# Compact Image Support in Fast Virtual Disk (FVD)

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

## Abstract

Fast Virtual Disk (FVD) is a flexible, high-performance image format designed for QEMU. It supports compact image, copy-on-write, copy-on-read, and adaptive prefetching. This paper focuses on the compact image feature of FVD. The other features of FVD are described in a separate paper [6]. Source code and documentation of FVD are available at the FVD Web site [2].

## 1 Introduction

Many popular image formats support compact images, including QEMU QCOW2 [3], QEMU QED [5], VirtualBox VDI [7], VMWare VMDK [8], and Microsoft VHD [4]. FVD is designed to address some fundamental limitations of these image formats. Below, we summarize several distinguishing features of FVD that help FVD achieve far superior performance.

**Eliminate the need for a host file system.** FVD can store a compact image directly on a logical volume without using a host file system. FVD automatically extends the size of the logical volume on demand as more storage spaces are needed. This feature solves multiple problems introduced by using a host file system to store an image, including runtime overhead, data fragmentation, and compromising data integrity.[1] This capability is not available in any other image formats, as they all need a host file system in order to support storage over-commit. Using a host file system incurs significant overheads. Experiments in [6] show that a RAW image on ext3 is 50-63% slower than a RAW image on a raw partition.

**Optimize on-disk data layout.** FVD optimizes the data layout of a compact image to mimic that of a RAW image, which helps alleviate the data fragmentation problem and achieve better performance. This capability is not available in other compact image formats, as they all use a data naive layout strategy, i.e., allocating 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 naive strategy may cause severe fragmentation and increase disk seek distances. Specifically, when a guest OS creates or resizes a guest file system, it writes out the guest file system metadata, which are all grouped together and put at the beginning of the image by existing compact image formats, despite the fact that the guest file system metadata's virtual block addresses are deliberately scattered across the virtual disk for better reliability and locality, e.g., co-locating inodes and file content blocks in block groups. As a result, it causes a long disk seek distance between accessing the metadata of a file in the guest file system and accessing the file's content blocks. Experimental results in Figure 7 of [6] show that the disk seek distance with a QCOW2 image is 5.6 times longer than that with an FVD image.

**Disable compact image when desired.** FVD's compact image data layout is optional rather than mandatory. FVD allows a copy-on-write image to be stored on a raw partition with data layout identical to that of a RAW image, i.e., not using the compact image data layout when high-performance is preferred. This capability (i.e., a copy-on-write image with a RAW-image-like data layout) is not available in any other image formats, as they mandate the use of a compact image for all features, including copy-on-write.

The rest of the paper is organized as follows. Section 2 presents the design of FVD. Section 2 reports the current implementation status. Section 4 concludes the paper.

## 2 Design of Compact Image Support in FVD

FVD is designed to address the issues discussed in the previous section. It use two on-disk metadata structures:

- **A one-level lookup table to implement compact image**. One entry in the table maps the virtual disk address of a chunk to an offset in the FVD image where the chunk is stored. The default size of a chunk is 1MB, as that in VirtualBox VDI (VMware VMDK and Microsoft VHD use a chunk size of 2MB). For a 1TB virtual disk, the size of the lookup table is only 4MB, which can be easily cached in memory.

[1] One example of the host file system's data integrity problem is discussed at http://lwn.net/Articles/348739/.

- **A bitmap to implement copy-on-write**. A bit in the bitmap tracks the state of a block. The bit is 0 if the block is in the backing file, and the bit is 1 if the block is in the FVD image. The default size of a block is 64KB, as that in QCOW2. To represent the state of a 1TB backing file, it only needs a 2MB bitmap, which can be easily cached in memory.

By design, the chunk size is larger than the block size in order to reduce the size of the lookup table. A smaller lookup table can be easily cached in memory. The bitmap is small because of its efficient representation. Using a smaller block size improves runtime disk I/O performance because, during copy-on-write and copy-on-read, a complete block need be read from the backing file and saved to the FVD image even if the VM only accesses part of that block.

