Chapter Two

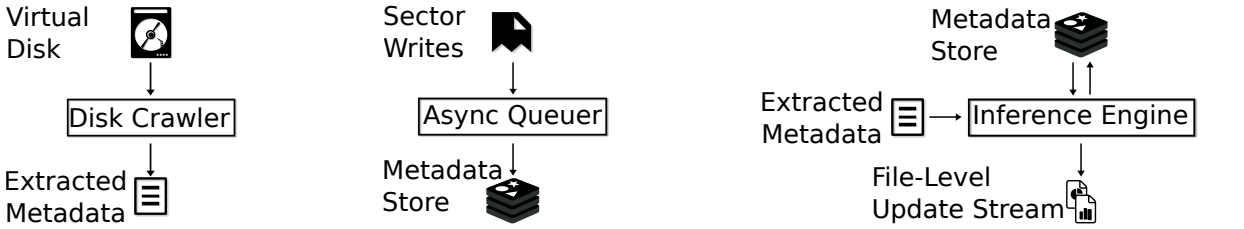
Distributed Streaming Virtual Machine Introspection (DS-VMI)

In this chapter, we explain the design and implementation of our mechanism that exposes live virtual disk state in near-real-time to monitors. The mechanism was designed to operate without any support from within the monitored system—the guest operating system in a VM. This means that it operates without paravirtualization, guest modifications, or specific guest configurations. We call this mechanism *distributed streaming VM introspection* (DS-VMI). DS-VMI infers file system modifications from sector-level disk updates in near-real-time and efficiently streams them to distributed or centralized monitors.

Our approach is based on the fact that virtual disks are emulated hardware. Hence, every disk sector write already passes through at least the hypervisor system. Remember that it may pass through many hosts depending on whether or not the backing storage is across the network. We transparently clone this stream to a userspace process on the hypervisor host, or any of the intermediate hosts. This minimally interferes with the running VM instances—generally they occasionally see higher latency writes. Only guest-flushed updates result in sector writes. Thus, we only handle file system updates that are flushed from the VM instances. Updates that have not been flushed, and therefore represent dirty state in guest memory, are outside the scope of this dissertation.

DS-VMI resembles classic VMI [42], with two crucial differences. First, we support streaming introspection from instances distributed throughout the cloud. The design of DS-VMI and its interfaces directly stems from this distributed setting. Second, instead of performing introspection synchronously, we always perform it asynchronously. We are able to minimize stalling of the VM during introspection because our goal is not intrusion detection: we are only monitoring guest actions, rather than trying to prevent tainted ones by enforcing policy. As stated in Chapter 1, enforcement of policy is outside the scope of this dissertation.

The rest of this chapter is organized as follows. Section 2.1 provides a very high-level view of the architecture of DS-VMI. Section 2.2 describes the requirements necessary from a hypervisor, file system, and guest kernel for DS-VMI. These requirements include theoretical underpinnings



(a) Disk Crawler: Offline or live, produces metadata used for inference.

(b) Async Queuer: Queues instance writes into an in-memory queue.

(c) Inference Engine: loads metadata and interprets queued writes.

Figure 2.1: Three-stage DS-VMI architecture.

grounded in a theory of file systems, and technical requirements such as guest kernel parameters. This section culminates with the description of an implementation of cloning writes using a modern hypervisor. Section 2.3 deals with bootstrapping metadata about file systems within a storage device, primarily a virtual disk, necessary for the DS-VMI runtime. It includes a discussion of the three file systems we support: `NTFS`, `ext4`, and `FAT32`. Section 2.4 describes the asynchronous architecture and an optimization to reduce overhead. Section 2.5 describes the DS-VMI runtime from receiving a sector-level write, to understanding its file-level implications. Section 2.6 describes how DS-VMI supports live attachment to an already running VM, and also detachment for when operators wish to drop overhead to zero. Section 2.7 describes extending the API of an existing cloud to integrate introspection. An evaluation of the overhead of this mechanism on four representative file-level workloads is provided in Section 2.8. We finish with a description of what DS-VMI is *not* designed for, and its fundamental limitations in Section 2.9.

2.1 Overview of the DS-VMI Prototype

We have built an experimental prototype of DS-VMI, for the KVM hypervisor [58] using QEMU [11] for disk emulation. Via custom introspection code, our prototype supports commonly-used file systems including `ext4`, `NTFS`, and `FAT32`. We show that our solution works out of the box with unmodified cloud images of Ubuntu Server 14.04 LTS as provided by Canonical [17]. Our prototype has a three-stage architecture. The first stage is an indexing step, performed once per unique virtual disk (not needed for clones), for initializing. The other two stages are specific to the runtime of each VM instance executing in a cloud, as shown in Figure 2.1. We summarize these stages below, with details in Sections 2.3 through 2.5:

1. **Crawling and indexing virtual disks. (Figure 2.1(a))** This stage generates indexes of file system data structures via a component called *Disk Crawler*. The indexes are generated live or loaded at instance launch from a central store such as the image library storing virtual disks.

