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

Chunqiang Tang
IBM T.J. Watson Research Center
ctang@us.ibm.com
http://www.research.ibm.com/people/c/ctang/

***Note**: This paper describes the copy-on-write, copy-on-read, and adaptive prefetching capabilities of FVD. The compact image capability of FVD is described separately in a companion paper entitled "Compact Image Support in Fast Virtual Disk (FVD)"*, which is available at `https://researcher.ibm.com/researcher/view_project.php?id=1852`

Abstract

This paper analyzes the gap between existing hypervisors' virtual disk capabilities and the requirements in a Cloud, and proposes a solution called FVD (Fast Virtual Disk). FVD consists of an image format and a block device driver designed for QEMU. Despite the existence of many popular virtual machine (VM) image formats, FVD came out of our unsatisfied needs in the IBM Cloud. FVD distinguishes itself in both performance and features. It supports instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. These are important use cases in an elastic Cloud, but are not well supported by existing image formats. FVD supports these use cases by adopting a combination of copy-on-write, copy-on-read, and adaptive prefetching. The latter two features are not available from existing image formats and their drivers.

In the design of FVD, performance is even more important than features. With copy-on-read and adaptive prefetching disabled, FVD can function as a pure copy-on-write image format. In this case, the throughput of FVD is 249% higher than that of QEMU QCOW2 when using the PostMark benchmark to create files. This superior performance is due to aggressive optimizations enabled by debunking a common practice in popular copy-on-write image formats (including QCOW2, VirtualBox VDI, VMware VMDK, and Microsoft VHD), which unnecessarily mixes the function of storage space allocation with the function of dirty-block tracking.

The implementation of FVD in QEMU is mature. Its performance is excellent and its features (copy-on-write, copy-on-read, and adaptive prefetching) are valuable in both Cloud and non-Cloud environments. We actively seek for adoption of FVD into the QEMU mainline (see http://sites.google.com/site/tangchq/qemu-fvd).

1 Introduction

Cloud Computing is widely considered as the next big thing in IT evolution. In a Cloud like Amazon EC2 [1], the storage space for virtual machines' virtual disks can be allocated from multiple sources: the host's direct-attached storage (DAS, i.e., local disk), network-attached storage (NAS), or storage area network (SAN). These options offer different performance, reliability, and availability at different prices. DAS is at least several times cheaper than NAS and SAN, but DAS limits the availability and mobility of VMs.

To get the best out of the different technologies, a Cloud usually offers a combination of block-device storage services to VMs. For instance, Amazon Web Services (AWS) [2] offers to a VM both ephemeral storage (i.e., DAS) and persistent storage (i.e., NAS). Amazon EC2 provides each VM with 170GB or more ephemeral storage space at no additional charge. Persistent storage 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 a popular choice to use ephemeral storage for a VM's root file system, especially for data-intensive applications such as Hadoop.

DAS is simple, cheap, and scalable. The aggregate storage space and I/O bandwidth of DAS scales linearly as hosts are added. However, using DAS slows down the process of VM creation and VM migration, and diminishes the benefits of an elastic Cloud. The discussion below uses KVM [13] and QEMU [3] as examples, because we work on the KVM-based IBM Cloud [23].

In a Cloud, VMs are created based on read-only image templates, which are stored on NAS and accessible to all hosts. A VM's virtual disk can use different image formats. The RAW format is simply a byte-by-byte copy of a physical block device's full 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) across a heavily

shared network in order to create a new RAW image on DAS. This problem is illustrated in Figure 1.

QCOW2 [16] is another image format supported by QEMU. It does copy-on-write, i.e., the QCOW2 image only stores data modified by a VM, while unmodified data are always read from the backing image. QCOW2 supports fast VM creation. The host can instantly create and boot an empty QCOW2 image on DAS, whose backing 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 backing image, generating excessive network traffic and I/O load on the shared NAS server.

Our solution to this problem is the FVD (Fast Virtual Disk) image format and the corresponding driver. In addition to copy-on-write (CoW), FVD also does copy-on-read (CoR) and adaptive prefetching. 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 idle time to copy from NAS to DAS the rest of the image that have not been accessed by the VM. These features are illustrated in Figure 1.

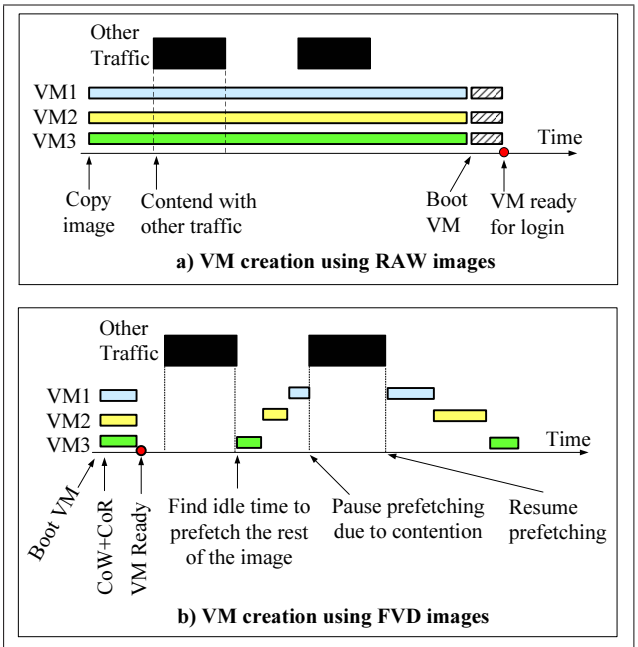


Figure 1: Comparison of VM creation processes. This example creates three VMs concurrently. Using the RAW image format, it has to wait for a long time until an entire image template is copied from NAS to DAS, and then boots the VM. The key observation is that, much of the copied image data is not needed during the VM booting process and may even never be accessed throughout the VM's lifetime. FVD instead boots the VM instantly without any image data on DAS, and copies data from NAS to DAS on demand as they are accessed by the VM. In addition, FVD's prefetching mechanism finds resource idle time to copy from NAS to DAS the rest of the image that have not been accessed by the VM. Prefetching is conservative in that if FVD detects a contention on any resource (including DAS, NAS, or network), FVD pauses prefetching temporarily and resumes prefetching later when congestion disappears.

A main challenge in FVD is to provide the rich features without degrading runtime disk I/O performance. This is a real challenge even for the widely used and supposedly well-understood feature of CoW. We analyzed the popular CoW image formats, including QCOW2, VirtualBox VDI [25], VMWare VMDK [26], and Microsoft VHD [17]. A key finding is that they all unnecessarily mix the function of storage space allocation with the function of dirty-block tracking. As a result, they not only generate more disk I/Os for metadata access but also increase the average disk seek distance due to an undesirable data layout on the physical disk. By contrast, FVD only performs dirty-block tracking and delegates the responsibility of storage space allocation entirely to the underlying layer, which can be a host file system, a host logical volume manager, or simply a raw partition. This simplicity allows FVD to aggressively optimize not only CoW, but also CoR and adaptive prefetching. These optimizations are critical. The throughput of FVD as a pure CoW format is 249% higher than that of QCOW2 when using the PostMark [12] benchmark to create files.

In addition to instant VM creation, FVD also supports instant VM migration, even if the VM's image is stored on DAS. Live migration is an important mechanism for workload management. A public Cloud often leverages the hypervisor's memory over-commit capability to pack a large number of VMs on a host. As the workload changes, the memory working sets of those VMs may increase beyond the host's physical memory capacity and cause thrashing. Ideally, some VMs should be immediately migrated to other hosts to mitigate thrashing. Unfortunately, among all existing hypervisors, KVM/QEMU is the only one that can migrate images stored on DAS, and even QEMU only supports pre-copy storage migration, i.e., the VM's disk data must be copied from the source host to the target host in its entirety before the VM can start to run on the target host. Pre-copy may take a long time due to the large size of the disk image, and VMs may experience long periods of severe thrashing. By contrast, 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 copy the image from the source host to the target host, without user-perceived downtime.

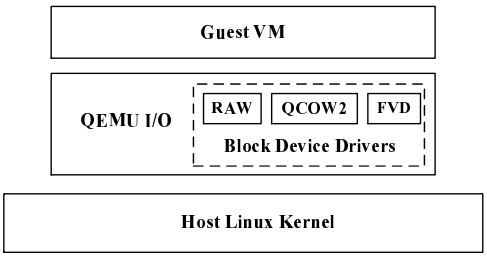


Figure 2: Architecture of KVM and QEMU.

1.1 Contributions

We make the following contributions in this paper.

- We analyze the gap between existing hypervisors' virtual disk capabilities and the requirements in a Cloud, and propose the FVD solution that combines CoW, CoR, and adaptive prefetching.
- We achieve a high-performance implementation of these features through aggressive optimizations enabled by debunking a common practice adopted in popular CoW image formats.
- We bring to the open-source community for the first time an extremely high-performance CoW implementation and the new features of CoR and adaptive prefetching.

The rest of the paper is organized as follows. Section 2 provides background. Sections 3 and 4 present and evaluate FVD, respectively. Related work is discussed in Section 5. Section 6 concludes the paper.

