MANCHESTER
1824
The University of Manchester

SVD: A Scalable Virtual Machine Disk Format

DOI:
10.1109/TCC.2024.3391390

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Nguetchouang, K., Bitchebe, S., Dubuc, T., Callau-Zori, M., Hubert, C., Olivier, P., & Tchana, A. (2024). SVD: A Scalable Virtual Machine Disk Format. *IEEE Transactions on Cloud Computing*, *12*(2), 684-696. https://doi.org/10.1109/TCC.2024.3391390

Published in:
IEEE Transactions on Cloud Computing

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact openresearch@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:13. May. 2025

SVD: A Scalable Virtual Machine Disk Format

Kevin Nguetchouang*, Stella Bitchebe†, Theophile Dubuc*‡, Mar Callau-Zori‡, Christophe Hubert‡, Pierre Olivier§, Alain Tchana¶

*ENS Lyon †McGill University ‡Outscale SAS §The University of Manchester ¶Grenoble INP UGA

***Abstract*—Contrary to CPU, memory, and network, disk virtualization is peculiar, for which virtualization through direct access is impossible. We study virtual disk utilization in a large-scale public cloud and observe the presence of long snapshot chains, sometimes composed of up to 1,000 files. We then demonstrate, through experimental measurements, that such long chains lead to virtualized storage performance and memory footprint scalability issues. To address these problems, we present SVD, a new virtual disk format. We implemented SVD by extending Qcow2, a popular format, and its Qemu driver. We evaluated our prototype, demonstrating that it brings significant performance enhancements and memory footprint reduction. For example, SVD improves the throughput of RocksDB by about 48% on a snapshot chain of length 500. SVD also reduces the memory footprint by 15x.**

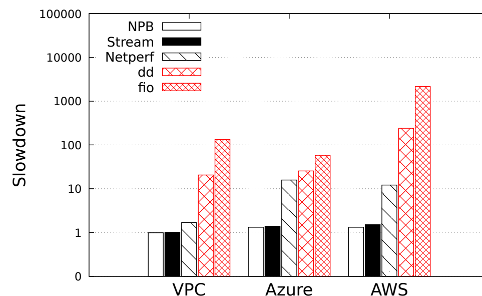


Figure 1: Performance slowdown incurred by virtualization for different types of applications. The results are presented in logarithmic scale. **Lower is better.**

I. Introduction

VIRTUALIZATION is the keystone technology that enables cloud computing. However, virtualization comes at the of overhead on application performance. This overhead has been well studied [4], [14], [18], [30], [42], [25], [1], [13]. Although it concerns all types of resources (CPU, RAM, network, disk), they are not all affected with the same intensity. Figure 1 shows the performance degradation coming from virtualization for a wide range of benchmarks including Stream [15] (memory intensive), NPB [10] (CPU-intensive), netperf [36] (network-intensive), as well as the Linux `dd` command (disk-intensive, throughput-oriented) and fio [20] (disk-intensive, latency-oriented), when they run in AWS EC2 (`t2.medium` instance type), Microsoft Azure (`Standard_B2s` instance type), a virtualized private cloud, and on a bare metal private cloud without virtualization[1]. We use the latter as the baseline. We can observe that the two disk-intensive applications (`dd` and `fio`) experience the highest slowdown. For `fio`, it is about 1,639× the degradation experienced by NPB.

Surprisingly, and contrary to the other resource types, very little research work focus on improving storage virtualization in the cloud. Hence, it is essential to bridge this gap, especially in the context of the exploding popularity of data-centric applications (big data, ML, and AI trends). Storage virtualization is peculiar as it is still implemented through complex multi-layered architectures [19], [45] (see §II). Further, disk virtualization generally uses complex virtual disk formats (Qcow2, QED, FVD, VDI, VMDK, VHD, EBS, and so on). These not only perform the task of multiplexing the physical disk into virtual ones, but also need to support standard features such as snapshots/rollbacks, compression, encryption, etc. All these layers of indirection are the source of the disk virtualization overheads.

This paper studies Linux-KVM/Qemu (hereafter LKQ), a very popular virtualization stack. LKQ supports several virtual disk formats, among which Qcow2 [34] is widely adopted in production [26]: all the principal Linux distributions offer Qcow2 cloud images [6], [8], [7], [35], and it is also the main supported format in major cloud computing platforms including OpenStack [29] and Apache CloudStack [3]. A salient feature provided by Qcow2 is the capacity to create incremental Copy-On-Write (COW) snapshots (backing files) in order to save the state of the virtual disk at a given point in time and to reduce storage space usage. The virtual disk of a VM can thus be seen as a chain linking multiple backing files. This paper identifies and solves scalability issues on such snapshot chains.

Our *first contribution* (§III) is the presentation of the characterization of chain length in the infrastructure of our cloud partner, a large-scale public cloud provider with several datacenters spread over the world, using LKQ/Qcow2 for virtualized storage. We observe that snapshot operations are frequent (some VMs are subject to more than one snapshot creation per day) for three main reasons. First, cloud users leverage snapshots to create recovery points for fault tolerance reasons periodically. Second, cloud users and providers use snapshots to achieve efficient virtual disk copy operations and share elements, such as the OS/distribution base image between several virtual disks. Third, cloud providers use the snapshot feature to transparently distribute a virtual disk made of multiple chained backing files, among several storage servers, for load-balancing and capacity reasons and to avoid

[1] We chose `t2.medium` and `Standard_B2s` to match the VM size that we used in our private cloud.

fragmentation. For all these reasons, we observe that the length of a chain can be very high. We identify chains composed of up to $1,000$ backing files. To our knowledge, this is the first paper performing such characterization. Prior works [9], [37], [2] mainly focused on the characterization of virtualized CPU and memory utilization in the cloud and there is also a patent submitted in 2015 but being extend without convincing solution nowadays [41].

Our *second contribution* (§IV) is to show by experimentation that long chains lead to performance and memory footprint scalability issues. For illustration, using a synthetic benchmark based on `dd`, we measured up to 91% of IO throughput decrease and up to 180× memory footprint increase for a chain composed of $1,000$ backing files. We found that the origin of this problem lies in the fundamental design of the Qcow2 format: the Qcow2 driver in Qemu manages each backing file *individually in a recursive fashion*, without a global view of the entire chain composing the virtual disk. The evaluations of other formats (Microsoft Hyper-V's VHDX, VMWare's VMDK, and IBM's FVD) show that they use a similar approach, thus suffering from the same issue.

Our *third contribution* (§V), called SVD, is to address these scalability issues by evolving the virtual disk format and introducing two principles: 1) direct access upon an I/O request, regardless of their position in the chain; 2) using a single metadata cache, avoiding memory duplication by being independent of the chain length. The implementation of these principles raises three challenges. First, we should allow backward compatibility, which is essential to facilitate cloud operators' adoption of our solution. Second, we should preserve all Qcow2 features. Third, we should preserve crucial optimizations such as prefetching that come naturally with the current Qcow2 format. To cope with the above challenges, SVD extends the Qcow2[2] format to indicate the backing file which contains each cluster of the virtual disk. We rely on reserved bits in Qcow2's metadata to preserve backward compatibility. We implement these principles by extending, on the one hand, the Qemu's Qcow2 driver and, on the other hand, the snapshot operation. We thoroughly evaluate our prototype and demonstrate that it tackles Qcow2's aforementioned scalability issues. For example, on a virtual disk backed up by a chain of 500 snapshots, RocksDB's throughput is increased by 48% versus vanilla Qemu. The memory overhead on that chain is also reduced by 15×.

Overall we make the following contributions:

- We characterize snapshot length in a large scale cloud provider.
- We assess for the first time performance and memory footprint scalability issues due to long snapshot chain and explain the origin of the problems.
- We design and implement a retro-compatible extension of the Qcow2 format addressing these problems.
- We evaluate a prototype of our solution in various situations, demonstrating the effectiveness of our approach.

[2]We consider the latest version of Qcow2 available at the time of the writing. Note that in some wiki pages [33] the term QcowX could be found. It refers to Qcow2 version X.

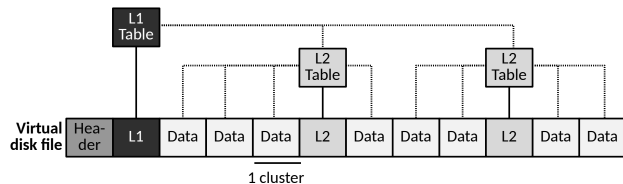


Figure 2: Overview of the Qcow2 format.