2. **Capture and cloning of disk writes. (Figure 2.1(b))**
   This stage is implemented via a userspace helper process called *Async Queuer* that receives a stream of write events from a minimally modified QEMU. Normally, it runs at the hypervisor hosting the VM for low latency, but it can technically receive this stream over the network.

3. **Introspection and translation. (Figure 2.1(c))**
   In this stage, the *Inference Engine* interprets sector writes, reverse maps them to file system data structures, and produces a stream of file update events. It operates either at the hosting hypervisor, or across the network.

To ground our discussion, lets follow an example write originating within a guest VM, and to better understand how monitoring might work, imagine a monitor application interested in monitoring a file called `/home/monitorme/clock.jpg`. Imagine that the `clock.jpg` file gets updated every 5 seconds by a webcam pointed at a clock with a second hand. Thus, our monitor should see modifications to this file every 5 seconds. DS-VMI can freely discard the rest of the virtual disk I/O because no other registered monitor exists.

Let us examine what happens when the file is modified, and for brevity we follow a single virtual disk block write. First, an instruction executed by the guest VM traps into the KVM kernel module as shown in Figure 2.2 by the arrow moving out of the box labeled, "Execute natively in Guest Mode," into the box labeled, "Handle Exit." The KVM kernel module identifies whether or not the trapped instruction is for an I/O operation. Because it is an I/O operation, the KVM kernel module invokes the userspace process emulating I/O devices for the guest VM—in this case a Qemu process. The steps described here are highlighted in Figure 2.2.

Before issuing the ioctl to the KVM kernel module to return to guest mode, the write is copied to the DS-VMI process running as a set of userspace processes not shown in Figure 2.2. It arrives at the Async Queuer shown in Figure 2.1(b). At this point DS-VMI takes over analyzing the write to determine its file-level implications in the introspection phase shown in Figure 2.1(c). The first step requires reverse mapping the partition table. DS-VMI identifies that the write is within a partition of interest—specifically the `ext4` formatted partition containing the `clock.jpg` file.

The write is then passed to an `ext4` specific handler that was initialized by crawling the containing virtual disk. The handler performs a series of reverse mappings to understand the individual file being modified and its full path. The first step in the process is to identify if the block represents data or metadata. Metadata for `ext4` includes the superblock, the block group descriptor table, inode bitmaps, block bitmaps, inode tables, and extent trees. In this case, the write is data so the first reverse mapping yields the inode responsible for this data block. The next reverse mapping yields the file name `clock.jpg` contained within the directory data block for directory `monitorme`. The directory data block reverse maps to an inode and this process recursively continues for the two other parent directories: `home`, and `/`. In the actual implementation, we optimize this lookup with a reverse index on full path names which avoids the recursion.

Thus, the third phase of DS-VMI as shown in Figure 2.1(c) has performed four reverse mappings: one for the initial data block to the responsible inode, and three for the three parent direc-

tories. DS-VMI now knows that this data block belongs to a file not a directory, and that the full path of the file is `/home/monitorme/clock.jpg`. The next step is to determine if any registered monitors are interested in this file-level update. In this example, there is a registered monitor for this file and DS-VMI uses interprocess communication via `cloud-inotify` (Chapter 3) to notify the monitoring process that there is new data to consume. The monitoring process receives the data block, updates its copy of the file, and refreshes the screen if enough new data has been written. There may be more block writes that are needed before the file is displayable. This process repeats every 5 seconds as new images are written to disk. If data blocks are written without metadata structures pointing to them, they must remain buffered by DS-VMI until it associates them with a file. For exposition, we assumed the data block was immediately associable with a file.

2.2 Hypervisor, Kernel, and File System Requirements

If virtual writes cross a network, introspecting them exactly like a deep packet inspecting firewall becomes the most logical choice. We assume techniques are already known for building such solutions. The difference between analyzing network traffic and introspecting file systems comes down largely to one of memory. Large amounts of memory may be needed for buffering writes and reconstructing complex file system metadata. If we ignore the more trivial network case, we are left with the more difficult position of adding introspection support to a hypervisor. This approach is explored in this dissertation, and this section specifically answers the question *what are the properties of the hypervisor, kernel, and file system which make correct introspection feasible*? By correct, we mean that DS-VMI provides guaranteed consistent views of file-level state.

2.2.1 Hypervisor Virtual Storage Hooks

We assume that either virtual disk writes come over the network, which makes them amenable to deep packet inspection techniques, or the hypervisor provides hooks to inspect each write. For correctness guarantees in later sections to hold, it is important that the hypervisor not coalesce or drop writes. Otherwise properties which we assume true about the guest kernel or guest file system may prove false due to hypervisor optimizations. Thus, we assume the existence of a method to essentially duplicate each virtual write to DS-VMI.