2 Background

2.1 KVM and QEMU

KVM [13] is a Linux kernel virtualization infrastructure. It uses QEMU [3] to perform I/O emulation in the user space. QEMU is also used in other hypervisors, including Xen-HVM and VirtualBox. Figure 2 shows the architecture of KVM and QEMU. When the guest VM issues an I/O instruction, it is trapped by the host Linux kernel, which redirects the request to QEMU running in the user space. QEMU gets services from the host Linux kernel through system calls, just like a normal user-space process. For example, when handling the guest VM's disk read request for a block, QEMU's QCOW2 driver may invoke system calls to read the block from a QCOW2 image file stored on a host ext3 file system. For readers familiar with Xen, conceptually QEMU's block device driver handles a gust VM's I/O requests in a way similar to how the *tapdisk* process works in Xen's Dom0.

2.2 Virtual Machine Image Creation Process

In a Cloud, a VM is created based on a read-only image template, which is stored on NAS and accessible to all hosts. Below is one example of the process to prepare a Linux image template. The image template uses the RAW image format. Suppose the initial image template size is 50GB. It is first installed with the needed software and fully tested. Then the ext3 file system in the image template is resized to its minimum size (e.g., from 50GB down to 12GB) by using the *resize2fs* tool. The image template is truncated to fit the minimum file system size (i.e., from 50GB to 12GB). The resizing and truncating step gets rid of garbage data generated during installation and testing, and produces an image template of a minimum size. A small image template helps reduce the amount of data transferred from NAS to DAS when create new VMs based on the image template.

Following the example above, this 12GB image template can be used to create VMs whose root file systems are of different sizes, depending on how much a user pays for a VM.[1] For example, the following command creates a 30GB QCOW2 image on DAS, based on the 12GB image template stored on NAS: *qemu-img create -f qcow2 -b /nfs/template.raw vm.qcow2 30GB*.

After using *qemu-nbd* to mount the 30GB QCOW2 image and using *fdisk* to expand the disk partition from 12GB to 30GB, *resize2fs* can be used to expand the image's ext3 file system from 12GB to 30GB, which will be the VM's root file system. Note that using *resize2fs* to expand (as opposed to shrink) a file system is a quick operation because it need not relocate blocks.

2.3 Limitations of Existing Copy-on-Write Image Formats

Because of storage virtualization, a block address is translated multiple times before it reaches the physical disk. When the guest VM issues a disk I/O request to the hypervisor using a *virtual block address* (VBA), QEMU's block device driver translates the VBA into an *image block address* (IBA), which specifies where the requested data are stored in the image file, i.e., IBA is an offset in the image file. How the translation is performed is specific to an image format. If the image is stored as a regular file in a host file system, the host file system further translates the IBA to a *physical block address* (PBA) and the I/O request is issued to the physical disk using the PBA. If the VM image is stored directly on a raw partition, IBA and PBA are identical.

QCOW2 uses the lookup index in Figure 3 to translate a VBA into an IBA. 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 in the data block Y.

[1] This capability is not available in Amazon EC2 but is available in the IBM Cloud.

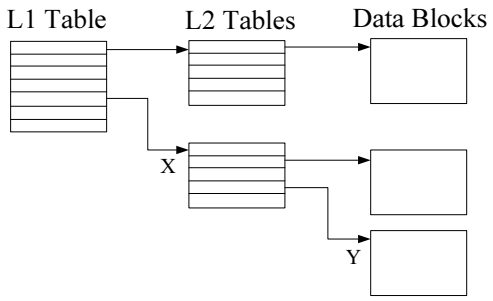


Figure 3: The two-level lookup index in QCOW2.

After a QCOW2 image is created, it initially only contains the L1 table with all entries empty, indicating that the L2 tables and the data blocks are not yet allocated. The size of the QCOW2 image file is only the size of the L1 table plus some header fields. When the guest VM writes data at the VBA d=(d_1, d_2, d_3), the QCOW2 driver checks if the d_1 entry of the L1 table is empty. If so, it allocates an L2 table at the end of the image file (which accordingly grows the size of the image file) and initializes the d_1 entry of the L1 table with the IBA of the newly allocated L2 table. Similarly, upon the first write to a data block, the data block is allocated at the end of the image file, and the corresponding entry in an L2 table is initialized with the IBA of the data block. When the guest VM reads data at the VBA d=(d_1, d_2, d_3), the QCOW2 driver checks whether the data block is allocated in the QCOW2 image. If so, the data is read from the QCOW2 image; otherwise, the data is read from the backing image. In other words, data not modified by the VM are always read from the backing image.

In QCOW2, a block's IBA solely depends on when it is written for the first time, regardless of its VBA. This mismatch between VBAs and IBAs may end up with an undesirable data layout on the physical disk and degrade performance. For example, when a guest OS creates or resizes 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.

To improve performance, it is a common practice to store a VM image directly on a raw partition, bypassing the overhead of a host file system. In this case, PBA equals to IBA. If the VM uses the RAW image format, IBA further equals to VBA. As a result, the block address perceived by the guest OS matches with the actual data layout on the physical disk, which makes many optimizations in the guest file system effective. Experiments in Section 4 show that storing a RAW image on a raw partition may improve performance by 63%, compared with storing the RAW image on a host ext3 file system. By contrast, storing a QCOW2 image on a raw partition may only improve performance by 15%, partially due to the mismatch between VBAs and IBAs.

This problem is not unique to QCOW2. It exists in all other popular CoW image formats, including VirtualBox VDI, VMWare VMDK, and Microsoft VHD. Although these formats use different storage space allocation units, they all use an index structure to translate between VBAs and IBAs, and allocate storage space for a data unit at the end of the image file when the first write to that data unit occurs. As a result, a data unit's IBA solely depends on when it is written for the first time, regardless of its VBA.

In addition to causing mismatch between VBAs and IBAs, another problem with the lookup index in Figure 3 is the performance overhead in reading and updating this on-disk metadata. The in-memory cache maintained by QCOW2 helps metadata reads but not metadata writes. Moreover, random disk I/Os issued by the guest VM may cause frequent cache misses. Experiments in Section 4 show FVD can achieve 126% higher disk I/O throughput than QCOW2 does by eliminating this overhead.

3 The FVD Image Format

FVD supports instant VM creation and instant VM migration, by adopting a combination of copy-on-write (CoW), copy-on-read (CoR), and adaptive prefetching. These features are illustrated in Figure 1. Figure 4 shows a simplified view of the FVD image format.

3.1 Key Difference as a CoW Format

Even without the new features of CoR and prefetching, FVD differs from existing popular CoW image formats (including QCOW2, VDI, VMDK, and VHD) in a simple but fundamental way. All those formats use a lookup index to track storage space allocation, which as a side effect also tracks dirty blocks written by a VM. These two functions are unnecessarily mingled together. We argue that a CoW image format should only perform dirty-block tracking, while delaying and delegating the decision of storage space allocation to the host OS.

This approach offers several advantages. First, the host OS has abundant storage options to optimize for a specific workload, e.g., storing a VM image on a raw partition, on a logical volume, or as a regular file in a host file system, with the choices of ext2/ext3/ext4, JFS, XFS, ReiserFS, etc. An image format is merely a middle layer. Prematurely deciding storage space allocation in an image format destroys opportunities for end-to-end optimizations, and causes problems such as mismatch between VBAs and IBAs. Second, separating dirty-block tracking from storage allocation avoids the overhead associated with reading or updating the on-disk

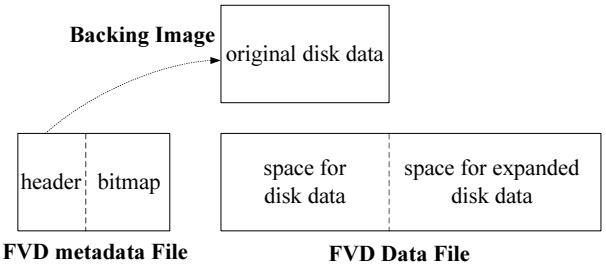


Figure 4: An abstract view of the FVD image format.

lookup index needed for performing storage allocation. The lookup index in Figure 3 is almost identical to the one used in a host file system. Doing storage allocation twice (first time in a CoW image and second time in a host file system) is simply redundant and unnecessary.

The main function of a CoW image format is to keep track of dirty blocks written by the VM. This function can be more easily and more efficiently fulfilled by a simple bitmap without using a lookup index. The difference may seem trivial, but the consequence is profound. Without being burdened with the function of storage allocation, a simple bitmap enables many aggressive optimizations that would otherwise be impossible. An image format in QEMU (which is simply called "COW") also uses a bitmap to track dirty blocks, but it is a naive format that by design does not guarantee data integrity in the event of a host crash.

Abandoning the lookup index in Figure 3 loses few features of QCOW2. In addition to copy-on-write, QCOW2 features sparse image, encryption, snapshot, and compression. Without the lookup index, sparse image can be supported by a host file system's sparse file capability, since almost every modern file system supports sparse files, including GFS, NTFS, FFS, LFS, ext2/ext3/ext4, reiserFS, Reiser4, XFS, JFS, VMFS, and ZFS. (The only notable exception is FAT12/FAT16/FAT32, which is unlikely to be used as a hypervisor's file system anyway.) Encryption does not depend on the lookup index. Snapshot can be implemented without the lookup table, by following VMware VMDK's approach of starting a new CoW image for each snapshot, as opposed to QCOW2's approach of storing all snapshots in one image. Compression is the only feature that cannot be supported by a simple bitmap. Given the high overhead of doing runtime de-compression and its limited use in production, we decide to forgo compression.