The rest of the paper is organized as follows. §II presents the background. §III presents virtual disk characterization results. §IV presents and assesses the scalability issues handled in this paper. §V presents our design to addresses the identified scalability issues. §VI presents the evaluation results of our design. §VII presents the related work. §VIII concludes the paper.

II. Background

The goal of our work is twofold: the characterization of snapshot chains in a public large-scale cloud and handling of two scalability issues happening in long snapshot chains. This section presents the necessary background to understand our contributions.

a) Qcow2 Overview.: The Qcow2 format enables copy-on-write snapshots by using an indexing mechanism implemented in the format and managed at runtime in the Qcow2 driver, running in Qemu, to map guest IO requests addressing virtual sectors/blocks to host offsets in the Qcow2 virtual disk file(s). Figure 2 shows an overview of the Qcow2 format. Without any snapshot, a virtual disk is contained in a single file. The file is divided into units named clusters, containing either metadata (e.g., a header, indexing tables, etc.) or data representing ranges of consecutive sectors. The default cluster size is 64 KB. Indexing is made through a 2-level table, organized as a radix tree: the first-level table (L1) is small and contiguous in the file, while the second-level table (L2) could be spread among multiple non-contiguous clusters. The header occupies cluster 0 at offset 0 in the file, and the L1 tables come right after the header. For performance reasons, the RAM caches L1 and L2 entries (see below).

b) Qcow2 Snapshotting.: Today, the most common way to create a live incremental snapshot of a virtual disk F for a given VM is to create a new empty Qcow2 file E and set it as the current disk (called *active volume*) for the VM while the previous virtual disk F is set as the *backing file* for E. The backing file will be queried for clusters read by the VM that are not present in E. All write operations made by the VM will be directed to the active volume (E), while read operations will be directed either to E if the addressed sectors are present or to backing files if not. Hence, a virtual disk snapshotted several times is composed of an active volume and a chain of backing files per snapshot. With time, backing file chains can become very long (see §III), being sometimes composed of hundreds or even thousands of files.

Our first contribution characterizes backing file chains in our cloud partner's infrastructure. Our second contribution slightly modifies the snapshotting algorithm.

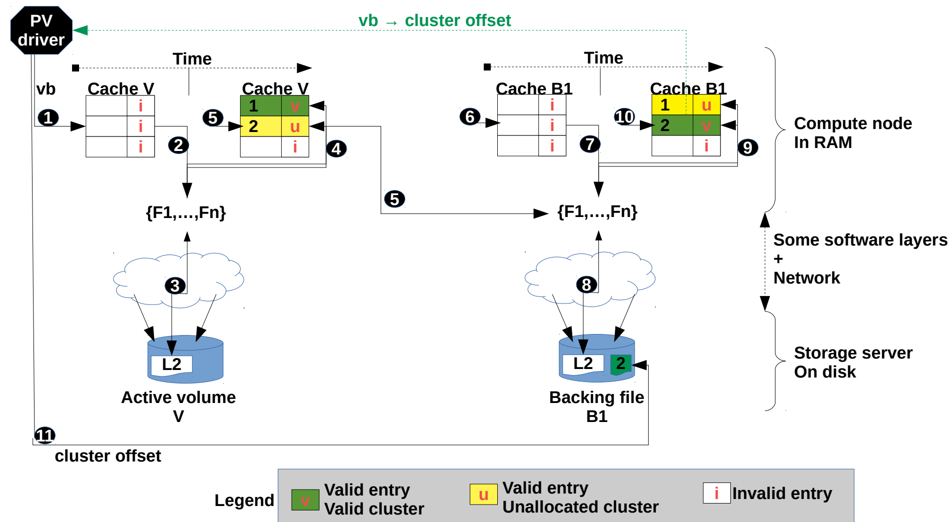


Figure 3: The journey of an IO request. (vb stands for virtual disk block)

c) Qcow2 Cache Organization.: To speed up access to L1 and L2 tables, Qemu caches them in RAM. It creates and manages one cache for the active volume and one cache per backing file. Each cache is managed independently of the others. In the following paragraph, we describe how these caches work. Qemu maintains a separate cache for the L1 and L2 table entries. With its small size, the entire content of L1 is loaded in RAM at VM boot time. The cache of L2 entries is populated on-demand, with a prefetching policy. We, therefore, focus on describing the caching of L2 entries, as they are likely to suffer from misses, thus influencing IO performance. On cache miss, Qemu brings into the cache a set of L2 entries, a *slice* of configurable size, among which the entry at the origin of the miss. The slice is also the granularity of the cache eviction policy, which is LRU. A cache entry includes: the file offset of the slice (noted `l2_slice_offset`), the number of threads which currently use the slice (noted `ref`), the actual L2 entries composing the slice, and a field indicating whether a data cluster referenced by an L2 entry has been modified (noted `dirty`); `l2_slice_offset` services as the tag when searching an entry in the cache, which is fully associative.

d) Qcow2 Cache Utilization.: Every IO request issued by the guest OS to virtual disk vb traps inside Qemu. It is then handled by a thread running the para-virtualized disk driver in Qemu. One of its (driver) main goals is to translate vb to a data cluster offset inside the active volume or a backing file.

From vb, Qemu computes `l2_slice_offset`, `l2_slice_index`, and `l2_index`. It then looks if there is an entry in the cache that matches `l2_slice_offset`. If it exists, Qemu increments the corresponding `ref`. Next, thanks to `l2_slice_index`, it reads the L2 entry. If the latter describes an allocated data cluster (hereafter *cache hit*), then Qemu reads the offset of the data cluster. If the cluster is not allocated (hereafter *cache hit unallocated*) then Qemu considers the cache of the following backing file in the chain. If the slice is not in that cache, Qemu will try to fetch it from the actual backing file associated with the current cache. If the slice exists on disk, it is brought into the cache. Otherwise, Qemu considers the cache of the following backing file and so forth.

Write requests needs additional actions. First, the `dirty` field of the slice is set to 1. If the L2 entry is found in a backing file (not the active volume), Qemu allocates a data cluster on the active volume and performs the copy-on-write. If despite the whole chain scanning, the L2 entry is not found, Qemu creates a new data cluster in the active volume. In any case, Qemu configures L1 and L2 tables accordingly, both on disk and in the active volume's cache. A cache entry can be evicted either when the VM is terminated or when the cache is full.

e) IO Request Journey on a Chain.: Qemu manages a chain snapshot-by-snapshot, starting from the active volume. Figure 3 illustrates the journey of an IO request for a chain of size 2: the base image (B) and the active volume (V). We assume that all L2 indexing caches are empty. We also assume a scenario in which virtual disk files are hosted on an NFS server mounted on a compute node running the VM. Let us assume that cluster number 2 is the target cluster, and it resides in B (meaning that it has not been modified since the creation of V). ❶ The driver starts by parsing V's indexing cache. To handle the cache miss, Qemu performs a set of function calls with some of them ❸ accessing over the network the Qcow2 file to fetch the missed entry from V's L2 table. According to its prefetching feature, Qemu fetches a slice of L2 table entries from V, and ❹ fills V indexing cache. In Figure 3, we assume that the size of a slice is two entries. Thus, V's cache includes two valid entries at the end of the first cache miss handling process: cluster 1 and cluster 2. After this step, ❺ PV driver hits V's cache, but the state of cluster 2 is marked unallocated because the referenced data cluster resides on B. ❻ This cache hit unallocated event triggers the same Qemu functions used for handling a cache miss. Qemu moves to the parent snapshot (B) for cache hit unallocated events. In fact,

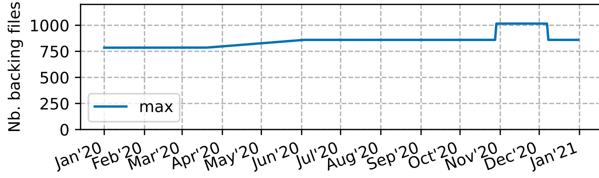


Figure 4: Evolution of the longest chain's size over the year 2020.

