FVD: a High-Performance Virtual Machine Image Format for Cloud

Chunqiang Tang
IBM T.J. Watson Research Center
ctang@us.ibm.com

Abstract

Fast Virtual Disk (FVD) is a new virtual machine (VM) image format and the corresponding block device driver developed for QEMU. QEMU does I/O emulation for multiple hypervisors, including KVM, Xen-HVM, and VirtualBox. FVD is a holistic solution for both Cloud and non-Cloud environments. Its feature set includes flexible configurability, storage thin provisioning without a host file system, compact image, internal snapshot, encryption, copy-on-write, copy-on-read, and adaptive prefetching. The last two features enable instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. As its name indicates, FVD is fast. Experiments show that the throughput of FVD is 249% higher than that of QCOW2 when using the PostMark benchmark to create files.

1 Introduction

Despite the existence of many popular virtual machine (VM) image formats (e.g., QEMU QCOW2 [5], VirtualBox VDI [10], VMWare VMDK [11], and Microsoft VHD [6]), FVD came out of our unsatisfied needs in the IBM Cloud [9]. FVD distinguishes itself from existing image formats in multiple aspects: *flexible configurability*, *high performance*, and *rich features*.

Flexible configurability. As virtualization becomes pervasive, virtual disks of virtual machines (VM) may be used in diverse settings. For example, the main requirement can be storage thin provisioning or high disk I/O performance. A disk image can be stored as a regular file or a logical volume. It can use direct-attached storage, network-attached storage, or storage-area network.

Existing image formats support diverse settings in a one-size-fit-all manner, by bundling all functions into one inseparable, monolithic piece. This can cause inefficiency even in common cases. Consider, for example, the copy-on-write (CoW) feature of virtual disk. QCOW2, VDI, VMDK, and VHD all mix the function of CoW dirty block tracking with the function of storage space allocation. In a common setting where the image is stored on a host file system, this leads to doing storage allocation twice (first in the image format and then in the host file system), which causes data fragmentation twice and doubles the disk I/O overhead for metadata access.

By contrast, a design principle of FVD is to make all functions orthogonal so that each function can be enabled or disabled individually, even for two virtual disks attached to the same VM. The purpose is to support diverse use cases without being burdened with the overhead of all functions. This is a significant departure from existing image formats and challenges some conventional wisdom in image format design. For the specific example above, FVD can be configured to enable CoW dirty block tracking, disable its own storage allocation, and delegate storage allocation entirely to the host OS, which has abundant options to optimize for a given workload, e.g., storing the image on a logical volume, or storing the image as a regular file in a host file system, with the choices of ext2/ext3/ext4, JFS, XFS, ReiserFS, etc.

VM mobility in a Cloud. FVD is a feature-rich, holistic solution for both Cloud and non-Cloud environments. This paper is focused on its copy-on-read and adaptive prefetching features, which improve VM disk data mobility in a Cloud.

In a Cloud like Amazon EC2, the storage space for a VM can be allocated from multiple sources, which offer different performance, reliability, and availability at different prices. In EC2, a VM is provided with 170GB or more ephemeral storage (i.e., direct-attached storage (DAS)) at no additional charge. Persistent storage (i.e., network-attached storage (NAS)) is more expensive, which is charged not only for the storage space consumed but also for every disk I/O performed. For example, if a VM's root file system is stored on persistent storage, even the VM's disk I/O on its temporary directory */tmp* incurs additional costs. As a result, it is popular to use ephemeral storage for a VM's root file system. However, using DAS slows down the process of VM creation and VM migration, which diminishes the benefits of an elastic Cloud.

The discussion below uses KVM and QEMU as examples. In a Cloud, VMs are created based on read-only image templates stored on NAS and accessible to all hosts. A VM's virtual disk can use different image formats. QEMU's RAW format is simply a byte-by-byte copy of a physical disk's content stored in a regular file. If a VM uses the RAW format, the VM creation process may take a long time and cause resource contentions, because the host needs to copy a complete image template (i.e., gigabytes of data) from NAS across a heavily shared network in order to create a new RAW image on DAS. This problem is illustrated in Figure 1(a).