3.2 The FVD Format and Basic Operations

Figure 4 shows a simplified view of the FVD image format. An FVD image consists of a metadata file and a data file.[2] Similar to QCOW2, an FVD image is based on a read-only backing image. The FVD header stores a reference to the backing image. The header is followed by a bitmap, with one bit for each data block in the virtual disk. If a bit is set, the corresponding block's current content is stored in the FVD image. Otherwise, the block's current content is stored in the backing image.

FVD maintains a linear mapping between a block's VBA and IBA. When the VM writes to a block with VBA d, the FVD driver stores the block at offset d of the FVD data file, without any address translation. FVD totally relies on the host OS for storage allocation. If the FVD data file 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 data are written into that block.

To start a new VM, the host creates an FVD metadata file on its DAS, whose backing image points to an image template on NAS. Initially, the FVD data file is completely empty. In Figure 4, the FVD data file is larger than the backing image, which reflects the fact that a single backing image can be used to create VMs whose virtual disks are of different sizes (see Section 2.2). *resize2fs* can expand the file system in the backing image to the full size of the virtual disk. During this process, new or modified file system metadata are stored in the FVD data file because of the copy-on-write behavior. The VM is then booted with the expanded root file system. At this point, the FVD data file is still almost empty.

The FVD driver in QEMU handles disk I/O requests issued by a VM. Below, we first describe a naive implementation of the FVD driver to illustrate the basic operations, and then present optimizations that help significantly improve performance.

When handling a disk write request issued by a VM, the FVD driver executes the following steps sequentially to guarantee data integrity in the even of a host crash: 1) store data in the FVD data file and wait until the data are persisted on the physical disk, 2) update the bitmap and wait until the bitmap is persisted on the physical disk, and 3) acknowledge to the VM the completion of the write. Step 2 is skipped if the corresponding bits in the bitmap are set previously. A bit in the bitmap represents the state of a full block. If the I/O request is not aligned on the block boundary, in Step 1, the driver reads a full block from the backing image, merges it with the data being written, and writes the full block to the FVD data file.

When handling a disk read request from the VM, the FVD driver checks the bitmap to determine if the requested data are in the FVD data file. If so, the data are read from the FVD data file. Otherwise, the data are read from the backing 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 data file and the bitmap is updated accordingly. Fu-

[2] FVD also supports storing metadata and data in a single file.

ture reads for the same data will get them from the FVD data file on DAS rather than from the backing image on NAS. This copy-on-read behavior helps avoid generating excessive network traffic and I/O load on NAS.

3.3 Optimizations

Compared with the RAW image format, a copy-on-write image format always incurs additional overheads in reading and updating its on-disk metadata. In FVD, a sequence of sequential write requests from the VM may generate the following write sequence on the physical disk: write d_1, write $bit(d_1)$, write d_2, write $bit(d_2)$, write d_3, write $bit(d_3)$, $\cdots$, and so forth. Here d_1, d_2, and d_3 are blocks with consecutive VBAs, and $bit(d_i)$ is the state bit in the bitmap for block d_i. In this example, the disk head moves back and forth between the FVD metadata file and the FVD data file, which is obviously inefficient.

Below, we first summarize and then explain in detail several optimizations that eliminate disk I/Os for reading or updating the on-disk bitmap 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.

- **Free writes to no-backing 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 4. This is a common case if the backing image is reduced to its minimum size by *resize2fs*.

- **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 backing image is completely filled with zeros. This is a common case if the backing image is not reduced to its minimum size and has many empty spaces.

- **Free copy-on-read**: eliminate the need to update the on-disk bitmap when the FVD driver saves a block in the FVD data file due to copy-on-read.

- **Free prefetching**: eliminate the need to update the on-disk bitmap when the FVD driver saves a prefetched block in the FVD data file.

- **Zero overhead once prefetching finishes**: entirely eliminate the need to read or update the bitmap, once all blocks in the backing image are prefetched.

3.3.1 In-memory Bitmap

Because of its small size, it is trivial to keep the entire bitmap in main memory. In Figure 4, the size of the bitmap is proportional to the size of the (smaller) backing image rather than the size of the (larger) FVD data file. No state bits are needed for blocks residing in the "space for expanded disk data", because those "no-backing" blocks simply cannot be in the backing image. The FVD driver always reads and writes no-backing blocks directly without checking the bitmap.

QCOW2's unit of storage space allocation is 64KB. If one bit in FVD's bitmap represents the state of a 64KB block, the size of the bitmap is only 20KB for a 1TB FVD image based on a 10GB backing image. As a reference point, 10GB is the maximum backing image size allowed for an Amazon EC2 VM running on DAS (although the IBM Cloud allows backing images larger than 10GB). Even if the backing image is unreasonably as large as 1TB, the size of the bitmap is still only 2MB.

3.3.2 Free Writes to No-backing Blocks

As described in Section 2.2, it is a best practice to reduce an image template to its minimum size. 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. As a result, disk writes issued by a running VM mostly target blocks residing in the "space for expanded disk data" in Figure 4. Since those "no-backing" blocks have no state bits in the bitmap, there is simply no need to update the bitmap when writing to those blocks.

3.3.3 Free Writes to Zero-Filled Blocks

It is a best practice but not mandatory to reduce an image template to its minimum size using *resize2fs*. If an image template is not reduced to its minimum size, it can be a sparse file with many "empty" data blocks never written before. Reading an empty block returns an array of zeros. Below, we describe a VM creation process that help eliminate the need to update the on-disk bitmap when the VM writes to a block whose original content in the backing image is completely filled with zeros. These zero-filled blocks can be either empty blocks in a sparse file or non-empty blocks whose contents happen to be zeros.

For an image template *image.raw* stored in the RAW format, we use the *qemu-img* tool to create an FVD metadata file *image.fvd* with *image.raw* as its backing image. Let S_{in_fvd}=1 and $S_{in\_backing}$=0 denote the two states of a bit in the bitmap. When creating *image.fvd*, *qemu-img* searches for zero-filled blocks and set their states in the bitmap to S_{in_fvd}. The states for non-zero blocks are set to $S_{in\_backing}$. The creation of *image.fvd*

is an offline process and is only done once for an image template. *image.fvd* is stored on NAS together with *image.raw*. When creating a new VM on a host, it copies *image.fvd* from NAS to DAS, and optionally expand the size of the virtual disk by changing the *disk-size* field in *image.fvd*. Copying *image.fvd* is fast because it is a small FVD metadata file consisting of mostly the bitmap.

When the VM boots, the FVD driver automatically creates an empty, sparse FVD data file according to the virtual disk size specified in *image.fvd*. Suppose the VM issues a read request for a block whose original content in the backing image is filled with zeros, and the VM did not write to the block before. Because the block's state is initialized to S_{in_fvd} when creating *image.fvd*, the FVD driver reads the block from the FVD data file. Because the VM did not write to the block before, the block is an empty block in the FVD data file, and the read returns an array of zeros. This outcome is correct and is identical to reading from the backing image. This optimization eliminates the need to read the block from the backing image stored on NAS. When the VM writes to this block, the FVD driver stores the data in the FVD data file without updating the on-disk bitmap, because the block's initial state in *image.fvd* is already S_{in_fvd}.

3.3.4 Free Copy-on-Read and Free Prefetching

When the FVD driver copies a block from the backing image into the FVD data file due to either copy-on-read or prefetching, it does not immediately update the block's state in the on-disk bitmap from $S_{in\_backing}$ to S_{in_fvd}, which reduces disk I/O overhead. This does not compromise data integrity in the event of a host crash, because the block's content in the FVD data file is identical to that in the backing image and reading from either place gets the correct data.

This optimization needs to handle the subtle case that a block brought in through copy-on-read or prefetching is later overwritten by the VM. For this purpose, the FVD driver maintains three copies of the bitmap, called *on-disk bitmap*, in-memory *accurate-state*, and in-memory *stale-state*, respectively. When a VM boots, the on-disk bitmap is loaded into memory to initialize both the accurate-state and the stale-state. At runtime, the FVD driver always keeps the accurate-state up-to-date, but lazily updates the on-disk bitmap in order to reduce disk I/O overhead. The stale-state is an in-memory mirror of the on-disk bitmap for efficient access.

When handling the VM's read request for a block whose accurate-state is S_{in_fvd}, the FVD drivers reads the block from the FVD data file. When handling the VM's read request for a block whose accurate-state is $S_{in\_backing}$, the FVD driver reads the block from the backing image and returns it to the VM. In the background, the FVD driver writes the block into the FVD data file and updates the block's accurate-state from $S_{in\_backing}$ to S_{in_fvd}. However, the block's on-disk bitmap and stale-state are not updated and remain $S_{in\_backing}$, which reduces disk I/O overhead. The accurate-state is flushed to update the on-disk bitmap lazily, either periodically (e.g., once every hour) or only when the VM is shut down or suspended.