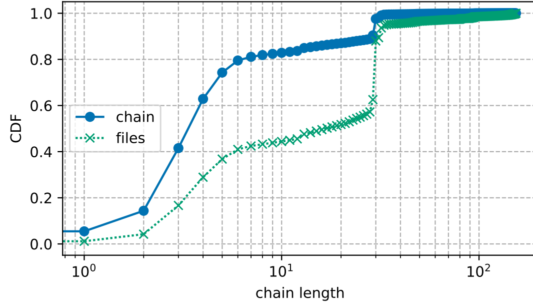


Figure 5: CDF of the chain length for a daily measurement.

at VM startup, Qemu initializes a linked list corresponding to the snapshot chain of the VM's virtual disk. The caches of all the snapshots are also initialized at that time. ❻The first access to B's cache generates a miss ❼. After handling this miss (❽-❿), the offset of cluster 2 is returned to the driver. From there, ⓫ the latter can issue the IO request (consider read here).

III. Chain Length Characterization

Over the entire year 2020, we performed a daily measurement of the length of each chain in our cloud industry partner's infrastructure. The study targets a datacenter located in Europe. The number of VMs in 2020 in this region is 2.8 million, corresponding to one VM booted every 12 seconds, demonstrating the large scale of our study. The software and workloads running in the VMs are highly varied. However, it is worth noting that our partner specializes in business-to-business. Hence, the VMs run enterprise workloads as opposed to private individual ones. Similar to existing cloud providers, the region runs VMs internal to our partner in addition to client VMs. Qcow2 chains back up the VMs' virtual disks in the region.

a) Chain Length.: Our study covers a vast dataset, with the number of daily chains considered in the hundreds of thousands. Figure 4 presents the evolution of the longest chain's length over the period. As we can see, there is always a chain with at least a length of 800 snapshots, and the longest chain can have a length of more than 1,000.

We studied in details a daily measurement made during the period when the longest chain was of a length greater than 1000. Figure 5 shows the CDF for chains and files (active volumes and backing files) with respect to the chain length (for a file, to the length of the chain it belongs to). Most chains are relatively slight: chains of length ten or lower represent nearly 50% of the total number of files, and more than 80% of the chains, in the platform. We can observe a jump around size 30, with chains of size 30-35 files representing a relatively large proportion: 10% of the chains and 25% of the files. This is because, for a subset of the chains, the backing file merging operation, named streaming, is triggered around size 30. That operation merges the layers corresponding to multiple backing files into one. The files that can be merged in this way correspond to unneeded snapshots, i.e., deleted client snapshots and the ones made by the provider. Streaming helps reduce the size of some chains, however, note that valid (non-deleted) client snapshots cannot be merged. Further, although they are infrequent, there is a non-negligible number of chains of size 100 and above.

Take-away 1: *Long chains, with up to 1,000 backing files, do exist. The chain size threshold triggering streaming will cap the maximum size of many chains in the infrastructure.*

b) Snapshot Creation Frequency.: We investigated the frequency of snapshot (i.e., backing files) creation. We looked in our daily measurement, for each snapshot creation operation, the time elapsed since the creation of the previous link in the chain (either a backing file or the active volume for a first snapshot).

We observed that most of the snapshots have chains of size inferior to 30. This is due to the high number of chains of these sizes, stemming from the streaming threshold set to 30. Further, although the frequency of snapshot creation is overall highly variable, many snapshots are created with a relatively high frequency (daily or more). Past work [31] noted peaks at up to 58 snapshots per hour. One can also observe that the long chains result from relatively frequent (daily/weekly) snapshotting done by clients (i.e., non-mergeable through streaming).

Take-away 2: *Although the snapshot-creating frequency varies widely among chains, a non-negligible amount of chains experience high frequency snapshotting. Long chains belong to this subset, with daily/weekly snapshots created. These snapshots are made by clients and cannot be merged with streaming.*

c) Origins of Long Snapshot Chains: The emergence of long snapshot chains in modern virtualized environments is due to a combination of factors. First, for data backup/fault tolerance purposes, most cloud providers offer the client the possibility to create disk snapshots regularly, for example, every 24 hours or on-demand through an API. The chain length will thus grow according to the snapshot frequency. Even when the client deletes certain snapshots, they are kept by the provider as they form a necessary part of the Qcow2 chain backing the disk the VM in question is currently using. Second, snapshots may be performed by the cloud provider itself due to thin provisioning strategies; Virtual disk space being allocated on-demand, a disk may grow above the boundaries of the physical disk storing it. In addition, combined with distributed storage, a snapshot allows the virtual disk to transparently continue to grow on another physical disk

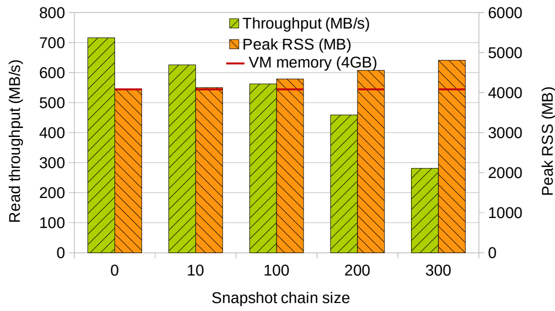


Figure 6: I/O performance and memory footprint evolution with snapshot chain size.

without data transfer. Although they are not visible to the client, such snapshots will be placed in the chains in the same way as the client-made snapshots and will participate in the chain's size increase.

As mentioned above, reducing chain size with streaming has a limited effect because the cloud provider has no control over the client-made snapshots. Furthermore, streaming impacts guest I/O performance: we measured the disk latency from the guest with `ioping` on a standard SSD (WD Blue) and noted a 100x increase during streaming. Streaming can be quite long according to the size of the merged snapshots, and a streaming operation needs to abort if the client decides to reboot/halt the VM.

Take-away 3: *Long chains are due to the client- as well as provider-made snapshots, and to the limitations of the methods (e.g., streaming) to reduce their length.*

IV. Problem With Long Snapshot Chains

A. Problem Statement

From the illustration presented in Figure 3, the reader can intuitively see the two scalability issues posed by Qcow2 for long chains. The first is the memory footprint increase caused by L2 entry duplication in indexing caches. In Figure 3, clusters 1 and 2 are present in the two indexing caches. The second consequence is the negative impact on IO request latency. We can formalize the average cache miss cost (Y) using this equation:

$$Y = [(Hit_{\%} \times T_M) + (Miss_{\%} \times (T_D + T_L + T_F)) + (UnAl_{\%} \times T_F)] \times N \quad (1)$$

where T_M is the RAM access time (about 100ns), T_D is the disk access time (about 80μs), T_L is the time to traverse all software and network layers (about 5μs), T_F is the time to traverse only the software (about 1μs), N is the chain length, $Hit_{\%}$, $Miss_{\%}$, and $UnAl_{\%}$ are respectively the hit, miss and unallocated events ratios. Because T_D, T_L, and T_F are too high compared to T_M, even a tiny miss and unallocated ratio will lead to significant performance degradation [38]. Long snapshot chains exacerbate this degradation.

B. Assessment

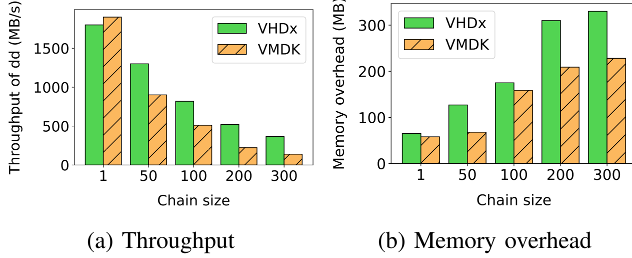
A VM running on a long Qcow2 snapshot chain sees its performance and memory footprint seriously impacted. To demonstrate these points, Figure 6 shows the evolution of these two metrics for a VM running on a virtual disk with variable chain sizes, ranging from 0 to 300 snapshots. The total virtual disk size is 20 GB and each snapshot contains an incremental layer of 60 MB. All files reside locally on the host's SSD. The VM has 4 GB of allocated RAM, 4 vCPUs and runs Ubuntu 18.04. The read throughput is measured within the VM by reading the entire disk with `dd` right after 1) a first call to `dd` on the entire disk to ensure L1/L2 caches are fully populated and 2) a guest page cache drop to assure that the Qcow2 file is accessed. The memory footprint is measured from the host as the hypervisor's peak Resident Set Size (RSS) observed during the execution of the `dd` command.

