offload the server by having the clients be responsible for both segment and block fingerprint computations. This avoids overloading the server when it is connected by too many clients.

Upon receiving the unique segments and metadata information of a version, the server writes them to disk and links the version with the existing segments that are already stored. The server performs reverse deduplication as described in §3.2, including: loading metadata files and building the block fingerprint index on the fly, searching for duplicates and updating direct/indirect references, and removing duplicate blocks from segments via block punching or segment compaction.

Our RevDedup prototype is implemented in C in Linux. We use SHA-1 for both segment and block fingerprint computations. The RevDedup server mounts its data storage backend on a native file system. RevDedup requires that the file system support hole-punching, and here we use Ext4 for Linux. In the following, we address several implementation details.

Mitigating interference. Since a client may perform fingerprint computations for a running VM, minimizing the interference to the running VM is necessary. Here, we can leverage the snapshot feature that is available in today's mainstream virtualization platforms. The client can directly operate on the mirror snapshot in the background and destroy the snapshot afterwards.

Communication. The client-server communication of RevDedup is based on RESTful APIs, which are HTTP-compliant. A client can retrieve a VM image by issuing a correct HTTP request. The server can process multiple requests from different simultaneously.

Multi-threading. RevDedup exploits multi-threading to achieve high read/write performance. In writes, the server uses multiple threads to receive segment uploaded by the clients and to perform reverse deduplication. In reads, the server uses dedicated threads to pre-declare the segment reads in kernel by using the POSIX function `posix_fadvise(POSIX_FADV_WILLNEED)`. With read pre-declaration, the kernel can make effective pre-fetching of segments to improve read performance.

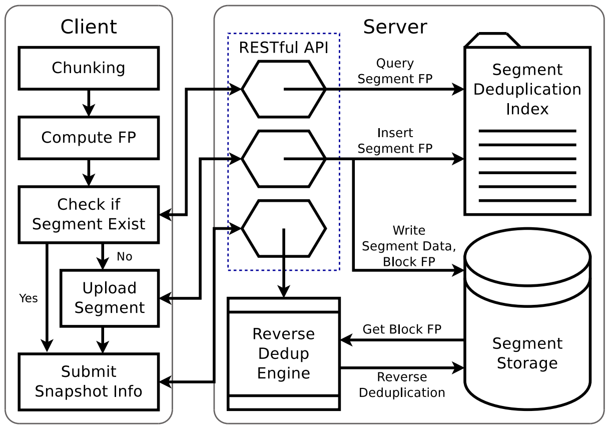


Figure 4: RevDedup's client-server model.

In addition, the server uses a separate thread to trace the chains of indirect references of blocks when old versions are read. Once the direct reference is found, the thread sends the block address to another thread for reading the block content. Both threads run concurrently. This reduces the overhead of tracing long indirect reference chains.

Handling of null blocks. In practice, VM images contain a large number of null (or zero-filled) blocks [5]. In RevDedup, the server skips the disk writes of any null blocks appearing in the segments submitted by a client. When a null block is to be read, the server generates null data on the fly instead of reading it from disk.

4 Evaluation

We conduct testbed experiments on our RevDedup prototype using real-life VM workloads. We show that RevDedup achieves high deduplication efficiency, high backup throughput, and high read throughput.

Testbed. Our experiments are conducted on a machine with a 3.4GHz Intel Xeon E3-1240v2 quad-core, eight-threaded processor, 32GB RAM, and a RAID-0 disk array with eight ST1000DM003 7200RPM 1TB SATA disks. We choose RAID-0 to maximize the disk array throughput for stress tests. The machine runs Ubuntu 12.04 with Linux kernel 3.2.0. Our measurements indicate that our testbed achieves the raw write and read throughputs at 1.37GB/s and 1.27GB/s, respectively.

Configurations. We consider four segment sizes for global deduplication: 4MB, 8MB, 16MB, and 32MB.

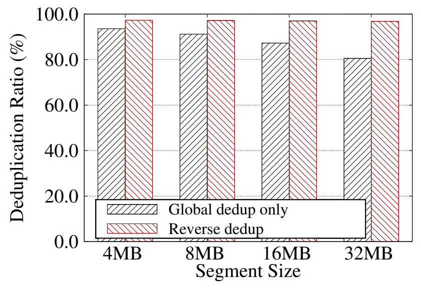


Figure 5: Deduplication ratio

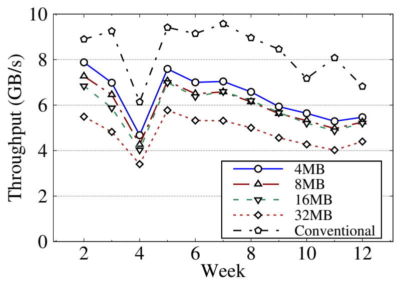


Figure 6: Backup throughput

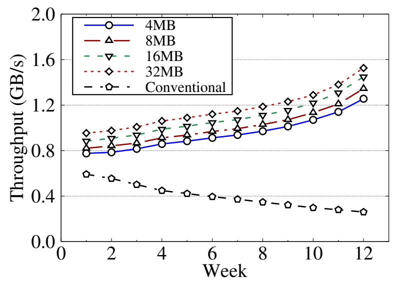


Figure 7: Read throughput

We fix the block size at 4KB for reverse deduplication to match the file system block size. We set the default rebuild threshold at 20% for our block removal mechanism. Suppose that the clients can generate VM snapshots and compute fingerprints offline before connecting to the server. Thus, we pre-compute all segment and block fingerprints before benchmarking. Our throughput results are averaged over five runs.