This is not a giant leap. Many modern hypervisors already support targeting more than one virtual storage device in RAID 1 style configurations. For example, KVM has a built-in command called `drive-mirror` [92]. The purpose of this command is to mirror a virtual disk to another location, and it has a mode for duplicating writes. As another example, VMware has a storage driver called Mirror [122]. Its job is to mirror a drive to another location, and it accomplishes this by duplicating live writes. For yet another example, Xen has a highly configurable `blktap` API [90] for writing custom virtual disk implementations. Using `blktap`, one could rapidly implement mirroring as needed by introspection.

Of course, configurations exist which neither send writes over the network, nor through the costly layers of a hypervisor. Performance critical VMs have storage directly dedicated to them. In such a configuration, the writes and management of storage are directly handled by the VM. Obviously, DS-VMI can not operate in such a setting. However, generally DS-VMI does not apply in such settings anyways, and these configurations are rare in cloud computing. First, the performance sensitive application probably can not tolerate the potential latency and overhead from deep monitoring. Second, such direct access configurations are rare because they enable hardware-level attacks by guests, prevent useful management services such as snapshotting by the cloud, and expose the guest to underlying hardware failures. If a guest ties itself to local hardware, migration becomes impossible should that hardware begin to fail.

Extending an Existing Hypervisor

Figure 2.2 shows the design of KVM's I/O path when guest I/O's trap into the hypervisor. By design, all I/O's get handed over to a userspace helper program—QEMU. Such a design is ideal for DS-VMI because the writes already come to userspace for inspection and further routing—very similar to the proposed design of packet filters [80]. We just need to copy the write stream to a userspace DS-VMI process from the emulator of the disk. Although QEMU handles both reads and writes, we only need to introspect write operations.

We use the open source virtual storage introspection engine for QEMU/KVM as described in [98] with a few key differences which make this feasible within a cloud framework such as OpenStack [85]. The original implementation used QEMU's tracing framework to capture writes. QEMU's tracing framework was designed for debugging purposes, and is not built into the QEMU executable by default. No vendor provides production QEMU builds with tracing activated—indeed QEMU advises against including tracing into production builds. Thus, the original method of capturing disk writes was unsuitable for production deployments especially in modern clouds.

Our new implementation [126] changes a total of 50 lines of C within QEMU, and extends a pre-existing hypervisor command called `drive-backup` [91]. The QEMU command `drive-backup` originally implemented copy-on-write, point-in-time virtual disk snapshotting. We added a mode to `drive-backup` called `continuous` which duplicates every write to another target. When introspecting, we set this target to a network drive using the Network Block Device (NBD) protocol. We have a compact, less than 1,000 lines of C code, implementation of NBD which passes writes along not to a real storage device, but instead to the DS-VMI framework.

When `drive-backup` executes at a QEMU hypervisor, writes get duplicated over a TCP socket via the NBD protocol as shown in Figure 2.3. This provides flexibility—introspection may run locally or remotely depending on the load of a node hosting a guest VM. The writes terminate at the NBD Queue as shown in Figure 2.3. Presumably this queue resides on the same node performing introspection, although technically the introspection process can also exist on a separate node, accepting raw writes over TCP.

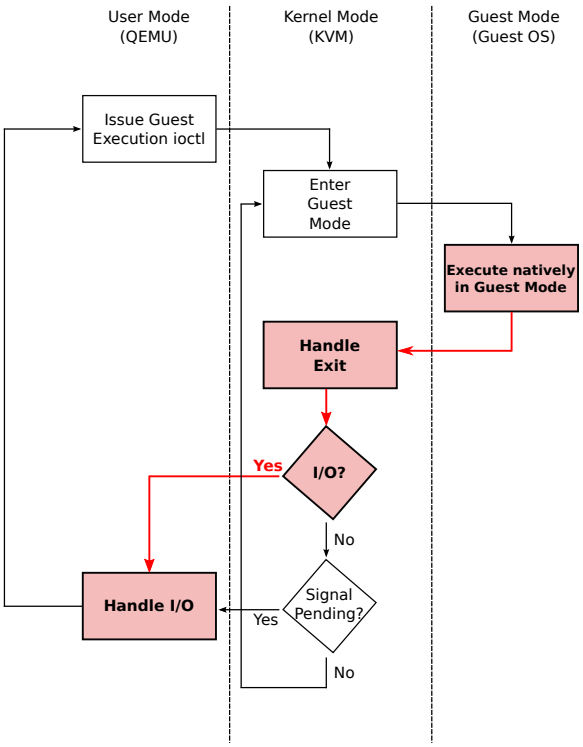


Figure 2.2: KVM architecture showing userspace, kernel, and physical boundaries. Figure reconstituted from [58].