As one can observe, although with small chains the read throughput is not substantially impacted, when the chain size grows performance drops significantly. On a virtual disk with a chain size of 300, the read throughput only reaches 39 % of what can be achieved on a disk with no snapshots. Regarding memory consumption, with no or a few snapshots the memory overhead that Qemu presents on top of the 4 GB used by the VM is negligible. However, with long snapshot chains that overhead becomes significant: with 300 snapshots, 711 MB of additional RAM are consumed by Qemu. We used the `massif` heap profiler of Valgrind to investigate memory consumption and discovered that the memory footprint increase is due to various data structures that are allocated on a per-snapshot basis. The main culprit for the high memory consumption with long chains is the L2 indexing cache. There is one Qcow2 driver instance running in the hypervisor for each Qcow2 snapshot in a chain. Although the maximum L2 cache size defaults to 1 MB [12], in our experiment we set it to ≈2.7 MB which is the maximum value to manage all the cache of a 20 GB disk – setting it lower seriously impacts performance. However, because there is one cache per driver instance and one instance per snapshot, one can conclude that the cache-related memory footprint increases linearly with the number of snapshots in the chain. We also profiled the Qemu hypervisor from the host during the execution of the aforementioned `dd` test on the 300 snapshots-long case and found that the guest only executes for 7 % of the time. Qemu's disk driver threads consume the remaining time.

These numbers were gathered on Qemu 4.2 but we also confirmed this behavior on a very recent (v6.0) version. We focus on 4.2 in the rest of this paper as it is the version used by our cloud provider partner. Although this version may seem outdated, cloud providers notoriously use old software versions with backported security updates (i.e. long-term support) for obvious stability reasons.

a) Other formats: We also evaluated other popular virtual disk formats, including Hyper-V's VHDX [24] and VMWare's VMDK [40]. As shown in Figure 7, these formats suffer from the same scalability problems. problems. We notice that the amount of overhead is not the same here as in the Qcow2 format, this is because virtual disk and snapshots management are different in each format but despite this fact, the scalability problem remains present.

Take-away 4: *Long chains lead to memory footprint and IO performance scalability issues.*



(a) Throughput (b) Memory overhead

Figure 7: I/O performance and memory footprint for various disk formats.

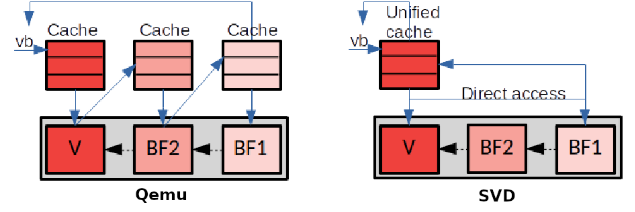


Figure 8: Vanilla Qemu (left) compared to our SVD, which follows two principles: direct access and unified indexing cache.

V. SVD: Scalable Virtual Disk

This section presents SVD, a solution to the two scalability issues identified in the previous sections regarding performance and memory consumption. Ideally, both metrics should be as independent as possible from the length of the backing file chain length.

A. Principles and Challenges

SVD relies on two fundamental principles, illustrated in Figure 8: 1) direct access to on-disk indexing/data clusters, regardless of their position in the chain, upon guest I/O requests; 2) using a single unified indexing cache, avoiding duplication of cache entries by being independent of the chain length. In the rest of the document, we note *vQemu* and *vQcow2* respectively *vanilla Qemu* and *Qcow2* current format. We apply the first principle through a backward-compatible modification of vQcow2, requiring the storage of additional metadata in virtual disk images and an update to the Qcow2 driver in the Qemu storage stack. Applying the second principle only requires a careful modification of the Qcow2 driver.

A significant challenge the implementation of SVD faces is its transparent and fast integration within the infrastructure of our cloud partner (and within cloud infrastructures in general). Our solution should be compatible with the different backends that can hold disk backing files in today's cloud infrastructure. These can be stored directly on the host disk, accessed by the host through the network, and served by centralized NFS servers or distributed file systems. Hence, we propose to modify a popular existing disk format rather than propose a new one [38]. A related challenge is also backward compatibility. Existing Qcow2 images lacking our format's metadata should still work with our updated version of Qemu (without performance/memory consumption gains on long chains). In addition, images using our format should work with vQemu that do not run our updated Qcow2 driver (once again without gains on long chains). Alternatively, vanilla disk images can be easily converted to our format to benefit from the performance/memory footprint enhancement on long chains.

B. Format Improvement

When a guest issues an IO request, vQemu sequentially scans the active volume and all the backing files in the chain until the proper one is found, which is inefficient. Our design eliminates that chain scanning operation. We introduce new metadata in the format indicating, for each data cluster, the backing file that contains the latest (i.e., valid) version of the cluster. We call this metadata the `backing_file_index`. We leverage unused bits in L2 table entries to store that information.More precisely, we use 16 bits to encode `backing_file_index` in each L2 entry.

C. Unified Cache and Direct Access

With direct access, we maintain a single unified cache for the entire disk, independently of the length of the backing file chain. Our cache has the same organization as the vanilla Qcow2 cache presented in Section II. As a reminder, a cache entry corresponds to a slice and contains: `l2_slice_offset` (playing the role of the tag), `ref`, `dirty`, and the L2 entries composing the slice. As noted in the previous section, in SVD a L2 entry contains `backing_file_index` in addition to the default vQcow2 values.

Contrary to the vanilla version where `l2_slice_offset` was specific to each backing file, in our version, `l2_slice_offset` is related to the active volume. In addition, one can find L2 entries describing data clusters belonging to distinct backing files in the same slice. Therefore, the read and write operations are performed as follows in SVD. Let us consider vb, the offset of a virtual block that the guest wishes to read. Using the same functions a vQemu, SVD computes `l2_slice_offset`, `l2_slice_index` and `l2_index`. If both the slice and the L2 entry exist in the unified cache and that `backing_file_index` contained in the L2 entry corresponds to the active volume, there is a cache hit and the offset of the cluster data to be read is in the L2 entry. If `backing_file_index` does not correspond to the active volume, this is a cache hit unallocated. SVD locates on disk the backing file corresponding to `backing_file_index` and reads from it the slice at offset `l2_slice_offset`. Let s_b be that slice and s_v be the slice currently contained in the unified cache. SVD traverses all s_b entries and updates the L2 entries in s_v with the corresponding contents in s_b under the following condition: the value of `backing_file_index` of the L2 entry in s_v is lower or equal to that of `backing_file_index` of the L2 entry in s_b. We call "*cache correction*" these replacement operations. Then it sets `dirty` to 1 in s_v, so the slice will be written to

disk when it is evicted from the cache. If the L2 entry does not exist in the slice, there is a cache miss, and the entry needs to be allocated as in vQemu. This means the guest is asking for a data cluster that does not yet exist on the virtual disk. If the slice is not yet present in the cache, there is a cache miss, and the slice is either fetched from the active volume if it exists, or allocated if not. These operations are similar to vQemu. It is important to note that cache correction is executed once when we resolve a cache miss for a data cluster and is no longer executed till the data cluster is evicted from the cache and needs to be prefetched from the image disk later.

D. Snapshotting

In vQemu, a new Qcow2 active volume is created on snapshot creation, with very little information (the header, the L1 table, and refcounts). We updated the snapshot creation logic to copy the entire content of both L1 and L2 tables from the previous active volume to the newly created active volume, now a backing file. The algorithm that we implement is as follows. Let `new_volume` be the file that will become the new active volume and `old_volume` the old one. We intervene in the creation of the L1 table. Recall that it is always located at the second cluster of any Qcow2 file. Let `new_l1` be the new L1 table and `old_l1` the L1 table of `old_volume`. After the allocation of `new_l1`, we parse all the `old_l1` entries. For each entry we create the corresponding L2 table in `new_volume`, then we set the current `new_l1` entry with the offset of that L2 table. Let `new_l2` be that new L2 table and `old_l2` be the L2 table pointed to by `old_l1` in `old_volume`. Then we copy the whole content of `old_l2` to `new_l2`.