When the VM issues a write request for a block, the FVD drivers checks the stale-state (as opposed to the accurate-state) to determine the appropriate action. If the block's stale-state is already S_{in_fvd}, the FVD driver simply writes the block to the FVD data file. If the block's stale-state is $S_{in\_backing}$, the FVD driver writes the block to the FVD data file, updates the block's on-disk bitmap state, stale-state, and accurate-state all to S_{in_fvd}, and finally acknowledges to the VM the completion of the write operation.

3.3.5 Zero Overhead once Prefetching Finishes

A block's state may be initialized to S_{in_fvd} if its original content in the backing image is completely filled with zeros. As the VM runs, a block's state may also be changed from $S_{in\_backing}$ to S_{in_fvd} due to a write issued by the VM, a copy-on-read operation, or a prefetching operation. Once prefetching finishes, every block's state is S_{in_fvd}. The FVD metadata file can be discarded and the FVD data file can be used as a pure RAW image. Even it is still opened as an FVD image, a field in the FVD metadata file indicates that prefetching has finished and the FVD driver simply passes through all disk I/O requests issued by the VM to the RAW driver without adding any overhead in reading or updating the bitmap.

3.3.6 Discussion of Alternative Optimizations

FVD stores the bitmap and the data blocks separately. One optimization is to partition data blocks into block groups, like that in file systems. Each block group has its own bitmap and data blocks. This optimization reduces disk seek time between writing a block and updating its state bit. However, we decide not adopt this optimization because 1) with the other optimizations, it is a rare operation to update the on-disk bitmap, and 2) this optimization makes the layout of the FVD data file different from that of a RAW image. Once prefetching finishes, an FVD data file is identical to a RAW image, which has the best performance and can be easily manipulated by many existing tools.

When the FVD driver performs a copy-on-read operation, one potential optimization is to delay the action of saving the data block into the FVD data file. This offers two benefits. First, it may avoid interference with other disk reads or writes that are on the critical path of the VM's execution. Second, after a short delay, the save operation may no longer be needed, if the VM's oper-

ation on the block follows a read-modify-write pattern, i.e., the VM reads the block, modifies it, and then writes it back. Since the block is modified, it is unnecessary and actually incorrect to save the old content to the disk.

All the optimizations discussed so far focus on improving runtime performance. Other optimizations may help offline manipulation of an FVD image, e.g., image backup and format conversion. A main challenge is to efficiently identify parts of an FVD image that are sparse. Due to the lack of support in Linux system calls, a user-level program cannot tell whether a block is empty until it reads the block and checks if the block is filled with zeros. Our measurement shows that a singled-threaded program can perform the reading and checking operation on a sparse file at the throughput of about 822MB/s. At this rate, it takes 22 minutes to scan through a 1TB completely empty file.

To facilitate offline image manipulation, FVD can be configured to track the sparseness of "no-backing" blocks residing in the "space for expanded disk data" in Figure 4, by adding state bits for those blocks. When handling the VM's write request for a no-backing block, the FVD driver writes the block to the FVD data file, updates the in-memory bitmap, but does not update the on-disk bitmap immediately. The in-memory bitmap is flushed to update the on-disk bitmap lazily, either periodically or when the VM is shut down, which reduces disk I/O overhead. The FVD metadata file has a field that indicates whether the VM went through a clean shutdown last time and hence the on-disk bitmap is up-to-date. In the rare event of a host crash, an offline tool can scan through the image and fix incorrect states in the bitmap. The does not compromise data integrity or a running VM's correctness.

To facilitate offline image manipulation, FVD can also be configured to more precisely track the state of a block residing in the "space for disk data" in Figure 4. Instead of using one bit, it can use two bits to represent four states: $S_{in\_backing\_zero}$, $S_{in\_backing}$, $S_{in\_fvd\_clean}$, and $S_{in\_fvd\_dirty}$. $S_{in\_backing\_zero}$ means the block's current content is in the backing image, and that content is filled with zeros. $S_{in\_backing}$ means the block's current content is in the backing image, and that content is not filled with zeros. $S_{in\_fvd\_clean}$ means the block's content is in the FVD data file, and this content is identical to that in the backing image. $S_{in\_fvd\_dirty}$ means the block's current content is in the FVD data file, and this content differs from that in the backing image. The FVD driver updates the on-disk bitmap lazily. Only a state change to $S_{in\_fvd\_dirty}$ need be written to the disk immediately.

3.4 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 idle time to copy not-yet-touched blocks from NAS to DAS. Below, we describe the details of FVD's adaptive prefetching algorithm.

3.4.1 What to Prefetch

There are multiple ways of choosing the data to prefetch: locality-based prefetching, profile-directed prefetching, and whole-image prefetching. With locality-based prefetching, when the VM reads a data block d that is currently stored on NAS, the driver copies from NAS to DAS not only block d but also other blocks whose VBAs are close to the VBA of d. FVD does not do this type of prefetching, because it is already performed by other components, e.g., guest OS, NFS server, and disk controller.

FVD supports profile-directed prfetching. It uses offline profiling to identify data blocks that are read during typical uses of a VM image template, e.g., booting the VM, starting a Web server, and serving some Web requests. The VBAs of those blocks are sorted based on priority and locality (e.g., blocks needed earlier have a higher priority and blocks with close-by VBAs are grouped together) and stored in the header of the FVD metadata file. At runtime, the FVD driver prefetches those blocks from NAS to DAS accordingly.

After profile-directed prfetching finishes, the FVD driver may optionally perform whole-image prefetching. It finds idle time to sequentially copy the entire image from NAS to DAS. A data block is skipped during prefetching if it is already stored on DAS. Once whole-image prefetching finishes, a flag is set in the FVD metadata file, and all subsequent reads or writes to the image incur no overhead in checking or updating the bitmap, as described in Section 3.3.5.

3.4.2 When to Prefetch

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 1) delay prefetching, 2) limit prefetching rate, and/or 3) automatically pause prefetching when a resource contention is detected.

A policy controls when prefetching starts. For instance, for the use case of instant VM creation, prefetching may start after a VM runs for 12 hours so that prefetching is not performed for short-lived VMs. For the use case of VM migration, prefetching may start immediately after the VM runs at the new location.

Once prefetching starts, its operation follows a producer-consumer pattern. A producer reads data from

the backing image, and puts the data in a constant-size in-memory buffer pool. A consumer writes data in the buffer pool to the FVD data file. At any moment in time, the producer has at most one outstanding read to the backing image, and the consumer has at most one outstanding write to the FVD data file. If the buffer pool is full, the producer stalls. If the buffer pool is empty, the consumer stalls.

Two throughput limits (KB/s) for the producer are specified in the FVD metadata file, i.e., the lower limit and the upper limit. The producer periodically measures and adapts the throughput of reading the backing image. The throughput is capped at the upper limit using a leaky bucket algorithm. If the throughput drops below the lower limit, the producer concludes that somebody else is using the network or NAS, and a resource contention has occurred. The producer then 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, with a 50% probability it pauses prefetching again, and so forth. If multiple VMs run on different hosts and their FVD drivers attempt to do prefetching from the same NAS server at the same time, they will detect the contention and 50% of them will pause prefetching in each round, until either all of them pause prefetching or the bottleneck resource is relieved of congestion, whichever comes first. If an FVD driver contends with a non-FVD component (e.g., another VM accessing persistent storage stored on NAS), on average, the FVD driver pauses prefetching after two rounds of making a randomized decision (note that $1+0.5+0.25+0.125+\cdots=2$).

Similarly, the consumer is controlled by two throughput limits. The consumer's throughput drops below the lower limit if there is a contention on DAS. To filter out noises, the FVD driver measures prefetching throughput as a moving average.

Unlike PARDA [9] and TCP congestion control, FVD's prefetching is more conservative in using resources. In the face of a contention, it pauses prefetching rather than trying to get a fair share of the bottleneck resource, because prefetching is not an urgent operation.

4 Experimental Results

We implemented FVD in QEMU. Currently, the alternative optimizations described in Section 3.3.6 are not supported. The features of FVD include copy-on-write (CoW), copy-on-read (CoR), and adaptive prefetching. We evaluate these features both separately and in a combination.

The experiments are 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. The benchmarks include PostMark [12], Iperf [11], IBM WebSphere Application Server (http://www.ibm.com/software/websphere), Linux booting, Linux kernel compilation, and a micro benchmark similar to iozone [10]. A QCOW2 or FVD image V is stored on the local disk of a blade X, whereas the backing image of V is stored on another blade Y accessible through NFS. A RAW image is always stored on the local disk of a blade.

4.1 Copy-on-Write

To evaluate the CoW feature, we compare QCOW2 with a version of FVD that disables copy-on-read and prefetching. QCOW2 is a good baseline for comparison, because both QCOW2 and FVD are implemented in QEMU, and QCOW2 is the well-maintained "native" image format of QEMU. Out of the 15 image formats supported by QEMU, only QCOW2 and RAW are "primary features" (see http://wiki.qemu.org/Features). QCOW2 has also been ported to Xen, since Xen does not have its own CoW image format,

4.1.1 Microbenchmark Results

Figure 5 presents the results of running the *RandIO* micro benchmark we developed. RandIO is similar to the random I/O mode of iozone [10] but with a key difference. iozone does not differentiate the first write and the second write to a block, while RandIO does. This difference is important in evaluating a CoW image format, because the first write incurs metadata update overhead and hence is slower than the second write.