2.2.2 Guest Kernel Invariants

Kernels are free to reorder and coalesce writes from file systems as long as they do not violate correctness. However, ordering of writes and observing all intermediate state is necessary to fulfill the file system invariants described in Section 2.2.3. Hence, kernels must respect these invariants otherwise DS-VMI may lose track of critical file system state. In addition, the tunable timeout before disk flushes are forced directly affects the minimum latency in DS-VMI. Thus, there are two questions explored in this section. First, *what are the minimal set of invariants needed from a guest kernel to ensure DS-VMI correctness*, and *how can a guest kernel assist DS-VMI*? Although DS-VMI requires no modifications whatsoever, it seems likely that either guest kernel tuning for better introspection, or direct guest modifications à la paravirtualized kernels will become standard for introspecting clouds.

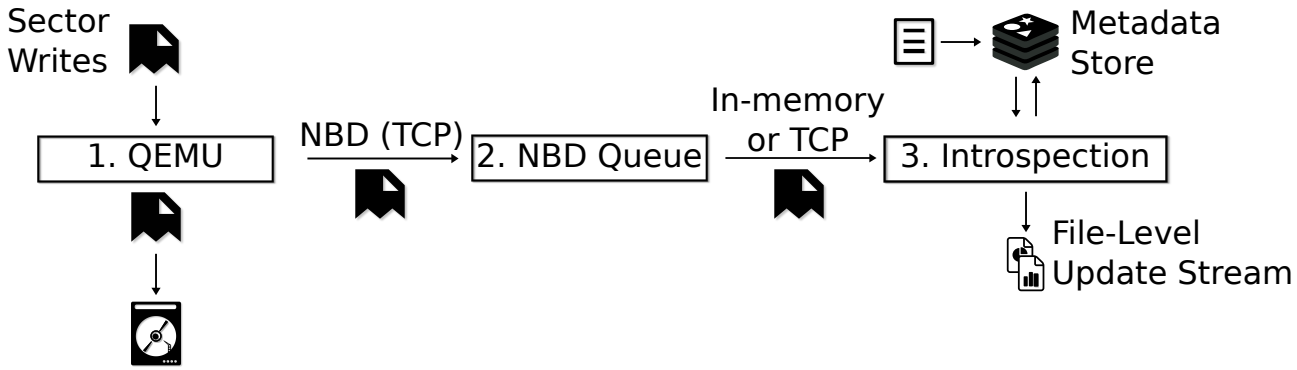


Figure 2.3: QEMU duplicates writes over a TCP socket using the Network Block Device protocol as its application-layer protocol. These writes queue for introspection, and are transferred for introspection either via an in-memory transport or another TCP hop.

Write Buffer Flush Frequency

Operating systems introduce the notion of a page cache to optimize both the read and the write path of slow persistent storage. Reads often exhibit temporal locality—the notion that there is a high likelihood of accessing recently read data in the near future again. Writes also exhibit temporal locality, thus sending every single write to a slow storage layer is not logical. Page caches serve as a fast, in-memory store of data for reading and writing backed by persistent storage. Temporal locality magnifies their benefit. They directly limit the writes visible to DS-VMI, and add additional latency from the time of write to its storage-level visibility.

At a minimum, DS-VMI requires just enough writes to be visible such that the file system invariants as laid out in Section 2.2.3 are maintained. Ideally, to minimize latency and the chance of violating a necessary file system invariant, the kernel disables its page cache for perfect visibility of every write to DS-VMI. However, the performance cost of disabling a page cache is prohibitively costly, and we can not expect this to become a best practice.

This extra latency causes incorrect decisions based on old information. For example, in elastic cloud computing scenarios, compute elements often scale to ensure service level agreements (SLAs). By acting upon old information, they scale to handle the load of the past—not the load of the present. Thus, there is a high likelihood of underprovisioning or overprovisioning when information is not exposed quickly to the introspection infrastructure. Overprovisioning directly causes waste, and underprovisioning indirectly causes waste by delaying or causing failures in dependent systems.

Either paravirtualized drivers communicating information to the hypervisor before it gets flushed, or tuning of guest kernel flush parameters to make them more frequent is needed to ensure timely introspection results. For example, `ext3` forces flushing of metadata every 5 seconds by default. On the other hand, `ext4` forces flushing of critical data every 30 seconds by default. This led to the discussion of using `fsync` much more aggressively by userspace [28]. These default cache flush

intervals fundamentally limit the minimum latency of an introspection framework. Note that for *correctness*, latency tuning is not necessary.

Write Barriers and Ordering

Kernels reorder writes for various reasons, often for minimizing the seek time across a set of writes. The famous Elevator algorithm [62] tries to order and dispatch writes in a minimal seek time ordering based on the current direction of the disk head. As mentioned in the previous section, in the worst case this reordering could lead to confusing DS-VMI. However, if file system invariants are left intact, which they must be to maintain file system consistency, DS-VMI will be unaffected by kernel-level write reordering. In fact, to enforce file system invariants, the kernel offers write barriers. Write barriers ensure critical operations are flushed to disk with a certain ordering.