Consequently, a new active volume always contains all L2 tables of the previous backing files. The copy of L2 tables may lengthen disk snapshotting time compared to the vanilla version. We could have implemented a copy-on-demand solution. However, that would mean impacting the critical path of I/O requests. This approach would increase tail latency, requiring chain scanning to find the valid backing file. The evaluation results show that the disturbance brought by the snapshot operation upon guest I/O performance is mainly acceptable, as the total size of L2 tables is, in the worst case, in the order of MB. In addition, we believe that VM owners are likely to accept the small price of a slight increase in snapshotting time, to benefit from an essential boost in I/O performance.

VI. Evaluation

Here we present an evaluation of SVD, aiming to answer the following three questions:

Q_1) Does SVD eliminate the memory footprint scalability issue of Qemu? (§VI-B)

Q_2) Does SVD eliminate the IO performance scalability issue of Qemu? (§VI-C-VI-D)

Q_3) To what extent does SVD increases snapshotting time and disk overhead? (§VI-E)

A. Evaluation Setup

a) Methodology.: We systematically compare SVD with Qemu. We limited our comparison to only Qemu because its current version embeds the optimizations that prior academic works presented. We evaluate several configurations by varying three parameters: the chain length (1-1,000); the virtual disk size (50GB, 150GB); as well as the cache size (from 30 % to 100 % of the cache size needed to hold the entirety of L2 entries to index a full disk, i.e., from 1.9 MB to 6.25 MB for a 50GB disk size, and from 5.6 MB to 18.75 MB for 150 GB). For all experiments, we uniformly distribute valid clusters on the backing files of the disk's chain; meaning that all clusters are equally likely to be accessed. The virtual disk is populated at 90 % with random data for experiments with micro-benchmarks using the Linux `dd` command, and at 25 % for experiments with macro-benchmarks using the RocksDB client [11]. The release of SVD includes a highly configurable chain generation script.

Unless otherwise indicated, the size of the L2 cache is set so that it can hold all L2 entries to index the entire disk. All results presented in this section are an average value of 5 runs.

b) Testbed.: To have a representative test environment, we employ two servers: the compute node running VMs and the storage node holding virtual disk files. Each server has 32 Intel Xeon Gold CPU cores, clocked at 2.10GHz, 192 GB of RAM, Samsung
MZ7KM480HMHQ0D3 SATA SSD. They are linked with a 10Gbps Ethernet connection. The storage node serves the virtual disk files through NFS. Both servers run Debian 10 with Linux 4.19.0 as the host OS. All VMs run Ubuntu 18.04 with Linux 4.15.0 and are configured with 4GB of memory and four vCPUs. Unless otherwise indicated, the virtual disk size is 50GB.

c) Metrics and Benchmarks.: We collect two types of metrics, *high-level* and *low-level* metrics. The former directly impact the end-user's perceived Quality of Service. We consider VM startup time, memory overhead, application execution time, and I/O disk throughput. The memory overhead is the additional memory consumed by Qemu on top of the VM's allocated pseudo-physical memory. Low-level metrics represent internal costs that help explain high-level metrics. They are: the total number of cache misses, the number of cache hit unallocated, and the cache lookup latency. The lookup latency is the time to find a data cluster's valid offset in the caching system. Storage benchmarks are run in the guests. We use microbenchmarks, including Linux `dd` (which sequentially read the entire disk from the guest i.e. `dd if=/dev/sda of=/dev/null bs=4M`) and fio [20] (70% random access, half reads and half writes, with iodepth of 32 and with direct write), as well as macro-benchmarks, RocksDB-YCSB [11] and a measurement of the VM boot time.

For the rest of this section, when we talk about snapshots, it will be a chain of snapshots referencing each other with the data evenly distributed over all the snapshots, like the users (or the cloud provider) who would take snapshots regularly. Another difficult choice made in this section was the choice of the length of snapshots (50 and 500). It was not significant to

make the experiments with all possible chain lengths. So we focus on two lengths. The first one (50 snapshots in a chain) used to represent the common length in cloud environment validated by our cloud provider partner and the former one (500 snapshots) used to represent the worst case which doesn't appear that much but when it appears, is very impacting from the user point of view.

d) Note: It is important to note that most of the benchmarks are read-intensive because Qemu bottlenecks only appear when we have to fetch data in the whole chain, while write operations are always done in the active snapshot (the last one in the chain).

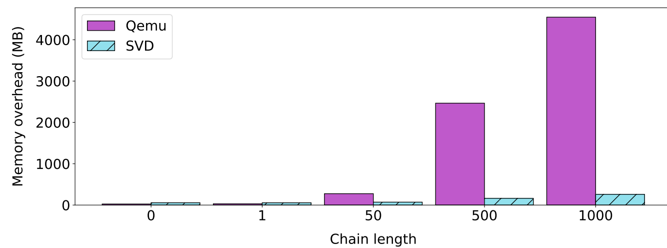


Figure 9: Memory overhead of SVD and Qemu after reading the entire disk from the guest with `dd`, while varying chain length. **Lower is better.**

B. (Q_1) Memory overhead

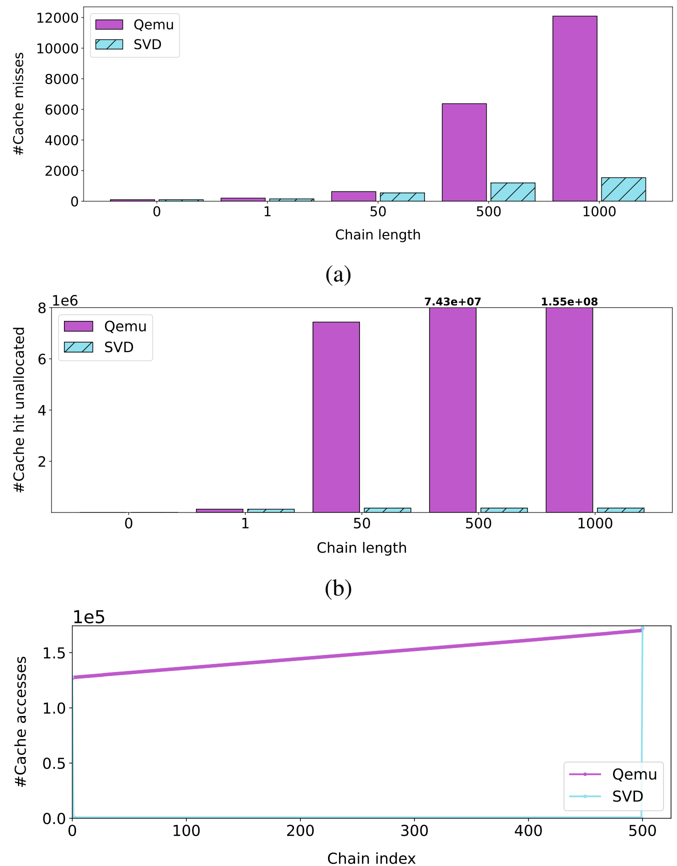
For this experiment, we measured Qemu's resident set size after having read the entire disk from the guest using `dd` and subtracted from this measurement the amount of RAM given to the VM (4GB) to compute Qemu's memory overhead. Figure 9 shows the results. One can observe that SVD significantly reduces the memory overhead when the chain length increases. The memory savings are as follows: 205 MB for a chain length of 50 (3.9× reduction), 2303 MB for a length of 500 (15.2×), and 4289 MB for a length of 1,000 (17.6×). Although it scales much better than vanilla Qemu, SVD's memory overhead slightly increases with the chain size. This is due to other per-snapshot data structures in Qemu that are not directly related to the caches. Finally, note that SVD comes at the cost of a slight memory footprint increase over vanilla when the disk has no or a tiny number of snapshots – a cost that is amortized by the better scalability starting from 5 snapshots.

C. (Q_2) Low-level Metrics

We use the same setup as in the previous section.

a) Cache Misses and Cache Hit Unallocated.: We instrumented SVD and vanilla Qemu to measure the number of cache misses, the number of cache hits unallocated, and the number of cache accesses per backing file of the chain. Figure 10 shows the results.

We can see that SVD leads to fewer cache misses compared to Qemu, as Figure 10a shows. We measure up to 10× for chain length 1,000. This difference is explained by the fact that Qemu does not implement a cache correction mechanism as we do in SVD (see §V). Therefore, when an L2 entry is only present in the cache of the backing file of index m in the chain, Qemu will generate $n - m + 1$ cache misses walking the chain to get it, where n is the chain length.



(c) Chain length of 500 snapshots.