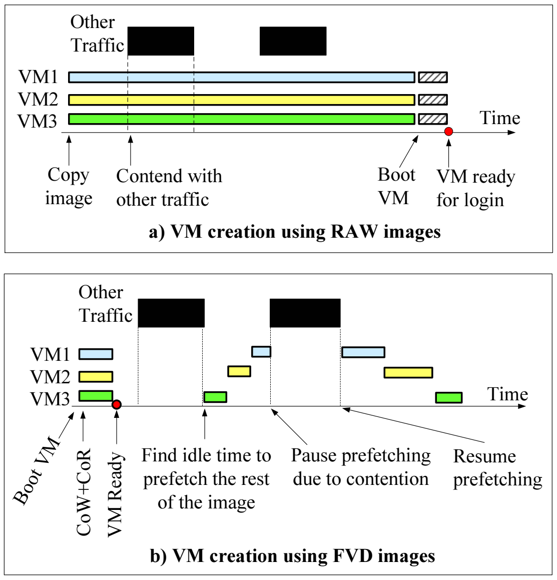

Other Traffic
VM1
VM2
VM3
Time
Copy image
Contend with other traffic
Boot VM
VM ready for login
a) VM creation using RAW images
Other Traffic
VM1
VM2
VM3
Time
Boot VM
CoW+CoR
VM Ready
Find idle time to prefetch the rest of the image
Pause prefetching due to contention
Resume prefetching
b) VM creation using FVD images

Figure 1: Comparison of the VM creation processes. This example creates three VMs concurrently.

QCOW2 [5] is another image format supported by QEMU. It does copy-on-write, i.e., the QCOW2 image only stores data modified by a VM, whereas unmodified data are always read from the base image. QCOW2 supports fast VM creation. The host can instantly create and boot an empty QCOW2 image on DAS, whose base image points to an image template stored on NAS. Using QCOW2, however, limits the scalability of a Cloud, because a large number of VMs may repeatedly read unmodified data from the base image, generating excessive network traffic and I/O load on NAS.

The solution in FVD is to do copy-on-read (CoR) and adaptive prefetching, in addition to copy-on-write (CoW). CoR avoids repeatedly reading a data block from NAS, by saving a copy of the returned data on DAS for later reuse. Adaptive prefetching uses resource idle time to copy from NAS to DAS the image data that have not been accessed by the VM. These features are illustrated in Figure 1(b).

In addition to instant VM creation, FVD also supports instant VM migration, even if the VM's image is stored on DAS. FVD can instantly migrate a VM without first transferring its disk image. As the VM runs uninterruptedly on the target host, FVD uses CoR and adaptive prefetching to gradually move the image from the source host to the target host, without user perceived downtime.

2 Overview of FVD

This paper is focused on the Cloud-inspired features of FVD, i.e., copy-on-read and adaptive prefetching. To set stage for the detailed discussion in Section 3, this section first describes how a virtual disk works today, using KVM [4], QEMU [1], and QCOW2 [5] as examples, and then presents an overview of the holistic FVD solution.

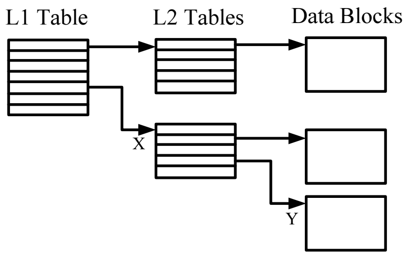


Figure 2: QCOW2's two-level lookup index.

2.1 How a Virtual Disk Works Today

When a VM issues a disk I/O request for data at a *virtual block address* (VBA), the host Linux kernel forwards the request to QEMU running in the user space. QEMU's QCOW2 driver translates the VBA into an *image block address* (IBA), which specifies where the requested data are stored in the QCOW2 image file, i.e., IBA is an offset in the image file. Specifically, QCOW2 uses the index in Figure 2 to perform the address translation. A VBA d is split into three parts, i.e., $d = (d_1, d_2, d_3)$. The d_1 entry of the L1 table points to an L2 table X. The d_2 entry of the L2 table X points to a data block Y. The requested data are located at offset d_3 of the data block Y.