In Figure 5, "H: Hypervisor" means running RandIO directly in a native Linux without virtualization, and RandIO performs random I/Os on a 50GB raw disk partition not formatted with any file system. "R: RAW" means running RandIO in a VM whose virtual disk uses the RAW image format. The host stores the RAW image on a raw partition. Inside the VM, the virtual disk is divided into two partitions. The first partition of 1GB stores the root ext3 file system, while the second partition of 50GB is unformatted. RandIO runs in the VM and performs random I/Os on the second raw partition. Using raw partitions both in the host and in the VM, it avoids the overhead of a file system and precisely measures the performance of a QEMU block device driver.

In Figure 5, the configuration of "F: FVD" is similar to that of "R: RAW" except that the image format is FVD. The backing image of the FVD image uses the RAW format, contains a basic installation of Ubuntu server 9.04, and is reduced to its minimum size (501MB) by *resize2fs*. The backing image's root file system is expanded to occupy the first 1GB partition of the FVD image. The

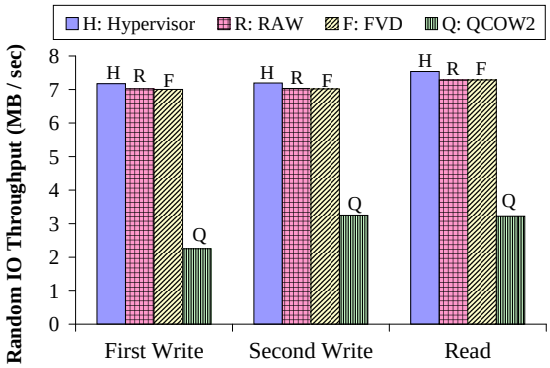


Figure 5: Comparing different image formats by performing random I/Os on a 50GB raw disk partition.

image creation procedure is trivial. It is automated by a shell script with only five commands: 1) "qemu-img create" to create the FVD image; 2) "qemu-nbd -c" to mount the image; 3) "fdisk" to change the partition table; 4) "resize2fs" to expand the root file system; and 5) "qemu-nbd -d" to unmount the image. The execution of the script takes only 0.4 seconds. The configuration of "Q: QCOW2" is similar to that of "F: FVD" except that it uses the QCOW format.

Figure 5 shows that FVD significantly outperforms QCOW2. For the different I/O operations (first write, second write, and read), the throughput of FVD is 211%, 116%, and 126% higher than that of QCOW2, respectively. The overhead in QCOW2 is mainly due to two factors: 1) on the first write, QCOW2 needs to allocate storage space and update is on-disk metadata; and 2) on read or the second write, QCOW2 needs to load parts of its on-disk metadata into memory in order to locate the read or write target. QCOW2's in-memory metadata cache is not large enough to hold all metadata and the cache hit rate is low for random I/Os. By contrast, FVD incurs almost no overhead in reading or updating its on-disk metadata, due to the optimization of "free write to no-backing blocks" described in Section 3.3.2. The throughput of "F: FVD" and "R: RAW" is only about 2.4% lower than that of "H: Hypervisor", indicating that the overhead of storage virtualization is low. This optimistic result is due to the experiment setup—the I/O size is relatively large (64KB), it uses paravirtualization (*virtio*), and it incurs no overhead of a host file system. In other configurations, the overhead can be much higher.

4.1.2 PostMark Results

Figure 6 shows the performance of PostMark [12] under different configurations. PostMark is a popular file system benchmark created by NetApp. 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, 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. The setup of this experiment is similar to that in Figure 5, but the second 50GB partition in the virtual disk is formatted into an ext3 file system, on which PostMark runs.

In Figure 6, 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. 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. For the "IDE-partition" group, the VM's block device uses the IDE interface and the VM image is stored on a raw partition in the host.

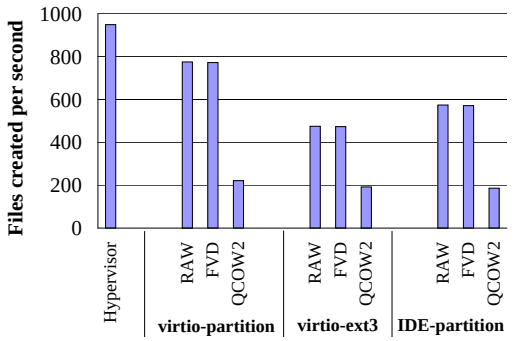
The paravirtualized *virtio* interface shows significant performance advantages over the IDE interface. In Figure 6(a), the throughput of virtio is 35% higher than that of IDE (by comparing the "RAW" bar in the "virtio-partition" group with the "RAW" bar in the "IDE-partition" group). However, even with virtio, storage virtualization still incurs significant overhead. The "Hypervisor" bar is 22% higher than the "RAW" bar in the "virtio-partition" group.

Storing the VM image on a raw partition provides much better performance than storing the image on a host ext3 file system. In Figure 6(a), the throughput of raw partition is 63% higher than that of ext3 (by comparing the "RAW" bar in the "virtio-partition" group with the "RAW" bar in the "virtio-ext3" group).

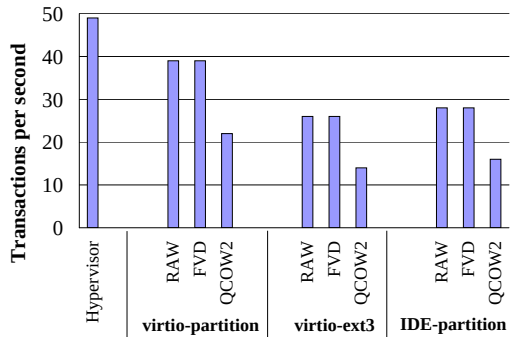
Figure 6 again shows the significant advantages of FVD over QCOW2. In the file creation phase, the throughput of FVD is 249% higher than that of QCOW2 (by comparing the "FVD" bar and the "QCOW2 bar in the "virtio-partition" group of Figure 6(a)). In the transaction phase, the throughput of FVD is 77% higher than that of QCOW2 (by comparing the "FVD" bar and the "QCOW2 bar in the "virtio-partition" group of Figure 6(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 6(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. However, this still does not fully explain the 249% difference in throughput between FVD and QCOW2. The other factor is increased disk seek distance, as explained below.

Figure 7(a) and (b) shows the histogram of of disk I/Os issued by QCOW2 and FVD, respectively. A point on the x-axis is a normalized location of the raw partition that

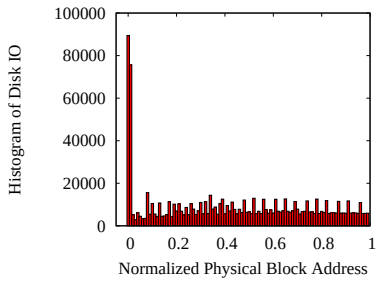


(a) File creation throughput

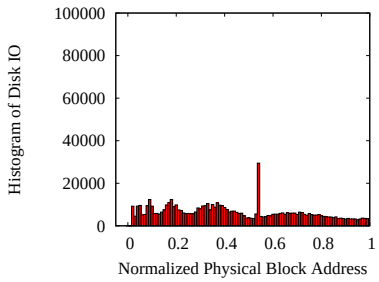


(b) Transaction throughput

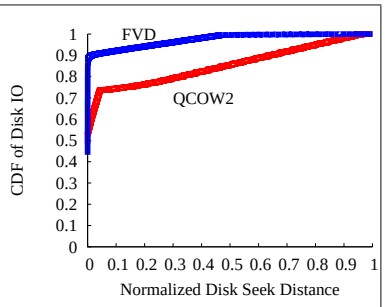
Figure 6: Performance of PostMark under different image formats.



(a) Histogram of disk I/Os issued by FVD.



(b) Histogram of disk I/Os issued by QCOW2.



(c) Comparing CDF of disk seek distances in FVD and QCOW2.

Figure 7: QCOW2 increases disk seek distance because it stores all metadata of the guest file system at the beginning of the image, despite the fact that the metadata's VBAs are deliberately scattered for better reliability and locality.

stores the VM image. Here 0 means the beginning of the partition, 0.5 means the middle of the partition, and 1 means the end of the partition. The y-axis is the number of I/O requests that fall on a location of the raw partition.

Figure 7(b) has a spike in the middle of the raw partition, which is due to frequent accesses to the guest file system's journal file. Figure 7(a) has a different shape. QCOW2 causes two large spikes at the beginning of the raw partition, which are due to frequent accesses to the guest file system's metadata and journal file. As described in Section 2.3, QCOW2 puts all those metadata at the beginning of the raw partition, despite of their scattered VBAs. As a result, in QCOW2, the disk head travels through a long distance between accessing the metadata of a file in the guest file system and accessing the file's content blocks. This effect is clearly shown in Figure 7(c). The x-axis is the normalized disk seek distance between two back-to-back I/O operations, where 1 means the disk head moves all the way from one end of the raw partition to the other end. The average of the normalized seek distance is 0.146 for QCOW2, whereas it is only 0.026 for FVD, i.e., 5.6 times lower. The long disk seek distance is the reason why QCOW2 issues only 45% more disk I/Os than FVD does, but the difference in file creation throughput is as high as 249%.