Figure 10: During an entire disk read with `dd`, the number of (a) cache misses, (b) the number of cache hit unallocated, and (c) the distribution of cache lookups according to which backing file holds the addressed data. **Lower is better.**

Concerning the number of cache hits unallocated, it is constant under SVD, see Figure 10b. The increase compared to a virtual disk composed of a single active volume (chain length 1) is less than 1% for the chain length 1,000. Concerning Qemu, the number of cache hit unallocated increases $10,000,000\times$ for the chain length 1,000; $4,000,000\times$ for the chain length 500 and $\approx 600,000\times$ for the regular chain of length 50. This is once again explained by the fact that Qemu looks up several caches during the chain walk.

In the experiment with a chain of length 500, we count the total number of cache lookups and plot their distribution, according to which backing file in the chain holds the requested data, in Figure 10c. Due to the chain walks, caches are much more frequently accessed under Qemu compared to SVD. The gap is about 1,500%. The spike that appears for backing file zero, the base virtual disk image, corresponds to the boot of the VM. In fact, during that time, several IO read requests are performed on read-only files (such as vmlinuz). The spike on snapshot 500 corresponds to the accesses made on the active volume.

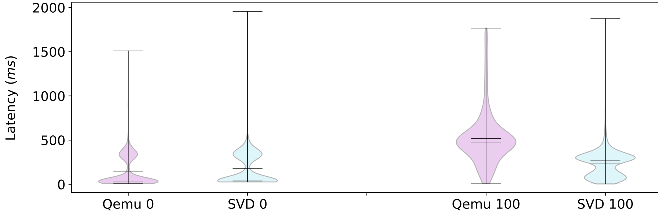


Figure 11: Cache lookup latency distribution. **Lower is better.**

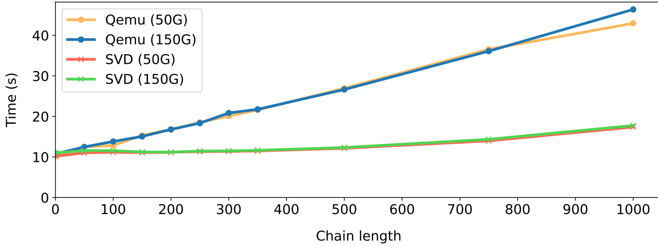


Figure 12: VM Boot Time. **Lower is better.**

b) Cache Lookup Latency.: We measured the cache lookup latency on two chain lengths: 1 and 100. Figure 11 presents the distribution of cache lookup latencies for all IO requests performed during the execution of the `dd` benchmark. We can observe that for both systems, the mean latency value changes according to the chain length. However, SVD leads to a better latency than Qemu when the chain length increases: the mean latency is 490 ms under Qemu and 270 ms with SVD, i.e., 1.8x faster. Contrary to Qemu, latency values under SVD are located around two mean values, 120 ms and 270 ms. 120 ms corresponds to the cache hit mean latency, while 270 ms corresponds to the cache hit unallocated mean latency. Theoretically, according to the direct access principle implemented by SVD, only one value of cache hit unallocated latency can be observed compared to Qemu. We do not observe the same kind of distribution under Qemu because, in this experiment, data clusters are uniformly distributed over all backing files. Therefore, most IO operations lead to a variable amount of cache hits unallocated, i.e., chain walks of variable length, according to the target data location in the chain. This translates into highly variable and, on average higher latencies in Qemu.

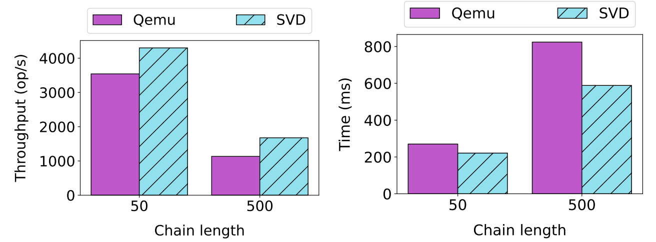
Concerning the variance of latencies value it appears that Qemu has a higher value than SVD. For the chain length of 1 snapshot, the variance is quite the same (≈ 10) because there are not many backing files and many caches to go through. But when we go up to a chain of 100 snapshots, the variance latency value of Qemu (17) is higher than SVD's value (12) while in Qemu we go through all the caches of all snapshots, we have more diverse values of latencies where we go through essential and the same caches with SVD and then we have fewer latency values to process the variance.

D. (Q_2) High-level Metrics

1) Macro-benchmarks:

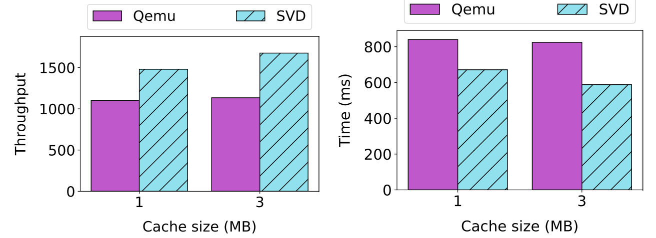
a) VM Boot Time.: VM boot time is a critical metric in the cloud [22], [27]. Figure 12 compares the time it takes to boot a VM under SVD and Qemu while varying the chain length and the virtual disk size. The boot time increases rapidly with the chain length under Qemu: it goes from about 10 seconds on a chain of size 1 to more than 40 seconds ($4\times$) on a chain of size 1000. On the contrary, with SVD that increase is moderate: from 10 seconds to 17 seconds ($1.7\times$).

The increase in boot time for SVD can be explained by the slight increase in the number of cache misses and cache hit unallocated discussed above. We can see that the size of the virtual disk has a negligible impact on the results.



(a) Throughput, with 3MB cache size. **Higher is better.**

(b) Execution time, with 3MB cache size. **Lower is better.**



(c) Throughput, with chain length 500. **Higher is better.**

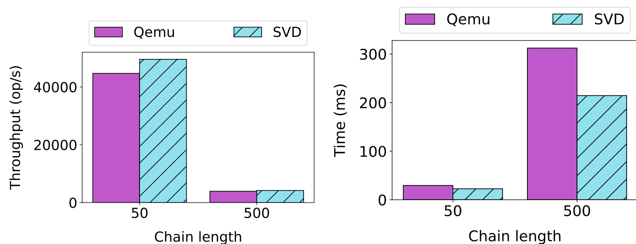
(d) Execution time, with chain length 500. **Lower is better.**

Figure 13: RocksDB-YCSB results for YCSB-C.

b) Cloud Workload: RocksDB-YCSB.: We created a RocksDB database that fills 40% of the VM disk size, populated using the YCSB client, generating a uniform distribution of valid clusters on the Qcow2 chains generated. We used three YCSB workloads: YCSB-C, which simulates a user performing read-only requests; YCSB-B, which simulates a user performing a mix of write and read requests but most reads; and YCSB-D, which realizes more write vs. read requests We experimented two L2 cache sizes (1 MB and 3 MB) and two chain lengths (50 and 500 snapshots). We measured the throughput and execution time (RocksDB's two performance metrics) of YCSB for a total of 500K requests.

Figures 13a and 13c show the results for the throughput metric. Even if the performance of both versions decreases when the length of the chain increases, SVD still outperforms Qemu for both chain lengths (33% for length 50 and 47% for length 500). Further, with a fixed chain length at 500, the throughput of YCSB is almost constant while varying the cache size, regardless of the Qemu system.

Figures 13b and 13d present the execution time results. As for the throughput, SVD improves Qemu. Considering a chain of 50 backing files, SVD reduces the execution time of YCSB by 36% for 1MB cache size and 22% for 3MB cache size. For a chain of 500 snapshots, the improvement is about 40%



(a) Throughput, with 3MB cache size. **Higher is better.**

(b) Execution time, with 3MB cache size. **Lower is better.**

Figure 14: RocksDB-YCSB results for YCSB-D.

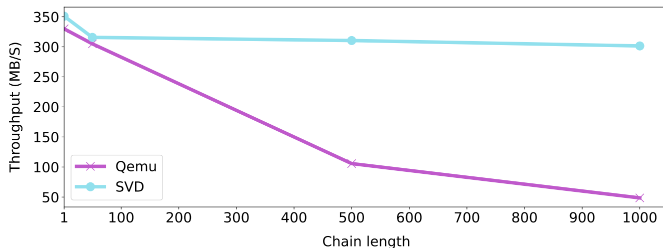


Figure 15: Throughput of Linux dd under SVD and Qemu with various chain length **Higher is better.**

with 1MB of cache size and 36% with 3MB cache size. For throughput and execution time, the improvement of SVD over Qemu is more significant when the chain length increases.