Write barriers are a technique for a kernel to ensure that file system metadata structures get updated in the proper order on disk. For introspection, write barriers ensure correctness by providing tight guarantees on the ordering of metadata updates. Introspection correctness is dependent on the guaranteed ordering of metadata updates from a guest kernel. The requirements for file system correctness are analyzed in the next section.

2.2.3 File System Invariants

Traditional file systems were not designed to support real-time introspection as a consumer of their writes. The properties a file system must guarantee to make real-time introspection possible are not obvious, but have been studied by the file system theory community. Sivathanu et al. [109] derive the properties a file system must exhibit for this type of file-level inference. The guarantee a file system must maintain, from [109], is,

$$\{t\,(A^x) = k\}_D \Rightarrow \{t\,(A^x) = k\}_M$$

In words, the type (k) of a block (A) on disk (D)—metadata or data—matches the file system view in memory (M) at some point in time (x). This ensures that incorrect inferences can not occur; a data block associated with a file can not be mistaken for a metadata block associated with the same file. For this guarantee to hold, a file system must exhibit, "a strong form of reuse ordering," and metadata consistency [109]. *Strong reuse ordering* means that the file system must commit the freed state of any block and its allocation data structures to disk before reuse, and *metadata consistency* means maintaining all file system metadata with a set of invariants [109] (e.g. directory entries point to valid file metadata) to ensure correct operation. A practical example occurs upon file deletion. The data blocks of a file must be marked as free and their corresponding inodes also marked free before any of those blocks are reused on disk. If they are not marked as free on disk before reuse, DS-VMI might incorrectly believe that their type, and therefore their contents, have not changed upon future writes.

What class of applications could transition to using the file-level update stream provided by DS-VMI? Any application that can resume or operate on data flushed to disk works with both whole disk snapshotting and DS-VMI. Recent research [46] reports that modern desktop applications frequently flush data to disk which means many common applications already work with snapshotting and by extension DS-VMI. Abstractly, a disk flush requires buffered I/O operations to be flushed to disk and the file system to have both metadata consistency and data consistency on disk after the flush. *Data consistency* [109] means that all flushed data safely resides on disk and the contents of the corresponding data blocks match the metadata structures pointing to them. Following a reboot, an application reading from the file system sees the side effects of all flushed I/O operations. Our technique preserves flushed file-level updates, which means applications properly flushing critical data to disk can safely use data obtained via DS-VMI.

Order of Metadata Updates

Given the analytical and theoretical underpinnings of the last section, how do we know whether or not a file system supports them? The easiest method is auditing that file system's source code. For `ext4` we have this luxury, for `FAT32` and `NTFS` we have no such luxury and must hope that they abide by certain rules. All that a guest kernel must ensure is *strong reuse ordering*, and *metadata consistency*. Basically, all metadata block type changes must directly match the view of the guest kernel's page cache at commit time in the near future. According to Sivathanu et al. [109], file systems such as `FAT32`, and `ext3` have features we desire. However, their study is not exhaustive and without access to source code difficult to prove for `NTFS`. Based on observations, we have gained high confidence that `NTFS` is crash consistent and therefore safe for DS-VMI.

2.2.4 Correctness of DS-VMI Relative to Snapshotting

DS-VMI offers the same fidelity at a file-level as current state-of-the-art snapshotting methods. We never report a file-level update that has not occurred within a file system on a virtual disk—there are no *false positives*. In addition, we never miss a file-level update that has been flushed to disk—there are no *false negatives*. If a snapshot of a disk is taken at a point in time, the inferred file-level updates DS-VMI reports are consistent with file-level changes observable in that snapshot. For example, if a tool such as Tripwire [118] runs on the snapshot and runs on a virtual disk maintained with updates from a stream of DS-VMI provided file-level updates, the results are identical. This does not mean that our technique provides the equivalent of a snapshot at a block-level.

Figure 2.4 superimposes our technique over snapshotting, and we use this figure to develop an example illustrating the difference between inferring file-level updates and snapshotting. Figure 2.4 shows an initial snapshot created at time τ, and another snapshot at time τ'. Assume that these snapshots occur at a block-level, which is how snapshotting virtual disks happens today. The snapshot at time τ' differs from the snapshot at time τ by exactly the disk blocks that were written in the intervening time period. This is usually accomplished via a technique called copy-on-write which copies disk blocks only when overwritten. Instead of providing a stream of disk block writes,