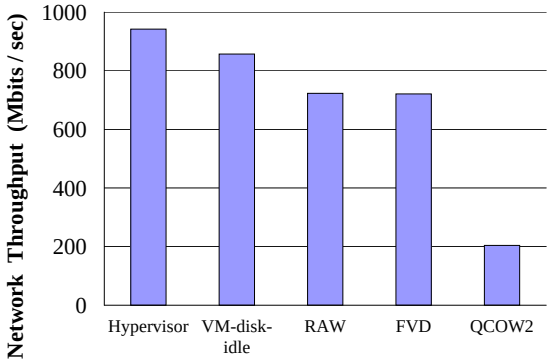


Figure 8: QCOW2's disk activities can severely impact network performance.

4.1.3 Disk I/O's Impact on Network I/O

Unlike the FVD driver's fully asynchronous implementation, the QCOW2 driver synchronously reads and

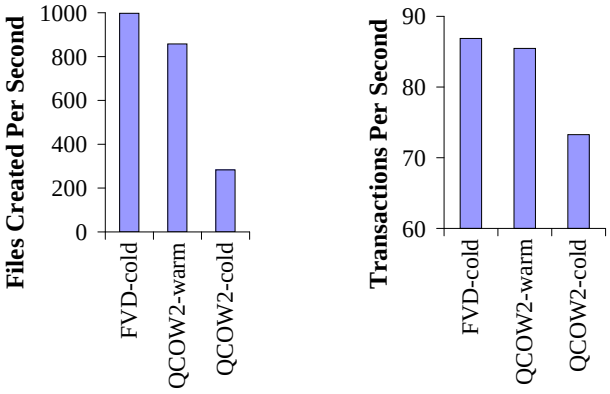


Figure 9: FVD still outperforms QCOW2 even if the experiment is designed to favor QCOW2.

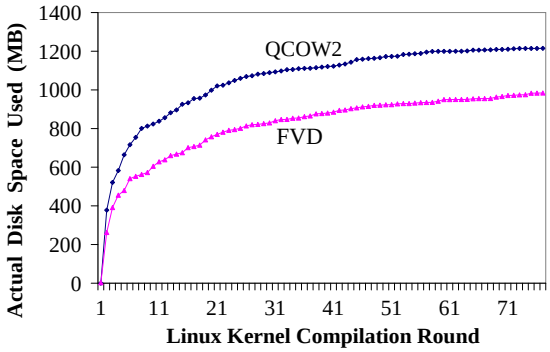


Figure 10: Both QCOW2 and FVD work well with sparse images. The disk space usage grows as needed.

writes its on-disk metadata in a critical region that blocks all other (disk or non-disk) I/O activities. Because disk I/Os are slow, the blocking time can severely affect other I/O activities. To measure disk I/O's impact on network I/O, the experiments in Figure 8 run two benchmarks concurrently: PostMark to drive disk I/O and Iperf [11] to drive network I/O. The VM's configuration is the same as that for the "virtio-partition" group in Figure 6(a), except that it is configured with two virtual CPUs to ensure that CPU is not the bottleneck. The VM uses the paravirtualized *virtio* interface for both network and disk. Figure 8 reports the network throughput achieved by Iperf during the file creation phase of PostMark. The "Hypervisor" bar means running Iperf directly in native Linux, without virtualization and without running PostMark. The "VM-disk-idle" bar means running Iperf in a VM alone, without running PostMark. In this case, the choice of image format does not matter. The "RAW", "FVD", and "QCOW2" bars means running Iperf and PostMark in a VM concurrently, while using different image formats. The network throughput achieved with FVD is 253% higher than that with QCOW2. QCOW2's synchronous access to its on-disk metadata is a known issue, but it is difficult to fix due to the complexity of QCOW2.

4.1.4 An Experiment that Favors QCOW2

To understand the "potential" of QCOW2, we deliberately design one experiment that allows QCOW2 to work most efficiently. This experiment's setup is similar to that for the "virtio-partition" group in Figure 6(a), but PostMark is configured to work on about 900MB data stored on a 1.2GB partition of the virtual disk. QCOW2's metadata for these 900MB data can completely fit in QCOW2's in-memory metadata cache. Moreover, due to the small size of this 1.2GB partition (1.2GB is only 1.7% of the 73GB physical disk), the disk seek distance on this 1.2GB partition is short, which makes QCOW2's VBA-IBA mismatch problem much less severe than that shown in Figure 7. This condition favors QCOW2. In Figure 9, the difference between "QCOW2-cold" and "QCOW2-warm" is that, before running PostMark, "QCOW2-warm" warms up QCOW2 by writing once to all unused free spaces in the guest ext3 file system. As a result, all the storage spaces that will be used by PostMark have already been allocated in the QCOW2 image, and all QCOW2's metadata are cached in memory. Both conditions favor QCOW2. Even in this case, FVD still outperforms QCOW2. The file creation throughput of "FVD-cold" is 252% and 16% higher than that of "QCOW2-cold" and "QCOW2-warm", respectively. The transaction throughput of "FVD-cold" is 19% and 1.6% higher than that of "QCOW2-cold" and "QCOW2-warm", respectively.

4.1.5 Sparse Image Support

The experiment in Figure 10 demonstrates that both QCOW2 and FVD support sparse images. In this experiment, the VM image is stored in a host ext3 file system, whose sparse file capability is leveraged by FVD to support sparse images. In the VM, a script repeatedly compiles the Linux kernel by executing *"make"* and *"make clean"*. Figure 10 records the the actual disk spaces used by QCOW2 and FVD after each round of compilation. Both QCOW2 and FVD work well sparse images. The actual disk spaces consumed grow as needed, and are much smaller than the full size of the virtual disk. QCOW2 uses about 23% more disk spaces than FVD, because QCOW2 allocates disk spaces at a granularity larger than ext3 does, i.e., 64KB vs. 4KB.

4.1.6 Different Ways of Using FVD

Figure 11 compares the different ways of using FVD. The experiment setup is similar to that for the "virtio-partition" group in Figure 6(a). "Min-size FVD" enables the optimization described in 3.3.2, which reduces the backing image to its minimum size. "Zero-aware FVD" enables the optimization described in 3.3.3, which ini-

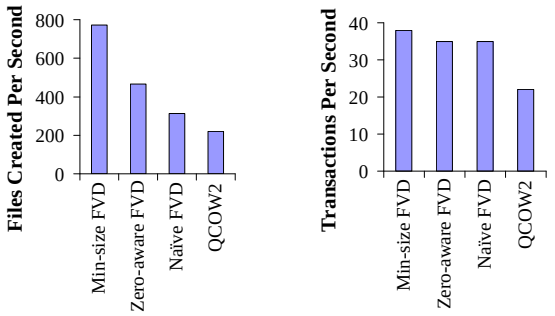


Figure 11: Comparison of different ways of using FVD.

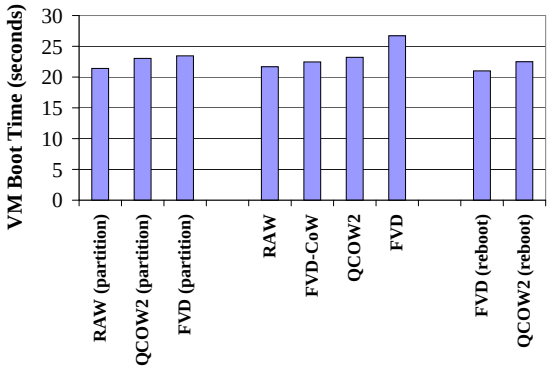


Figure 12: Time to boot a VM under different configurations.

tializes a block's state to S_{in_FVD} if the block's original content in the backing image is filled with zeros. "Naive-FVD" enables neither optimization. The results show that "min-size FVD" has significant advantages. The file creation throughput of "min-size FVD" is 65% and 146% higher than that of "zero-aware FVD" and "naive-FVD", respectively. The transaction throughput of "min-size FVD" is 8.6% higher than that of "zero-aware FVD" and "naive-FVD". "Zero-aware FVD" is less efficient than "min-size FVD" because the guest file system's metadata blocks are non-zeros and writing to those blocks in "zero-aware FVD" still requires updating FVD's on-disk bitmap. On the other hand, even the less efficient "naive-FVD" outperforms QCOW2 by 42% and 59% in file creation and transaction, respectively.

4.2 Copy on Read

Figure 12 compare the time it takes to boot a VM under different configurations. "RAW (partition)", "QCOW2 (partition)", and "FVD (partition)" store the VM image on a raw partition in the host. All the other configurations store the VM image on a host ext3 file system. The "FVD-CoW" configuration only enables copy-on-write, whereas the other "FVD" configurations enable both copy-on-write (CoW) and copy-on-read (CoR). As the VM boots, CoR-enabled FVD stores about 30MB data into the FVD data file due to the copy-on-read behavior. The boot time of the "FVD" bar is 3.5 seconds higher than that of the "QCOW2" bar. This is mainly due to the overhead of the host ext3 file system rather than the overhead of FVD itself. Without using the host ext3 file system, the VM boot time of "FVD (partition)" is only 0.4 seconds longer than that of "QCOW2 (partition)". Copying data into a sparse image file stored on ext3 incurs a high overhead, because it requires storage space allocation. However, the overhead of CoR is a one-time effect. Rebooting an FVD image (the "FVD (reboot)" bar) is actually faster than rebooting an QCOW2 image (the "QCOW2 (reboot)" bar).