Initially, a QCOW2 image contains only the L1 table, with all data blocks and L2 tables unallocated. A data block is allocated at the end of the image file upon the first write to that block. As a result, a block's IBA solely depends on when it is written for the first time, regardless of its VBA. This behavior may end up with an undesirable data layout on the physical disk. For example, when the guest OS creates a file system, it writes out the file system metadata, which are all grouped together and assigned consecutive IBAs by QCOW2, despite the fact that the metadata's VBAs are deliberately scattered for better reliability and locality, e.g., co-locating inodes and file content blocks in block groups. As a result, it may cause a long disk seek distance between accessing a file's metadata and accessing the file's content blocks. This problem is not unique to QCOW2. It exists in all popular image formats, including VDI, VMDK, and VHD.

When the guest VM reads data at the VBA d=(d_1, d_2, d_3), the QCOW2 driver determines whether the data block is allocated in the QCOW2 image by checking if the corresponding L1 or L2 table entry is empty. If so, the data are read from the base image. Otherwise the data are read from the QCOW2 image. Since the lookup index implements both dirty block tracking and storage allocation, the two functions become inseparable.

2.2 How FVD Works

Below, we summarize the on-disk metadata used by FVD to provide a diverse set of functions:

- A bitmap for implementing copy-on-write.

- A one-level lookup table for implementing storage allocation.
- A metadata journal for committing changes of the bitmap and the lookup table.
- A reference-count table for implementing internal snapshot.

Bitmap. The bitmap is enabled only if a new FVD image is created based on an existing image template (so-called *base image*). When the VM issues a disk write, the base image is not modified. Instead, the new data are saved in the FVD image. This behavior is called copy-on-write. A bit in the bitmap tracks where the latest content of a *block* is stored. The bit is 0 if the block is in the base image, and the bit is 1 if the block is in the FVD image. The unit of a block is configurable per virtual disk, with a default size of 64KB. To represent the state of a 1TB base image, FVD only needs a 2MB bitmap, which can be easily cached in memory. The bitmap also implements copy-on-read and adaptive prefetching.

Lookup table. The lookup table implements storage allocation. One entry of the look table maps a data *chunk*'s VBA to its IBA. The unit of a chunk is configurable per virtual disk, with a default size of 1MB. (Note that VDI uses 1MB chunks. VHD and the ESX version of VMDK use 2MB chunks.) For a 1TB virtual disk, the size of FVD's lookup table is only 4MB. Because of the table's small size, there is no need to use a more complicated two-level index as that in QCOW2.

Because FVD itself is capable of managing storage allocation, one valid configuration is to store an FVD image directly on a logical volume to avoid the overhead of a sophisticated host file system. This configuration still supports storage thin provisioning. The initial size of the logical volume can be small. During the execution of the VM, FVD asks the host OS to increase the size of the logical volume when more storage space is needed.

Separating the implementation of copy-on-write from the implementation of storage allocation provides several benefits. First, the lookup table can be optionally disabled to avoid the overhead and data fragmentation caused by doing storage allocation at the image level. In this case, FVD maintains a linear mapping between a chunk's VBA and IBA without any address translation, and relies on the host file system for storage allocation.

Another benefit is that it makes the metadata smaller and easier to cache, by using the bitmap to track data at the finer *block* granularity, and using the lookup table to track data at the coarser *chunk* granularity. The bitmap is small because of its efficient representation. The lookup table is small because the large chunk size leads to less table entries. For a 1TB virtual disk, FVD's bitmap and one-level lookup table together are only 6MB, whereas QCOW2's two-level lookup table is 128MB.