### 2.1 Eliminate the Need for a Host File System

FVD supports storing a compact image on a host file system, a logical volume, or a raw partition, among which a logical volume is the preferred choice. Storing a compact image on a raw partition does not support storage over-commit, because the full storage space need be allocated ahead of time. Storage over-commit means that, e.g., a 100GB physical disk can be used to host 10 VMs, each with a 20GB virtual disk. This is possible because not every VM completely fills up its 20GB virtual disk.

Storing a compact image on a host file system introduces multiple problems, including runtime overhead, data fragmentation, and compromising data integrity. Traditionally, a host file is optimized for storing small files (KBs) rather than large images (GBs), and is not designed with the support for virtualization in mind. Unfortunately, all existing compact image formats (including QCOW2, QED, VMDK, VDI, and VHD) store a compact image on a host file system. Even more ironically, when a host file system is used to store the image, the compact image capability of those image formats are no longer needed, because almost every modern file system already support compact files, including GFS, NTFS, FFS, LFS, ext2/ext3/ext4, reiserFS, Reiser4, XFS, JFS, VMFS, and ZFS. Storing a RAW image on those file systems will automatically get a compact image and support storage over-commit.

By contrast, FVD can store a compact image directly on a logical volume without using a host file system. Initially, the size of the logical volume is small, e.g., only 10% of the full size of the virtual disk. FVD automatically extends the size of the logical volume on demand as more storage spaces are needed. Even if all physical volumes on the local disks run out of storage space, FVD can add physical volumes from NAS or SAN and guarantees an uninterrupted execution of the VM. This behavior is fully customizable for different storage systems, as FVD can be configured to invoke a user-supplied script to add storage spaces. Extending the size of a logical volume incurs a minimal overhead. Our measurement shows that it takes only 0.15 seconds for the *lvextend* tool to grow a logical volume, and this operation is performed infrequently, e.g., by adding 1GB storage space each time.

### 2.2 Optimize On-Disk Data Layout

All existing compact image formats use a naive data layout strategy, i.e., allocating 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 naive strategy may cause severe fragmentation and increase disk seek distance, as discussed in Section 1.

When storing a compact image on a logical volume, FVD strives to optimizes the data layout to mimic that of a RAW image, which helps alleviate the data fragmentation problem and achieve better performance. We explain this using one concrete example.

Amazon EC2 [1] provides two virtual disks to a VM: a 10GB root disk and a 160GB data disk. The data disk is initially empty. Doing storage over-commit for the 160GB data disk may result in significant cost savings for a Cloud service provider. Suppose the target is do storage over-commit by a factor of two, i.e., on average a 160GB data disk consumes no more than 80GB actual storage space because not every VM completely fills up its 160GB data disk. Actually, many VMs may even never use their data disks.

Suppose FVD stores the 160GB data disk on a logical volume and the initial size of the logical volume is 20GB. When it runs out of space on the logical volume, FVD adds 10GB storage space to the logical volume each time until it reaches the full size of 160GB. The hope is that on average VMs use far less than 160GB. However, in the worst case it is possible that every VM on a host uses more than 80GB and the host runs out of storage space on its local disks. In this case, FVD can add physical volumes from NAS or SAN to guarantee that the VMs get the storage space they need and perceive no disruption.

Although the initial logical volume size (20GB) is much smaller than the full data disk size (160GB), it already allows FVD to produce a data layout that mimics the data layout of a RAW image. This is depicted in Figure 1, where $M_1$ $M_2$ and $M_3$ are the guest file system's metadata, and $D_1$, $D_2$, and $D_3$ are contents of files in the guest file system. Note that the mapping from the RAW image layout to the FVD image layout is done in a way that attempts to maintain the relative order of data chunks.