Figure 14 shows that even when there are more write requests (YCSB-D), as soon as the system uses read requests, performance are impacted.

c) Performance improvement status: It's important to recall that Rocksdb is a key-value store based on the log-structured merge-tree data structure so the data are more often kept in memory so it's almost normal we don't have the same gap of performance improvement compared to another storage. But despite this fact, there is still an overall non-negligible improvement from our SVD solution when dealing with heavy usage in cloud provider infrastructures.

2) Micro-benchmarks:

a) Disk Throughput: Linux `dd`*.:* The throughput of `dd` is presented in Figure 15 for both systems managing chains of various sizes. We can observe no degradation under SVD while Qemu severely degrades the throughput of `dd` when the number of backing files increases. Qemu incurs a slowdown of up to 84% for the chain length 1,000.

b) Impact of the Cache Size with fio.: We studied the effect of varying the cache size for SVD and Qemu. In this experiment, we use a chain of length 500 and set the total cache size used by Qemu to equal that used by SVD. Because Qemu uses one cache per layer in the chain, when SVD is given cache size of S, Qemu would get S/L with L being the chain length. We vary the cache size given to each system from 1MB to 4GB and measure the disk read throughput with fio performing random reads of small size (4 KB) on the disk node in `/dev` on one side and on another size random reads and few write.

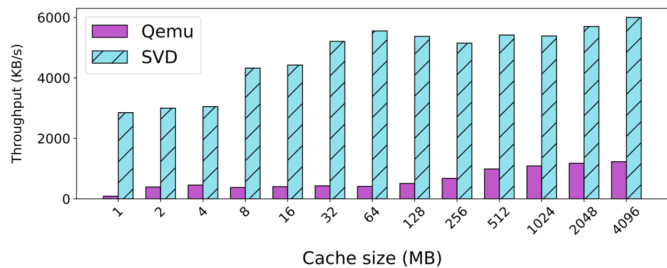


Figure 16: `fio` throughput while varying the cache size. (100% reads) Chain length of 500 snapshots. **Higher is better.**

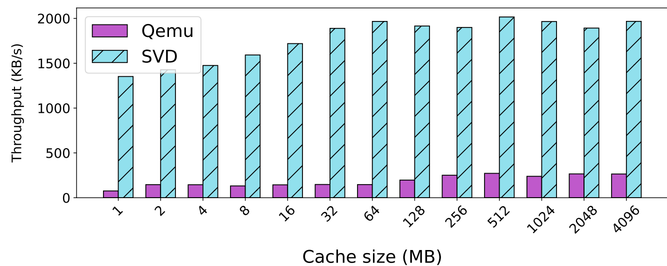


Figure 17: `fio` throughput while varying the cache size. (70% read - 30% write) Chain length of 500 snapshots **Higher is better.**

Figure 16 shows the results. We can observe that SVD significantly outperforms Qemu in all cases. With both systems, performance is sensitive to the cache size. Concerning Qemu, performance steadily increases up to 4 GB of cache. This is due to the large amount of memory required by this multi-cache solution. Regarding SVD, although peak performance is also achieved at 4 GB (6 MB/s vs. 2.5 MB/s for 1 MB of cache), from 32 MB, the payback from adding more cache size diminishes significantly. This value thus represents an excellent trade-off between near-peak performance and memory footprint. This demonstrates the high efficiency of SVD vs. Qemu.

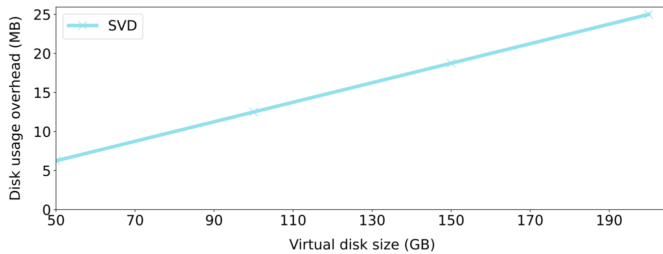
Figure 17 shows similar results when operations are mixed between reads and writes with more reads. Here, performance gains from SVD are lower than Qemu's because write operations do not need to go through the chain, avoiding the performance issue.

In summary, if we want to consider Qemu optimal in term of execution time and throughput, we need to choose the maximum value of cache chose considering the minimum value of cache that can cache all clusters' data metadata which is in the case of this experiment $4GB$ for each 500 snapshots i.e. $\approx 7MB$ per snapshot.

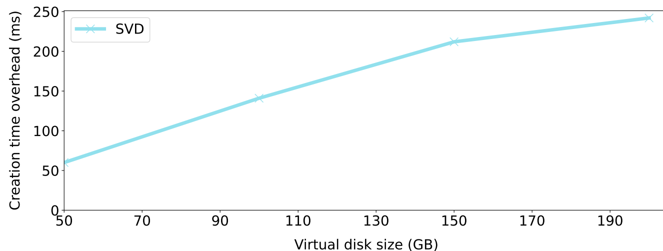
E. (Q_3) Overhead

As stated in §V-D, when creating a snapshot under SVD, L2 tables are copied to the newly created file. This may incur two overhead types: disk usage and snapshotting time.

a) Disk space.: The disk space overhead per snapshot depends on both the VM's disk size and cluster size and the number of allocated clusters in the disk. We can model that overhead in the worst-case scenario, i.e., when every cluster



(a) Disk usage overhead per snapshot.



(b) Snapshotting time.

Figure 18: Impact of SVD on snapshotting. **Lower is better.**

is allocated (the disk is full), as follows: given S_{SQ} and S_{VQ} being respectively the size of a newly created (i.e. empty) snapshot under SVD and Qemu, we compute the disk size of S_{SQ} using the following formula:

$$S_{SQ} = S_{VQ} + \frac{VM_disk_size}{cluster_size} \times L2_entry_size \quad (2)$$

By default, an L2 entry is 8 B, a cluster is 64 KB, and S_{VQ} is 256 KB. Using the above formula, we can compute S_{SQ} while varying the VM disk size from 50 GB to 200 GB. Figure 18a shows that per-snapshot overhead. It increases linearly with the size of the VM disk. To compute the total disk overhead (still in the worst case), the per-snapshot cost needs to be multiplied by the chain length. Recall that from our characterization, we observed that the dominant virtual disk size in the cloud is 50 GB, giving, according to our model, a per-snapshot overhead is about 6 MB. This gives a total overhead, in the worst case, of 60 MB for a chain of length 10 (0.1% of the virtual disk size), 600 MB for length 100 (1.2%), and 6,000 MB for length 1000 (12%).

b) Snapshotting Time.: We measured the time spent to create a new snapshot under SVD and Qemu for different VM disk sizes. The results are presented on Table I Due to the copy of all L2 entries, SVD takes much more time to create a snapshot than Qemu. For a 50GB VM, we need about 70ms to create a snapshot under SVD and 7× less time under Qemu. Furthermore, this overhead increases with the VM disk size. Indeed, for a 200GB VM, the snapshot creation time under SVD is about 12× that of Qemu. Nonetheless, in absolute, the snapshot creation latency is quite low under SVD (in the order of ms). It allows for a relatively high snapshot frequency.

Table I: Snapshot creation time.

VM Disk Size (GB)	**vQemu** Time(ms)	**SVD** Time(ms)
50	10	70
100	15	156
150	21	233
200	23	265

VII. Related Work

a) Virtual Disk Formats.: Fast Virtual Disk [38] is a virtual disk format proposed by IBM in 2011, that increases I/O performance by avoiding the use of a host filesystem, reducing the size of on-disk metadata, and using an on-disk journal. FVD supports only internal snapshots, which means that all the chain is stored in a single file. This may not be as flexible as the external snapshots offered by the format we focus on, Qcow2, for example when subsets of a chain need to be stored on different storage nodes for load-balancing or capacity reasons. It is also unclear how FVD performs on long chains composed of hundreds or thousands of snapshots. As FVD is old, the format QED which is an old version of Qcow2 has take his essential features such as copy-on-read. The system we propose is an evolution of Qcow2 which is backwards compatible with vanilla Qcow2 disk, something that makes adoption much easier versus proposing an entirely new format. Finally, FVD can be considered as deprecated as it was developed for Qemu 0.14, dating from 2011, and has not been ported to modern versions.