Metadata journal. When the bitmap and/or the lookup table need be modified, the changes are saved in the journal, as opposed to updating the bitmap and/or the lookup table directly. The journal size is configurable per virtual disk, with a default size of 16MB. When the journal is full, which happens infrequently, the entire bitmap and the entire lookup table are flushed to disk. Then the journal can be recycled for reuse. The flush avoids the overhead of fine-grained journal cleaning operations that are common in journaling file systems. The flush is quick, because the bitmap and the lookup table are small.

The journal provides several benefits. First, updating both the bitmap and the lookup table requires only a single write to the journal. Second, k concurrent updates to any potions of the bitmap or the lookup table are converted to sequential writes in the journal. Finally, it increases concurrency by allowing multiple parallel updates to the same sector in the bitmap or the lookup table.

Reference-count table. There are two ways of implementing virtual disk snapshot: external snapshot and internal snapshot. External snapshot can be easily implemented on top of any image form that already supports copy-on-write (CoW), including VMDK, QCOW2, and FVD. When the user takes a snapshot, the current image file S_{i-1} is made read-only and a new CoW image file S_i is created based on S_{i-1}. After a series of snapshots are taken, it creates a chain of dependent snapshot files $S_0 \leftarrow S_1 \leftarrow \cdots \leftarrow S_i$. Deleting a snapshot S_{j-1} in the middle of a snapshot chain can be a slow operation. Before removing the snapshot file S_{j-1}, it must physically copy from S_{j-1} to S_j those data chunks modified in S_{j-1} but not modified in S_j.

Internal snapshot avoids this problem by storing all snapshots in a single file. For each data chunk C in use, an entry in the reference-count table records the number of snapshots using C. Creating/deleting a snapshot simply amounts to incrementing/decrementing the reference count of data chunks that form the snapshot. A data chunk is free for reuse when its reference count becomes zero.

QCOW2 and FVD are the only two image formats that support internal snapshot, but they differ in implementation and performance. Conceptually, an image consists of an arbitrary number of read-only historical snapshots and a single writable current view (WCV). The WCV is the virtual disk content perceived by the running VM. QCOW2's reference-count table tracks all data chunks used by either snapshots or the WCV. Because the WCV changes as the VM runs, during normal executions of the VM, QCOW2 incurs disk I/O overhead for updating the on-disk reference-count table and memory overhead for caching the reference-count table. By contrast, FVD's reference-count table tracks chunks used by snapshots but does not track chunks used by the WCV (since the WCV is already tracked by the lookup table). Because read-only snapshots do not change during normal executions of the VM, FVD need not update or cache the reference-count table during normal executions of the VM.

3 Using FVD's Bitmap to Support Copy-on-Write, Copy-on-Read and Prefetching

FVD is a comprehensive solution with many features. Due to the space limitation, the rest of this paper is focused on using FVD's bitmap to support copy-on-write, copy-on-read, and adaptive prefetching. The discussion below assumes that the bitmap is enabled but all other metadata (the lookup table, the metadata journal, and the reference-count table) are disabled. This configuration by itself is a functional, high-performance image format. Figure 3 shows FVD under this configuration.

3.1 Basic Read/Write Operation

With the lookup table disabled, FVD maintains a linear mapping between a block's VBA and IBA. When the VM writes to a block with VBA d, FVD stores the block at offset d of the "FVD Data Region" in Figure 3, without any address translation. FVD relies on the host OS for storage allocation. If the FVD image is stored on a host file system that supports sparse files, no storage space is allocated for a data block in the virtual disk until the VM actually writes to that block.

To start a new VM in a Cloud, the host creates an FVD image on its DAS, whose base image points to an image template on NAS. The "FVD Data Region" in Figure 3 can be larger than the base image, because an image template can be used to create VMs whose virtual disks are of different sizes, depending on how much the user pays. *resize2fs* can expand the file system in the base image to the full size of the virtual disk.