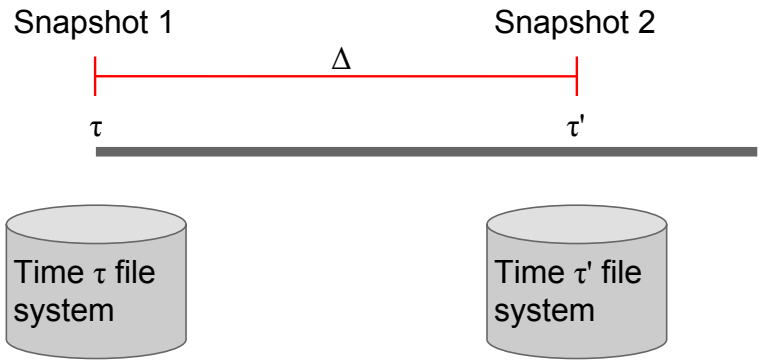


Figure 2.4: Combining file-level updates recorded as Δ with the file system at time τ yields a file system equivalent to the file system at time τ'. The file systems may be mutating state on disk when the snapshots occur.

our technique provides a stream of file-level updates in between these two time points which we refer to as Δ. Thus, Δ is a *well-ordered set* of inferred file-level updates. Applying our stream of file-level updates Δ to the snapshotted file system at time τ yields an equivalent view of the file system within the snapshot taken at time point τ'. With high probability, this view will not be consistent at a block-level with the snapshot at time point τ', and an example scenario leading to inconsistency is explained below. However, this does not mean that anything is lost *semantically* from the snapshot method.

Snapshotting techniques do not understand disk blocks at a semantic level, thus they blindly follow all disk block writes. File-level updates are reported when disk block writes are visible from a file system perspective. If disk block writes are invisible from a file system perspective, they provide no higher semantic meaning because they are uninterpretable and contribute no information to the file system. Disk block writes that are missing from inferred file-level updates are the file system invisible disk block writes. One example that leads to an inconsistency is file creation that fails to complete before the snapshot. The snapshot contains disk blocks associated with the file, but, if mounted, the file would not appear in the file system because the creation was not completely flushed to disk. A snapshot derived from inferred file-level updates would not contain disk blocks associated with the file because its creation did not complete.

2.3 Crawling Initial Virtual Disk State

DS-VMI requires a one-time crawl of each virtual disk before it commences real-time streaming of subsequent file system updates. This crawl builds a map of the virtual disk for DS-VMI so that it can very quickly infer the file system objects being modified from incoming sector updates at runtime. DS-VMI supports crawling offline virtual disks in addition to dynamically crawling online, running

```
{
'type'       : [BSON_STRING, 3] 'mbr'
'gpt'        : [BSON_BOOLEAN] false
'sector'     : [BSON_INT32] 0
'partitions' : [BSON_INT32] 1
'mbr'        : [BSON_BINARY, 512] ...
}
```

Figure 2.5: An MBR BSON document example showing details about an entire virtual disk, starting from the MBR.

VM instances.

The offline case typically occurs when a virtual disk is first added to a cloud. Upon addition of the virtual disk, the *Disk Crawler* produces serialized metadata associated with the virtual disk's partitions and stores it alongside the virtual disk in a virtual disk library. It only needs to run once per unique virtual disk.

In the online case, DS-VMI live-attaches to an already-running VM. Here, the Disk Crawler crawls and indexes the virtual disk while it is also being modified by the executing VM. To handle the transient dirty state and not miss any new state, the dynamic component of DS-VMI, described in Section 2.4, buffers the incoming write stream from the start of the disk crawl. Once the Disk Crawler finishes indexing, DS-VMI replays the dynamic write stream buffer to obtain the latest updates since the time of crawling and finally catches up to the live, real-time write stream updates.

The disk crawler is implemented in C with file system indexers for `ext2`, `ext3`, `ext4`, `NTFS`, and `FAT32`. The entire disk is crawled, and serialized metadata is produced for each active partition containing a valid supported file system. The metadata is formatted as serialized BSON [15] documents and compressed using `gzip`. We chose BSON because it is compact, supports binary data, has an open specification, and has been successfully used in scalable systems such as YouTube [48].

Figure 2.5 shows a sample BSON document. We do not construct a single nested BSON document per drive because that would require loading the entire document into memory, and thus would not scale to large virtual disks. Instead, we represent the metadata as a collection of documents, each of which can be loaded separately. We currently serialize five document types: MBR, Partition, File System, BGD (not used with `NTFS`), and File. A File document contains a serialized form of the related inode and, in the case of a directory, a list of files in the directory.

Disk analysis starts at the Master Boot Record (MBR) that contains a partition table. Each entry in this table may point to a valid primary partition or to a linked list of secondary partitions. As an illustrative example, we consider what happens when an `ext4` partition is detected. An `ext4` partition is analyzed by first examining and serializing its superblock. The `s_last_mounted` field identifies the most recent mount point of this file system, which helps in recreating pathnames. The superblock points to the Inode Table, which captures a wealth of information about each file. In `ext4` the "`i_block`" field is typically the header of an *extent tree*. Immediately following the header

are pointers to extents, each of which in turn points to a set of data blocks for the file. The disk crawler collects and serializes necessary metadata from all allocated files by walking the inode table and directory entries. Directory entries are contained in the data blocks of directories and map file paths to inodes.