Dataset. We collected a dataset from a real-life virtual desktop environment. The dataset contains image snapshots of VMs used by university students in a computer science programming course. We prepared a master image of size 7.6GB with 32-bit Ubuntu 10.04 installed. We then cloned 160 VMs, and assigned one to each student to work on three programming assignments in a semester. We generated 12 weekly versions for each VM. If no deduplication is applied, the total size of all versions over the 12-week span is 14.3TB. If we exclude null (zero-filled) blocks, there is 6.67TB of data. Note that the way of cloning multiple images from a master image is also used in enterprises to standardize the operating systems [2]. By no means do we claim that the dataset can be generalized for any virtual desktop environments. We only aim to show the baseline performance of RevDedup in a special real-life use case.

Storage efficiency. We first evaluate the storage efficiency of RevDedup when storing the 12 weekly version sets. We measure the actual disk usage including both data and metadata. We define the *deduplication ratio* as the percentage of space saved with deduplication to the total size of all VM images (excluding null blocks) without deduplication. Here, we compare two variants of RevDedup: (i) only coarse-grained global deduplication is used, and (ii) both global and reverse deduplication approaches are used. Figure 5 shows the deduplication ratios. Coarse-grained global deduplication itself achieves space saving of 80.5-93.6%, while reverse deduplication further removes duplicates and increases the saving to 96.8-97.3%. We also consider conventional deduplication that operates on data units of small size. Suppose we choose the default block size 128KB used by Opendedup SDFS [12] and ZFS [3]. Then conventional deduplication has a deduplication ratio 96.8% [11], which is similar to that of RevDedup.

Backup throughput. Next, we compare the backup throughput of RevDedup and that of conventional deduplication. To evaluate the latter, we configure RevDedup to use a 128KB segment size for global deduplication and disable reverse deduplication. In our evaluation, the server has no data initially. Then we submit the 12 version sets in the order of their creation dates. We measure the time of the whole submission operation, starting from when the clients submit all unique segments until the server writes them to disk and performs reverse deduplication (for RevDedup). We call `sync()` at the end of each write to flush all data to disk. Here, we plot the results starting from Week 2, in which RevDedup begins to apply reverse deduplication to the version sets being stored. Figure 6 shows the throughput of RevDedup and conventional deduplication in the backup of each weekly version set. Conventional deduplication has higher backup throughput than RevDedup, for example, by an average of 30% compared to RevDedup with segment size 4MB. Nevertheless, RevDedup still achieves high backup throughput in the range around 4-7GB/s (about 3-5$\times$ of the raw write throughput) since it discards duplicates on the write path in global deduplication. A smaller segment size implies higher throughput as more duplicates are discarded. Note that there is a throughput drop in Week 4 due to significant modifications made to the VMs.

Read throughput. We evaluate the read throughput of RevDedup. After submitting all 12 version sets, we measure the time of reading each version set. Before each measurement, we flush the file system cache using the command "`echo 3 > /proc/sys/vm/drop_caches`". Figure 7 shows the throughput of RevDedup in reading earlier versions after storing all versions. We also include the results of conventional deduplication here. We observe that RevDedup confirms our design goal, as the read throughput decreases with earlier versions being read. For example, the read throughput for Week 1 is up to 40% less than that for Week 12. The figure also shows the fragmentation problem in conventional deduplication. For Week 1, the read throughput can only achieve 606MB/s (at least 25% less than RevDedup), mainly due to fragmentation introduced in global deduplication with the small segment size at 128KB. The read throughput decreases further for later weeks. It drops to 266MB/s for Week 12, which is only around 20% of the raw read throughput (1.27GB/s).

Discussion. In our technical report [11], we present additional experimental results. We conduct preliminary microbenchmark experiments on a VM with a long chain of versions. We evaluate the block removal time and the disk fragmentation overhead of RevDedup for different rebuild thresholds. We also evaluate the overhead in tracing indirect references for earlier versions.

5 Conclusions and Future Work

We present RevDedup, a deduplication system designed for VM disk image backup in virtualization environments. RevDedup has several design goals: high storage efficiency, low memory usage, high backup performance, and high restore performance for latest backups. The core design component of RevDedup is reverse deduplication, which removes duplicates of old backups and mitigates fragmentation of latest backups. We extensively evaluate our RevDedup prototype and validate our design goals. In future work, we plan to study the impact of variable-size chunking on RevDedup and evaluate RevDedup using more representative workloads.

Availability. The source code of our RevDedup prototype presented in this paper is available for download at: **http://ansrlab.cse.cuhk.edu.hk/software/revdedup**.

References

[1] D. Bhagwat, K. Eshghi, D.D.E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. IEEE MASCOTS*, Sep 2009.

[2] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proc. USENIX ATC*, Jun 2009.

[3] Scott Dickson. Sillyt ZFS dedup experiment. `https://blogs.oracle.com/scottdickson/entry/sillyt_zfs_dedup_experiment`, Dec 2009.

[4] F. Guo and P. Efstathopoulos. Building a high performance deduplication system. In *Proc. USENIX ATC*, Jun 2011.

[5] K. Jin and E.L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. SYSTOR*, May 2009.

[6] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. SYSTOR*, Jun 2012.

[7] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proc. USENIX FAST*, Feb 2010.

[8] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, Feb 2013.

[9] LWN.net. Punching holes in files. `http://lwn.net/Articles/415889/`.

[10] Y. Nam, D. Park, and D. Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proc. IEEE MASCOTS*, 2012.

[11] C. Ng and P. Lee. RevDedup: A reverse deduplication storage system optimized for reads to latest backups. Technical report, CUHK, Jun 2013. `http://arxiv.org/abs/1302.0621v3`.

[12] Opendedup. `http://www.opendedup.org/`.

[13] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, Jan 2002.

[14] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. USENIX ATC*, Jun 2008.

[15] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. USENIX FAST*, Feb 2012.

[16] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. USENIX FAST*, Feb 2008.