When handling a disk write request issued by the VM, the FVD driver stores the data in the FVD image and updates the bitmap to indicate that those data now are in the FVD image rather than in the base image. The bitmap-update step is skipped if the corresponding bit(s) in the bitmap are set previously. If the write request is not aligned on the block boundary, before writing the data to the image, the FVD driver reads a full block from the base image and merges it with the data to be written.

When handling a disk read request issued by the VM, the FVD driver checks the bitmap to determine if the requested data are in the FVD image. If so, the data are read from the FVD image. Otherwise, the data are read from the base image and returned to the VM. While the VM continues to process the returned data, in the background, a copy of the returned data is saved in the FVD image. Future reads for the same data will get them from the FVD image on DAS rather than from the base image on NAS. This copy-on-read behavior helps avoid generating excessive network traffic and I/O load on NAS.

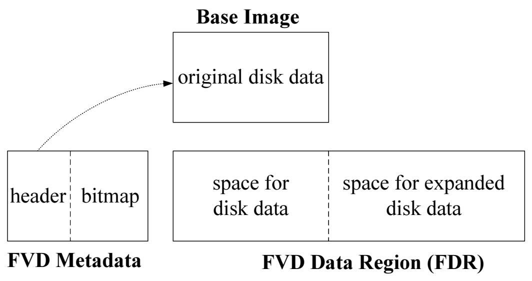


Figure 3: An simplified view of the FVD image format, with only the bitmap enabled.

3.2 Optimizations for Read/Write

Compared with the RAW image format, a copy-on-write image format always incurs additional overhead in reading and updating its on-disk metadata. Below, we summarize several optimizations that eliminate this overhead in common cases. The word "free" below means no need to update the on-disk bitmap.

In-memory bitmap: Eliminate the need to repeatedly read the bitmap from disk by always keeping a complete copy of the bitmap in memory. The bitmap is only 20KB for a 1TB FVD image based on a 10GB image template. Note that, in Figure 3, the bitmap size is proportional to the base image size rather than the FVD image size.

Free writes to beyond-base blocks: Eliminate the need to update the on-disk bitmap when the VM writes to a block residing in the "space for expanded disk data" in Figure 3. This is a common case if the base image is reduced to its minimum size by *resize2fs*. Note that 1) a minimum-sized image template has no unused free space, and 2) most data in an image template are read-only and rarely overwritten by a running VM due to the template nature of those data, e.g., program executable. Consequently, disk writes issued by a running VM mostly target blocks residing in the "space for expanded disk data" in Figure 3. Since those "beyond-base" blocks cannot reside in the base image and hence have no state bits in the bitmap, there is simply no need to update the bitmap when writing to those blocks.

Free writes to zero-filled blocks: Eliminate the need to update the on-disk bitmap when the VM writes to a block whose original content in the base image is completely filled with zeros. This is a common case if the base image is not reduced to its minimum size by *resize2fs* and has many empty spaces. This optimization is realized by using a tool to search for zero-filled blocks in the base image and preset their state bits to 1 in the FVD bitmap. This is an offline process only done once per image template.

Free copy-on-read and free prefetching: Eliminate the need to update the on-disk bitmap when the FVD driver saves a block in the FVD image due to either copy-on-read or prefetching. This does not compromise data integrity in the event of a host crash, because the block's content in the FVD image is identical to that in the base image and reading from either place gets the correct data.

Zero overhead once prefetching finishes: Entirely eliminate the need to read or update the bitmap, once all blocks in the base image are prefetched. This is because the bitmap's content is known in a priori to be 1 for all bits.

3.3 Adaptive Prefetching

FVD uses copy-on-read to bring data blocks from NAS to DAS on demand as they are accessed by the VM. Optionally, prefetching uses resource idle time to copy from NAS to DAS the rest of the image that have not been accessed by the VM. Prefetching is a resource intensive operation, as it may transfer gigabytes of data across a heavily shared network. To avoid causing a contention on any resource (including network, NAS, and DAS), FVD can be configured to limit prefetching rate and pause prefetching when a resource contention is detected.