The NTFS disk format poses special challenges. In this format, the Master File Table (MFT) plays a role analogous to the Inode Table in ext4. It stores File Records, which are the equivalent of ext4 inodes. However, the MFT itself is managed as a file and can become fragmented throughout a disk. The positions of metadata cannot be computed in advance with simple offsets. In addition, there are proprietary intricacies that are not documented openly and can only be inferred via the trial and error process of reverse engineering. In spite of these challenges, we have been successful in implementing support for NTFS.

The FAT32 disk format is the simplest of the three main file systems considered, but comes with its own challenges. For example, there are no separate inodes for files. File metadata is embedded within the containing directory. This means that it is impossible to assign unique identifiers to files in FAT32 that are divorced from path. In spite of this, FAT32 is well and openly documented now, and was the quickest to implement—by someone originally unfamiliar with introspection and our pre-existing codebase.

2.3.1 Impact on Virtual Image Library Operations

As described at the beginning of this chapter and in-depth in the last section, DS-VMI requires metadata from the file systems present in a virtual disk. This metadata is then used for inference when VM instances are instantiated. But how will a modern cloud obtain this metadata? Virtual disk libraries already collect metadata and act upon metadata during two key operations: check in of a new virtual disk, and checkout during instance creation.

DS-VMI's metadata naturally fits with these two operations. DS-VMI extends the check-in process with a crawl over the virtual disk to extract relevant metadata from guest file systems. When a cloud user requests that a VM instance boot from a particular virtual disk, we can retrieve the crawled metadata, and ship it to DS-VMI anywhere in the cloud. The rest of this section explores the feasibility of this approach with standard VM images by answering the following question:

What is the overhead of crawling for, transferring, and loading metadata? With our prototype implementation we found typical crawl time of 18-26 seconds, metadata size of 8-20 MB (compressed), and metadata load time of 30-73 seconds with a standard Ubuntu 12.04 LTS server virtual disk image, and a Windows 7 virtual disk image. Keep in mind that both disk images are 20 gibibytes in size, which takes approximately 3 minutes to transfer assuming an unloaded 1 gigabit network link. Moving our 20 mebibytes of metadata—0.2 seconds—and loading it in 73 seconds is dwarfed by the large virtual disk transfer time. In other words, the load has plenty of time to finish while the disk streams in the background. Of course, this assumes loading is a synchronous operation and we must wait for it to complete. We can instead buffer writes while the load continues in the background, and begin processing the buffered writes once the load completes.

ext4

Used (GB)	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
6.4	64	8.2	20.30 (1.91)	30.15 (0.15)
8.4	70	9.4	21.43 (2.00)	35.98 (0.20)
11	77	11	22.25 (1.68)	41.54 (0.25)
13	83	12	23.20 (1.85)	47.24 (0.45)
15	90	13	24.22 (1.74)	52.91 (0.43)
17	96	15	25.85 (1.83)	59.19 (0.43)

NTFS

Used (GB)	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
6.9	67	14	17.93 (2.12)	43.74 (0.20)
8.9	73	15	18.13 (2.01)	49.58 (0.23)
11	79	16	18.39 (2.26)	55.10 (0.24)
13	85	18	18.51 (1.85)	60.60 (0.36)
15	92	19	18.72 (2.01)	66.21 (0.65)
17	98	20	19.48 (2.47)	72.68 (0.82)

Table 2.1: Metadata size uncompressed, compressed, crawl time and load time into Redis (20 runs) as a function of used virtual disk space. Used is used disk space reported by `df`, Raw is the uncompressed metadata, gzip is compressed metadata with `gzip --best`, Crawl is the time taken to index a disk, and Load is the time to load metadata into Redis.

Checking in an Image

The critical factor affecting the check in process of a VM image into a virtual disk library is crawl time. Crawl time is dictated by the amount of metadata in the file systems of the virtual disk, its sprawl across the disk, and the amount of lookups required to unpack the file system metadata structures. Large amounts of sequentially packed metadata take time only in unpacking the data structures of the underlying file system. Metadata spread throughout the disk causes many seeks and could potentially slow down the crawl process. To examine this effect, and also to get a sense of the overhead of crawling in general, we measured crawl times with two file systems: `ext4`, and `NTFS`. `NTFS` packs all of its metadata into a single global area of the disk known as the Master File Table (MFT). In contrast, `ext4`, typically spreads metadata across block groups. Thus, for crawling, we expect the central, sequential MFT in `NTFS` to perform better.