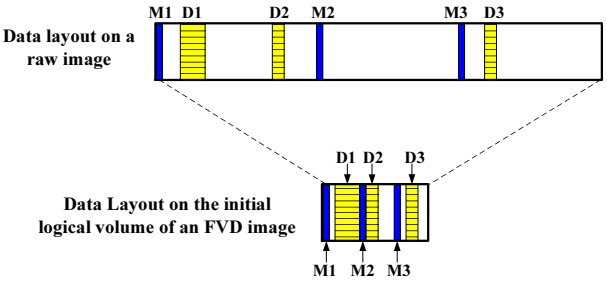


Figure 1: Optimized data layout of a compact FVD image.

Specifically, let $v$=160GB and $u$=20GB denote the full size of the virtual disk and the initial size of the logical volume, respectively. A virtual data chunk with virtual disk address $x$ in the RAW image is mapped to a physical chunk at offset $y = x \cdot \frac{u}{v}$ of the initial logical volume of the FVD image. If that physical chunk is already occupied, it is mapped to the next available physical chunk (this is the case with data chunk $D_2$ in the figure). If no physical chunk after $y$ is available, it wraps around and searches for an available physical chunk starting from the beginning of the logical volume.

If the initial 20GB segment of the logical volume is completely used up, FVD extends the logical volume by adding a new 10GB segment. Within this new 10GB segment, FVD attempts to maintain the relative order of new data chunks written, using a method similar to that depicted in Figure 1. In other words, FVD strives to maintain the order of data chunks within each segment but does not maintain the order of data chunks across segments, because doing so would require FVD to perform expensive online defragmentation. Regardless, this strategy helps improve performance because the data layout within a segment is close to sequential. Moreover, with a carefully chosen initial logical volume size, the majority of VMs may never need to grow the logical volume beyond its initial size and hence their data layouts are close to sequential.

Another optimization is to continuously measure the degree of fragmentation for the current segment $X$ under use, and add a new segment earlier if $X$ is fragmented and close to full, e.g., 80% full instead of 100% full. When a segment is close to full, it is harder to find an unused physical chunk at the ideal location and using a random physical chunk tends to increase fragmentation. If the logical volume eventually reaches its 160GB limit, FVD will come back to use those fragmented free physical chunks.

### 2.3 Using Journal to Reduce Metadata Update Overhead

The lookup table and the bitmap can be enabled separately to provide the function of compact image or copy-on-write. When both are enabled, they work together to provide a compact copy-on-write image. FVD uses a on-disk journal to store updates to the lookup table and the bitmap. The journal allows FVD to update a block's metadata (i.e., both the lookup table and the bitmap) in a single disk write, which not only is more efficient but also ensures their consistency.

## 3 Implementation Status

All features described in this paper have been implemented except the optimization depicted in Figure 1. Actually, we are also exploring several alternatives other than the one in Figure 1. That piece of code will be released publicly once we are done with more benchmarking and tuning.

## 4 Conclusion

FVD's compact image capability has significant advantages over all other compact image formats. First, FVD can store a compact image directly on a logical volume without using a host file system, which solves multiple problems introduced by a host file system, including runtime overhead, data fragmentation, and compromising data integrity. Second, FVD optimizes the data layout of a compact image to mimic that of a RAW image. Finally, FVD's compact image data layout is optional rather than mandatory. If desired, FVD allows a copy-on-write image to be stored on a raw partition with data layout identical to that of a RAW image.

Overall, FVD is the most flexible and best performing image format, not only for QEMU but also among all image formats supported by any hypervisor. In addition to its advanced compact image capability, FVD's copy-on-write, copy-on-read, and adaptive prefetching capabilities also significantly outperform the alternatives [6]. We strongly recommend the adoption of FVD into the QEMU mainline. More information about FVD can be found at the FVD Web site [2].

## References

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

[2] FVD Website. `https://researcher.ibm.com/researcher/view_project.php?id=1852`.

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

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

[5] QEMU QED. http://wiki.qemu.org/Features/QED.

[6] C. Tang. FVD: a High-Performance Virtual Machine Image Format for Cloud with Copy-on-Write, Copy-on-Read, and Adaptive Prefetching Capabilities. https://researcher.ibm.com/researcher/view_project.php?id=1852, October 2010.

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

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