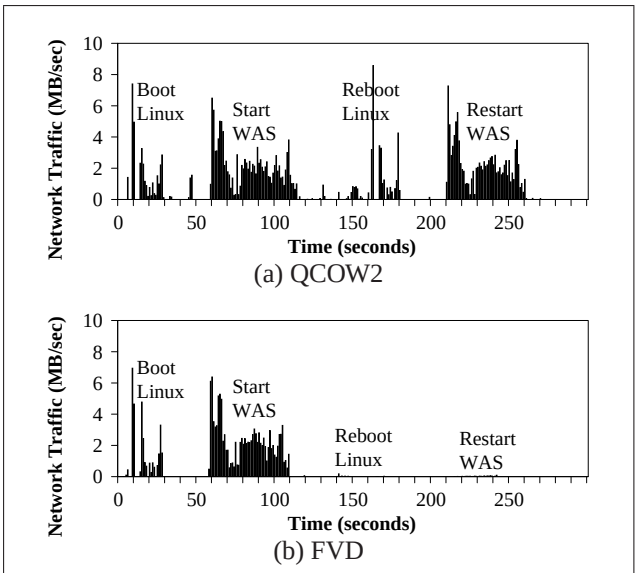


Figure 13: FVD's copy-on-read feature helps reduce network traffic and I/O load on the NAS server.

CoR avoids repeatedly reading a data block from NAS, by saving a copy of the returned data on DAS for later reuse, which helps avoid generating excessive network traffic and I/O load on NAS. This effect is shown in Figure 13. This experiment boots a Linux VM and then starts IBM WebSphere Application Server (WAS) in the VM, during which we measure the network traffic for reading data from the backing image. The VM and WAS are rebooted once to test the effect of CoR. With FVD, the first boot of WAS takes 51.0 seconds, and the second boot of WAS takes 42.1 seconds. The second boot is faster and introduces no network traffic because CoR during the first boot already saved the needed data on DAS. With QCOW2, the first boot and the reboot both take about 55.1 seconds, and generate roughly the same amount of network traffic.

4.3 Adaptive Prefetching

Figure 14 evaluates FVD's adaptive prefetching capability. The x-axis shows the time since a VM S_1 boots. The y-axis shows the network traffic generated by read-

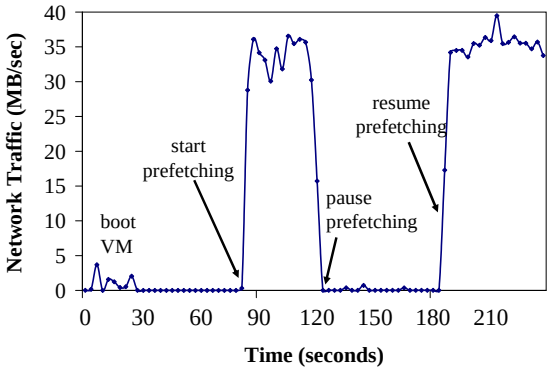


Figure 14: FVD automatically finds resource idle time to perform prefetching.

ing data from the backing image stored on NAS. Controlled by a policy, prefetching automatically starts at time 80 seconds. At time 120 seconds, another VM S_2 on another host starts to run PostMark on a EBS-like persistent virtual disk whose image is stored on NAS. The disk I/Os generated by the two VMs cause a contention on NAS, and the FVD driver of VM S_1 quickly detects that the prefetching throughput drops below the specified lower limit (20MB/s) and pause prefetching. From time to time, it temporarily resumes prefetching to check if the contention disappears. Finally, as PostMark running in VM S_2 finishes at time 178 seconds, the FVD driver of VM S_1 resumes prefetching permanently.

5 Related Work

FVD supports copy-on-write (CoW), copy-on-read (CoR), and adaptive prefetching. These features are motivated by the use cases in a Cloud, especially, instant VM creation and instant VM migration. An image template in a Cloud is used to create many VMs repeatedly. Therefore, it is worthwhile to perform one-time offline optimizations on the image template in exchange for superior VM runtime performance. Many optimizations in FVD follow this philosophy (see Section 3.3). These optimizations are novel and critical for performance. Previous works use the CoW and CoR techniques in various settings [5, 6, 15, 21], but do not study how to optimize the CoW and CoR techniques themselves.

Despite the widespread use of VMs and the availability of VM image format specifications to the public, there is no published research on how image formats impact disk I/O performance. Our study reveals that all popular CoW image formats (including QCOW2 [16], VirtualBox VDI [25], VMware VMDK [26], and Microsoft VHD [17]) use an index structure to translate between VBAs and IBAs, and 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 its VBA. This mismatch between VBAs and IBAs invalidates many optimizations implemented in guest file systems (see the discussion in Section 2.3).

Existing virtual disks support neither CoR nor adaptive prefetching. Some virtualization solutions do support CoR or prefetching, but they are implemented for certain specific use cases, e.g., virtual appliance [6], mobile computing [15], or VM migration [21]. By contrast, FVD provides CoR and prefetching as standard features of a virtual disk, which can be easily deployed in many different use cases.

Collective [6] provides virtual appliances, i.e., 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 VBAs and IBAs. As a result, it 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 experiencing a high latency. Our experiments show that it is hard to set a proper "percentage" to reliably detect contentions. Because storage servers and disk controllers perform read-ahead in large chunks for sequential reads, a very large percentage (e.g., 90%) of a VM's prefetching reads hit in read-ahead caches and experience a low latency. When a storage server becomes busy, the "percentage" of requests that hit in 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.

Both Xen [7] and VMware [18] support live VM migration if the VM image is stored on NAS or SAN. QEMU can migrate a VM image stored on DAS, but it takes a pre-copy approach, i.e., first copying the virtual disk and then making the VM fully functional at the new location. We argue that FVD's copy-on-read approach is more suitable for storage migration, because 1) storage has much more data than memory and hence pre-copy takes a long time, and 2) unlike memory accesses, disk I/Os are less sensitive to the network latency experienced during copy-on-read.

In terms of the CoW and CoR techniques, the VM migration work by Sapuntzakis et al. [21] is the closest to FVD. It also uses a bitmap to track the states of data blocks, but performs no optimizations to reduce the overhead in updating the on-disk bitmap, which is critical to disk I/O performance, as shown in Figure 11. It suffers from the residual dependency problem, as pointed out by Bradford et al. [5], i.e., after migration, a VM at the new location still depends on data at the old location. FVD solves this problem using prefetching.

CoW has also been implemented in logical volume managers and file systems [4, 19], where data locality issues exist, similar to the VBA-IBA mismatching problem

in CoW virtual disks. Peterson [20] and Shah [22] propose techniques to put the CoW data close to the original data, assuming they are stored on the same disk. This is not an issue in a Cloud because the CoW data are stored on DAS while the original data are stored on NAS.

Several existing works are related to FVD's adaptive prefetching algorithm. MS Manners [8] measures the progress of a low-importance process and suspend it when its progress is low so that it does not degrade the performance of high-importance processes. TCP Nice [24] and TCP-LP [14] use network resources conservatively to transfer low-priority traffic.

6 Conclusion

This paper presents the FVD image format and its device driver for QEMU. FVD distinguishes itself in both performance and features. It supports copy-on-write, copy-on-read, and adaptive prefetching. These features enable instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. To achieve high performance, the design of FVD debunks the common practice of mixing the function of storage space allocation with the function dirty-block tracking. Experiments show that FVD significantly outperforms QCOW2, owing to the aggressive optimizations.

Although FVD is motivated by our unsatisfied needs in the IBM Cloud, it is equally applicable in both Cloud and non-Cloud environments. At the very least, FVD's copy-on-write feature can be a high-performance alternative to QCOW2. FVD is mature and we actively seek for adoption in the QEMU mainline (see http://sites.google.com/site/tangchq/qemu-fvd).

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Amazon Web Services. `http://aws.amazon.com/`.

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

[4] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. In *FAST*, 2003.

[5] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg. LiveWide-Area Migration of Virtual Machines Including Local Persistent State. In *VEE*, 2007.

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

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005.

[8] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *SOSP*, 1999.

[9] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, 2009.

[10] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[11] Iperf Network Measurement Tool. `http://en.wikipedia.org/wiki/Iperf`.

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

[13] 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.

[14] A. Kuzmanovic and E. W. Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. In *INFOCOM*, 2003.

[15] M. Satyanarayanan et al. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing*, 2007.

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

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

[18] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*, 2005.

[19] Z. Peterson and R. Burns. Ext3cow: A Time-Shifting File System for Regulatory Compliance. *ACM Transactions on Storage*, 1(2):190—212, May 2005.