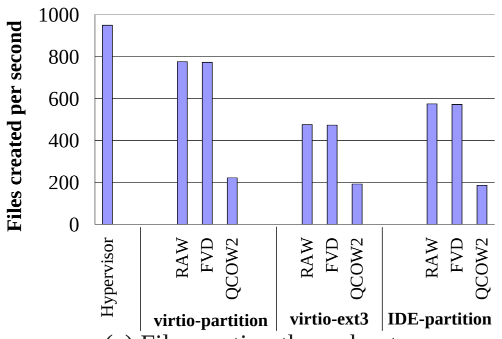
Two throughput limits (KB/s) control the behavior of prefetching data from the base image. The base image read throughput is capped at the upper limit using a leaky bucket algorithm. If the throughput drops below the lower limit, the FVD driver concludes that a resource contention has occurred. It makes a randomized decision. With a 50% probability, it temporarily pauses prefetching for a randomized period of time. If the throughput is still below the lower limit after prefetching resumes, it pauses prefetching again for a longer period of time, and so forth. Similarly, two throughput limits control the behavior of writing prefetched data to the FVD image.

4 Experimental Results

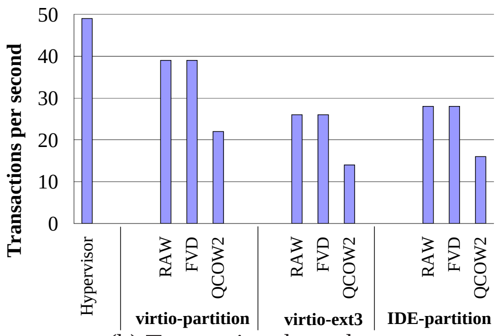
We implemented FVD in QEMU. Due to the space limitation, this paper only presents the results of one experiment. (More results are available in the longer version of this paper [8]). In this experiment, FVD's bitmap is enabled but all other metadata (the lookup table, the metadata journal, and the reference-count table) are disabled. Moreover, since QCOW2 does not support CoR and prefetching, those features are disabled in FVD in order to make a fair comparison of the basic CoW feature.

The experiment is conducted on IBM HS21 blades connected by 1Gb Ethernet. Each blade has two 2.33GHz Intel Xeon 5148 CPUs and a 2.5-inch hard drive (model MAY2073RC). The blades run QEMU 0.12.30 and Linux 2.6.32-24 with the KVM kernel modules. QEMU is configured to use direct I/O.

Figure 4 shows the performance of PostMark [3] under different configurations. The execution of PostMark consists of two phases. In the first "file-creation" phase, it generates an initial pool of files. In the second "transaction" phase, it executes a set of transactions, where each transaction consists of some file operations (creation,



(a) File creation throughput



(b) Transaction throughput

Figure 4: Performance of PostMark.

deletion, read, and append). In this experiment, the total size of files created in the first phase is about 50GB, and the size of an individual file ranges from 10KB to 50KB.

In Figure 4, the "Hypervisor" bar means running PostMark in a native Linux without virtualization. The "RAW", "FVD", and "QCOW2" bars mean running PostMark in a VM whose image uses the different formats, respectively. Like that in a Cloud, a QCOW2 or FVD image V is stored on the local disk of a blade X, whereas the base image of V is stored on another blade Y accessible through NFS. The base image contains Ubuntu 9.04, and is reduced to its minimum size (501MB) by *resize2fs*. A RAW image is always stored on the local disk of a blade. The VM's virtual disk is divided into two partitions. The first partition of 1GB stores the root file system. The second partition of 50GB disk is formatted into an ext3 file system, on which PostMark runs.

For the "IDE-partition" group in the figure, the VM's block device uses the IDE interface and the VM image is stored on a raw partition in the host. For the "virtio-ext3" group, the VM's block device uses the paravirtualized *virtio* interface and the VM image is stored on a host ext3 file system, which is reformatted before each run of the experiment. For the "virtio-partition" group, it uses *virtio* and the VM image is stored on a raw partition in the host.