Parallax [23] is a distributed architecture storing virtual disk images and using commodity servers as storage backends, as opposed to high-end storage arrays/switches. Among other features, Parallax offers low-overhead and high-frequency snapshots and note, similar to our work, that the performance overhead and memory consumption of traditional formats such as Qcow2 increases with snapshot chain sizes. Similar to FVD, migrating an existing cloud environment to Parallax requires significant changes to the virtualized storage system's architecture, whereas we rely on the widely used Qcow2 format, and are backward-compatible with environments that do not use our system. Further, contrary to our Qcow2 format, Parallax does not support sharing of virtual disk images, a feature heavily used in the industry to lower storage overheads of commonly used volumes e.g. base images.

b) Storage Performance and Availability during VM Migration.: Noting that VM migration significantly disrupts guest I/O performance, A few papers [17], [44] focus on maintaining good storage performance and availability during migration. Netchannel [17] proposes various techniques to maintain local/remote virtual disk availability during migration. One is the ability to seamlessly switch the physical device associated with a virtual one. Another proposed technique is the capacity for migrated VMs initially plugged to a local disk on the host to transparently keep using that disk through a proxy once they are migrated to another host. In another study [44], the author propose to study the storage I/O behavior of guests to infer the most efficient data transfer schedule to reduce disruption as much as possible during VM migration.

c) Scheduling Impact on Virtual Storage Performance.: Several studies [28], [16] noted that VM scheduling could have a non-negligible impact on guest I/O performance. The authors of a study [28] characterize the impact on processor and I/O performance of various VM scheduler configurations, for concurrently-running guest with CPU- and bandwidth-

intensive, as well as latency sensitive behaviors. In another paper [16], the authors propose a guest task-level priority boosting technique to selectively increase the priority of I/O-bound task to increase storage performance while maintaining CPU fairness.

d) Virtualized Storage Performance and Power Consumption.: Other studies focus more generally on virtualized storage performance and power consumption [43], [39]. Ye et al. [43] note that existing power consumption reduction techniques for non-virtualized HDDs do not apply in a virtualized setting. They propose to bridge the semantic gap between VM and VMM through several techniques tailored for such environments, reducing disk spin-ups and increasing disk sleep times, in order to save energy. Another paper [39] focuses on the particular problem of interrupt delivery to VMs, including the ones coming from block devices. The authors propose an optimized interrupt delivery system for KVM. It is mostly evaluated on network workloads but also shows moderate performance improvements on storage workloads.

e) Cloud Storage and File Systems.: Finally, several papers [32], [5], [21] focus on cloud storage and filesystems. The Frugal Cloud File System [32] proposes integrating multiple services (AWS EBS, Azure Cache, etc.) into a single solution that aims to be flexible from the performance and costs point of views. DepSky [5] introduces a cloud-based storage system targeting security/dependability by spreading and replicating storage over multiple clouds. Depot [21] proposes a cloud storage system that can tolerate buggy clients and servers in order to minimize trust assumptions.

VIII. Conclusion

For the first time, we presented the characterization of snapshot chain length in a large-scale public cloud. Among other results, our analysis revealed the presence of long snapshot chains, leading to scalability issues for both memory footprint and performance. We present SVD, a solution to these two issues in the form of a slight extension of the Qcow2 format while preserving backward compatibility. We built SVD following the principles of direct access and single indexing cache, regardless of the chain length. We evaluated SVD extensively and compared it with vanilla Qemu using a wide range of benchmarks, demonstrating that our solution effectively tackles the above issues. For instance, SVD improves the IO throughput of RocksDB by up to 48% compared to Qemu, and reduce the memory footprint by 15x when the chain length is 500.

Acknowledgments

This work was performed within the framework of the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[2] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. *Providing SLOs for Resource-Harvesting VMs in Cloud Platforms*. USENIX Association, USA, 2020.

[3] Apache Foundation. Apache cloudstack - storage overview, 2022. http://docs.cloudstack.apache.org/en/4.15.0.0/adminguide/storage.html.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[5] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *Acm transactions on storage (tos)*, 9(4):1–33, 2013.

[6] Canonical. Ubuntu cloud images, 2022. https://cloud-images.ubuntu.com/.

[7] CentOS Contributors. Centos cloud images, 2022. https://cloud.centos.org/centos/7/images/.

[8] Debian Contributors. Debian official cloud images, 2022. https://cloud.debian.org/images/cloud/.

[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[10] NASA Advanced Supercomputing (NAS) Division. Nas parallel benchmarks. https://www.nas.nasa.gov/software/npb.html.

[11] Facebook. A persistent key-value store for fast storage environments, 2012. https://rocksdb.org/.

[12] Alberto Garcia. Qemu - qcow2 cache documentation, 2018. https://github.com/qemu/qemu/blob/master/docs/qcow2-cache.txt.

[13] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: Bare-metal performance for i/o virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[14] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. *CLOSER*, 11:563–573, 2011.

[15] Mc Calpin John D. Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/.

[16] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, 2009.

[17] Sanjay Kumar and Karsten Schwan. Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 31–40, 2008.

[18] John R Lange, Kevin Pedretti, Peter Dinda, Patrick G Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. *ACM SIGPLAN Notices*, 46(7):169–180, 2011.

[19] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 8, USA, 2012. USENIX Association.

[20] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, 2008.

[21] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):1–38, 2011.

[22] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Dutch T Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J Feeley, Norman C Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 41–54, 2008.

[24] Microsoft Corporation. Virtual hard disk v2 (vhdx) file format, 2018. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-VHDX/%5bMS-VHDX%5d.pdf.

[25] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393. IEEE, 2015.

[26] Eyal Moscovici and Amit Abir. How to handle globally distributed qcow2 chains. KVM Forum, 2017.

[27] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift birth and quick death: Enabling fast parallel guest boot and destruction in the xen hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10, 2008.

[29] OpenStack. Openstack documentation - get images, 2022. https://docs.openstack.org/image-guide/obtain-images.html.

[30] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 137, 2007.

[31] Loïc Perennou, Mar Callau-Zori, Sylvain Lefebvre, and Raka Chiky. Workload characterization for a non-hyperscale public cloud platform. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 409–413. IEEE, 2019.

[32] Krishna PN Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 71–84, 2012.

[33] Qemu. Features/qcow3, 2016. https://wiki.qemu.org/Features/Qcow3.

[34] Qemu Contributors. Qcow2 documentation, 2020. https://github.com/qemu/qemu/blob/master/docs/interop/qcow2.txt.

[35] Red Hat. Preparing and uploading cloud images with image builder, 2022. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/composing_a_customized_rhel_system_image/creating-cloud-images-with-composer_composing-a-customized-rhel-system-image.

[36] Jones Rick. Hewlett-packard netperf benchmark. https://github.com/HewlettPackard/netperf.

[37] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*. USENIX Association, USA, 2020.

[38] Chunqiang Tang. Fvd: A high-performance virtual machine image format for cloud. In *USENIX Annual Technical Conference*, volume 2, 2011.

[39] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. *Acm Sigplan Notices*, 50(7):1–15, 2015.

[40] vmware. Vmdk disk format specification. https://www.vmware.com/app/vmdk/%3Fsrc%3Dvmdk.

[41] VMware. Tracking data of virtual disk snapshots using tree data structures, 2018. https://patents.google.com/patent/US10860560B2/en.

[42] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio, and Steve Gallo. A comparison of virtualization technologies for hpc. In *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pages 861–868. IEEE, 2008.

[43] Lei Ye, Gen Lu, Sushanth Kumar, Chris Gniady, and John H Hartman. Energy-efficient storage in virtual machine environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 75–84, 2010.

[44] Jie Zheng, Tze Sing Eugene Ng, and Kunwadee Sripanidkulchai. Workload-aware live storage migration for clouds. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 133–144, 2011.

[45] Ruijin Zhou, Sankaran Sivathanu, Jinpyo Kim, Bing Tsai, and Tao Li. An end-to-end analysis of file system features on sparse virtual disks. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 231–240, New York, NY, USA, 2014. Association for Computing Machinery.