[20] Z. N. J. Peterson. *Data Placement for Copy-on-Write Using VirtuaL Contiguity*. PhD thesis, University of California at Santa Cruz, 2002.

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

[22] B. Shah. *Disk performance of copy-on-write snapshot logical volumes*. PhD thesis, University Of British Columbia, 2006.

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

[24] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *OSDI*, 2002.

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

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

Appendix 1: FVD Guarantees Data Integrity in the Event of a Host Crash

Suppose the following events happen in a sequence: (1) the VM submits a disk write request; (2) the FVD driver (after some processing) acknowledges the successful completion of the write operation; and (3) the host immediately loses power. After power recovers and the VM reboots, the VM's next read to the same block should get the content written before the failure. Otherwise, data integrity is compromised, which may fail applications (e.g., databases) that rely on strong data integrity.

When a user boots a VM, QEMU allows the user to specify different caching policies for a virtual disk: *cache=none*, *cache=writethrough*, or *cache=writeback*. In Linux, *cache=none* translates to the *O_DIRECT* flag for disk I/Os, and *cache=writethrough* translates to the *O_DSYNC* flag for disk I/Os. Both of them guarantees that the data are safely stored on the physical disk before the write operation finishes. By design, QEMU does not guarantee data integrity for the *cache=writeback* configuration (in exchange for better disk I/O performance), because some dirty data may be cached in memory and not yet flushed to the physical disk when the host crashes. The discussion below assumes that the virtual disk is configured with either *cache=none* or *cache=writethrough*.

Figure 4 on page 5 shows a simplified view of the FVD image format. When performing copy-on-write, copy-on-read, or prefetching, the FVD driver needs to separately update the FVD data file and the FVD metadata file. Therefore, a host crash between the two updates might compromise data integrity. Below, we prove that this is not the case and FVD preserves data integrity regardless of when the host crashes.

We first prove that the basic version of FVD described in Section 3.2 preserves data integrity in the event of a host crash. We then extend the proof to cover the optimized version of FVD described in Section 3.3.

Data Integrity in the Basic Version of FVD

A bit in the bitmap can be in one of two states, $S_{in\_backing}$=0 or S_{in_fvd}=1, which means the corresponding block's content is in the backing image or the FVD data file, respectively. A block's state can only change from $S_{in\_backing}$ to S_{in_fvd}, and can never change from S_{in_fvd} to $S_{in\_backing}$. Three operations can change a block's state from $S_{in\_backing}$ to S_{in_fvd}: copy-on-write, copy-on-read, and prefetching. We first discuss copy-on-write.

Copy-on-write happens when the FVD driver handles a disk write request from the VM. For brevity, the discussion below assumes that the write request spans over two disk blocks (d_1, d_2). Let $bit(d_1)$ and $bit(d_2)$ denote the states of d_1 and d_2 in the bitmap, respectively. We further assume that, before the write operation, $bit(d_1) = S_{in\_backing}$ and $bit(d_2) = S_{in\_backing}$. Other cases with more data blocks involved and different initial states can be analyzed in a way similar to the example below.

It involves the following sequence of operations when the basic version of FVD (described in Section 3.2) handles a write request:

- *FVD:W1:* the VM issues a write request for two blocks (d_1, d_2).
- *FVD:W2:* the FVD driver stores d_1 in the FVD data file.
- *FVD:W3:* the FVD driver stores d_2 in the FVD data file.
- *FVD:W4:* the FVD driver updates the on-disk bitmap state $bit(d_1)$ from $S_{in\_backing}$ to S_{in_fvd}.
- *FVD:W5:* the FVD driver updates the on-disk bitmap state $bit(d_2)$ from $S_{in\_backing}$ to S_{in_fvd}.
- *FVD:W6:* the FVD driver acknowledges to the VM the completion of the write operation.

Note that *FVD:W2* and *FVD:W3* may be performed in a single write. We separate them for a worst-case analysis. Similarly, $bit(d_1)$ and $bit(d_2)$ may belong to the same block, and hence *FVD:W4* and *FVD:W5* may be performed in a single update. We separate *FVD:W4* and *FVD:W5* for a worst-case analysis.

The host may crash after any step above. We will prove that FVD preserves data integrity regardless of when the crash happens. Specifically, FVD introduces no more complication than what may happen to the RAW image format. In other words, we prove that the data

integrity of FVD is as good as the data integrity of the RAW image format.

If the VM uses the RAW image format, handling this disk write involves the following sequence of operations:

- *RAW:W1:* the VM issues a write request for two blocks (d_1, d_2).
- *RAW:W2:* the RAW driver stores d_1 on disk.
- *RAW:W3:* the RAW driver stores d_2 on disk.
- *RAW:W4:* the FVD driver acknowledges to the VM the completion of the write operation.

Note that *RAW:W2* and *RAW:W3* may be performed in a single write. We separate them for a worst-case analysis.

Now we consider all possible failure cases with FVD. Note that before the VM's write operation, the "old" contents of d_1 and d_2 are stored in the backing image. After the VM's write operation finishes successfully, the "new" contents are stored in the FVD data file.

- Fail after *FVD:W1*. In this case, FVD's behavior is equivalent to having a host crash with the RAW image format after *RAW:W1*. The effect is that the write operation is simply lost, which is an allowed, correct behavior, since the driver did not yet acknowledge to the VM the completion of the write.
- Fail after *FVD:W2*. In this case, d_1 is written to the FVD data file, but $bit(d_1)$ is not updated and remains $S_{in\_backing}$. After reboot, the VM's next read to d_1 gets its old content from the backing image, as if d_1's new content in the FVD data file does not exist. This behavior is correct and is equivalent to having a host crash with the RAW image format after *RAW:W1*. The effect is that the write operation is simply lost, which is an allowed, correct behavior, since the driver did not yet acknowledge to the VM the completion of the write.
- Fail after *FVD:W3*. Similar to the one above, after reboot, the VM's next read to d_1 or d_2 gets the old content from the backing image, as if the new content in the FVD data file does not exist. This behavior is correct and is equivalent to having a host crash with the RAW image format after *RAW:W1*.
- Fail after *FVD:W4*. After reboot, the VM's next read to d_1 gets its new content from the FVD data file, whereas the VM's next read to d_2 gets its old content from the backing image (because $bit(d_1) = S_{in\_fvd}$ and $bit(d_2) = S_{in\_backing}$). This behavior is correct and is equivalent to having a host crash with the RAW image format after *RAW:W2*.
- Fail after *FVD:W5*. After reboot, the VM's next read to d_1 or d_2 gets the new content from the FVD data file (because $bit(d_1) = S_{in\_fvd}$ and $bit(d_2) = S_{in\_fvd}$). This behavior is correct and is equivalent to having a host crash with the RAW image format after *RAW:W3*, i.e., the write operation is completed but not yet acknowledged.
- Fail after *FVD:W6*. After reboot, the VM's next read to d_1 or d_2 gets the new content from the FVD data file (because $bit(d_1) = S_{in\_fvd}$ and $bit(d_2) = S_{in\_fvd}$). This behavior is correct and is equivalent to having a host crash with the RAW image format after *RAW:W4*.

The analysis above shows that FVD's copy-on-write operation preserves data integrity. Following a similar process, it can be proven that FVD also preserves data integrity during copy-on-read and prefetching, by following the correct update sequence—first updating the FVD data file and then updating the on-disk bitmap.

Data Integrity in the Optimized Version of FVD

Next, we show that the optimizations described in Section 3.3 do not compromise data integrity in the event of a host crash.

- **In-memory bitmap**: This is merely a performance optimization. The FVD driver immediately update the on-disk bitmap whenever such an update is required to preserve data integrity. For bitmap updates that do not compromise data integrity, the FVD driver may update the physical disk lazily.
- **Free writes to no-backing blocks**: Since the bitmap has no state bits for the no-backing blocks, writing to those blocks need not update the bitmap. Therefore, this optimization introduces no complication for data integrity.
- **Free writes to zero-filled blocks**: Since those blocks' states are initialized to S_{in_fvd} and never change afterwards, this optimization introduces no complication for data integrity.

  a host crash does not affect those blocks.
- **Free copy-on-read**: 3.3.4 When the FVD driver copies a block from the backing image into the FVD data file due to copy-on-read, it does not immediately update the block's state in the on-disk bitmap from $S_{in\_backing}$ to S_{in_fvd}, which reduces disk I/O overhead. This does not compromise data integrity in the event of a host crash, because the block's content in the FVD data file is identical to that in the backing image and reading from either place gets the correct data. When the VM

overwrites a block that was previously saved in the FVD data file due a copy-on-read operation, the block's state in the on-disk bitmap is immediately updated from $S_{in\_backing}$ to S_{in_fvd}, by following the operation sequence *FVD:W1—FVD:W6* described above, which guarantees data integrity.

- **Free prefetching**: In terms of the impact on data integrity, this case is the same as the case above.
- **Zero overhead once prefetching finishes**: Once prefetching finishes, the FVD driver no longer reads or writes the bitmap. It simply passes through all disk I/O requests issued by the VM to the RAW driver. Therefore the data integrity of an FVD image is identical to that of a RAW image.

In summary, the analysis shows that both the basic version and the optimized version of FVD preserve data integrity in the even of a host crash.