Figure 4 shows significant advantages of FVD over QCOW2. In the file creation phase, the throughput of FVD is 249% higher than that of QCOW2 (by comparing the "FVD" bar and the "QCOW2" bar in the "virtio-partition" group of Figure 4(a)). In the transaction phase, the throughput of FVD is 77% higher than that of QCOW2 (by comparing the "FVD" bar and the "QCOW2 bar in the "virtio-partition" group of Figure 4(b)).

To understand the root cause of the performance difference, we perform a deep analysis for the results in the "virtio-partition" group of Figure 4(a). We run the *blktrace* tool in the host to monitor disk I/O activities. QCOW2 causes 45% more disk I/Os than FVD does, due to QCOW2's reads and writes to its metadata. Moreover, the average seek distance in QCOW2 is 5.6 times longer than that in FVD, due to QCOW2's VBA-IBA mismatching problem, as explained in Section 2.1.

5 Related Work

Despite the widespread use of VMs, there is no published research on how image formats impact disk I/O performance. Existing popular image formats (including QCOW2 [5], VDI [10], VMDK [11], and VHD [6]) all allocate storage space for a data block at the end of the image file when the block is written for the first time, regardless of the block's virtual address. This mismatch between VBA and IBA invalidates many optimizations in guest file systems, as discussed in Section 2.1. Moreover, they all unnecessarily mix the function of CoW dirty block tracking with the function of storage space allocation. This leads to doing storage allocation twice (first in the image format and then in the host file system), which causes data fragmentation twice and doubles the disk I/O overhead for metadata access.

Existing virtual disks support neither copy-on-read (CoR) nor adaptive prefetching. Some virtualization solutions do support CoR or prefetching, but they are implemented for specific use cases, e.g., virtual appliance [2] and VM migration [7]. By contrast, FVD provides CoR and prefetching as standard features of a virtual disk, which can be easily deployed in many different use cases. Moreover, those previous works use CoW and CoR but do not study how to optimize the CoW and CoR techniques themselves to reduce overhead.

Collective [2] provides desktop as a service across the Internet. It uses CoW and CoR to hide network latency. Its local disk cache makes no effort to preserve a linear mapping between VBA and IBA, and may cause a long disk seek distance as that in popular CoW image formats. Collective also performs adaptive prefetching. It halves the prefetch rate if a certain "percentage" of recent requests experience a high latency. Our evaluation shows that it is hard to set a proper "percentage" to reliably detect congestion. Because storage servers and disk controllers perform read-ahead in large chunks for sequential reads, a large percentage (e.g., 90%) of a VM's prefetching reads hit in the read-ahead caches and experience a low latency. When a storage server becomes busy, the "percentage" of requests that hit in the read-ahead caches may change little, but the response time of those cache-miss requests may increase dramatically. In other words, this "percentage" does not correlate well with the achieved disk I/O throughput.

6 Conclusion

FVD is a holistic virtual disk solution for both Cloud and non-Cloud environments. A design principle of FVD is to make all functions orthogonal so that each function can be enabled or disabled individually. The purpose is to support diverse use cases without being burdened with the overhead of all functions. Using copy-on-write, copy-on-read, and adaptive prefetching, FVD supports instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. The source code of FVD is publicly available at `https://researcher.ibm.com/researcher/view_project.php?id=1852`.

References

[1] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX FREENIX Track*, 2005.

[2] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *NSDI*, 2005.

[3] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[5] M. McLoughlin. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`.

[6] Microsoft VHD Image Format. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

[7] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *OSDI*, 2002.

[8] C. Tang. FVD: a High-Performance Virtual Machine Image Format for Cloud. This is the longer version of the USENIX'11 paper with the same title, available at `https://researcher.ibm.com/researcher/view_project.php?id=1852`.

[9] The IBM Cloud. `http://www.ibm.com/services/us/igs/cloud-development/`.

[10] VirtualBox VDI Image Format. `http://forums.virtualbox.org/viewtopic.php?t=8046`.

[11] VMware Virtual Disk Format 1.1. `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.