The metadata collected by the Disk Crawler grows with used disk space because it serializes a mapping from sectors to files, and with the number of reachable live inodes representing files in the system because each inode is serialized as well. Table 2.1 shows how metadata grows as a function of used disk space for `ext4` and `NTFS`. We increased used disk space by writing a single large file with random data inside a virtual disk. The relationship for both file systems is linear in the used disk space because we use a canonicalized form of metadata independent of file system. The raw

ext4

inodes	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
127,786	64	8.2	20.30 (1.91)	30.15 (0.15)
250,000	101	12	21.19 (1.85)	41.53 (0.32)
500,000	178	19	22.52 (1.29)	63.56 (0.29)
750,000	256	27	23.87 (1.68)	85.87 (0.61)
1,000,000	333	35	26.08 (1.75)	109.68 (0.68)
1,250,000	410	44	27.05 (1.47)	132.12 (0.68)

NTFS

inodes	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
103,152	67	14	17.93 (2.12)	43.74 (0.20)
250,000	106	16	24.36 (2.64)	58.53 (0.24)
500,000	174	19	34.95 (2.19)	83.02 (0.29)
750,000	242	23	44.04 (2.93)	108.31 (0.56)
1,000,000	309	26	54.62 (2.65)	132.96 (0.66)
1,250,000	377	29	63.99 (2.52)	159.32 (0.45)

Table 2.2: Metadata size uncompressed, compressed and load time into Redis (20 runs) as a function of number of inodes. The headers are the same as in Table 2.1 except the first column is a unitless count of live inodes in the file system rather than used disk space.

metadata grows at a rate of 6–7 megabytes per gigabyte of used disk space, but only 1 megabyte compressed. NTFS crawls are quicker because its on-disk metadata is in a single, linear stretch on disk. Load times are also comparable, but NTFS is slower because it starts with approximately 500 megabytes more used disk space. In addition, NTFS often has multiple names for the same file, further magnifying metadata. As we stated earlier in this section, we can mask these load times by concurrently loading metadata while the virtual disk transfers over the network to its host system. Crawling is a one-time operation, so the overhead in time shown in Table 2.1 is amortized over the lifetime of the virtual disk image in a cloud.

Checking out an Image

When checking out a virtual disk from a virtual disk library, the key parameter deciding overhead is the size of the metadata we need to send for DS-VMI to run close to the destination of the virtual disk. In this section, instead of exploring the relationship between used disk space and crawled metadata size, we explore the relationship between live paths and crawled metadata size. In other words, we artificially boosted the number of active inodes in our file systems trying to maximize the size of extracted metadata. In this manner, we have deliberately tried to stack the deck against DS-VMI. We again explore with both ext4 and NTFS.

Many portions of metadata are filled with zeroes, and runs of similar numbers or duplication within pathnames are common. Therefore, the extracted metadata compresses to generally less than 10% of its uncompressed size. In a cloud datacenter with gigabit or 10-gigabit networking, transferring on the order of 10 megabytes worth of compressed metadata will take less than one second. This seems very reasonable given the optimistic 18 seconds – 3 minutes necessary to transfer a 20 gibibyte virtual disk. As a percentage of used disk space, the compressed form of our extracted metadata is always less than 1 percent.

Table 2.2 shows how metadata grows as a function of live files in `ext4` and `NTFS`. These files were created by the `touch` command within a single directory. Once again, we see a linear relationship for both file systems. The raw metadata grows at a rate of approximately 323 bytes per file for `ext4` and 285 bytes for `NTFS`. Compressed, this overhead drops to approximately 34 bytes per file for `ext4` and 13 bytes for `NTFS`. For `ext4`, the average path length was 32 characters, and for `NTFS`, it was 24 characters $(1,250,000$ cases$)$. This implies approximately 8 more bytes are in an `ext4` path, although with BSON serialization each string also has a type byte and a field of four bytes representing length. Thus, we were adding approximately 13 bytes more length per file in `ext4` then `NTFS`. Paths are stored at least twice: once as a full path associated with a file in an index, and once as a directory entry. Because of this duplication, we have an extra 26 bytes approximately per active inode with `ext4`. This accounts for the discrepancy in raw metadata size between `ext4` and `NTFS` as we scale up the number of active inodes.

2.3.2 Crawling a Virtual Disk

We make no assumptions about a virtual disk and currently our DS-VMI prototype supports two different forms of partition table, along with three file systems. The old-style MBR partition table was the simpler and first format our prototype supported, and GUID Partition Table (GPT) support was implemented in the fall of 2014 by a two-person graduate student team initially unfamiliar with our prototype codebase. Because of the modular design, they were able to implement support quickly and independently in their own module for parsing GPT partition tables. In this section we explain the typical layout of a Linux-based virtual drive image.

Figure 2.6 shows a typical virtual drive layout. At the start of the disk is an MBR that contains a partition table. Each entry in this table may point to a valid primary partition or to a linked list of secondary partitions. In Figure 2.6, two partitions are defined. One is for "Swap," which is not analyzed further because it represents memory pages and not an actual file system; the other is a valid `ext4` partition. The `ext4` partition is analyzed by first examining and serializing its superblock. Figure 2.6 shows that even a cursory look into the superblock of an `ext4` partition reveals a lot about the underlying file system. For example, the `s_last_mounted` field identifies the most recent mount point of this file system; this information helps in recreating pathnames.

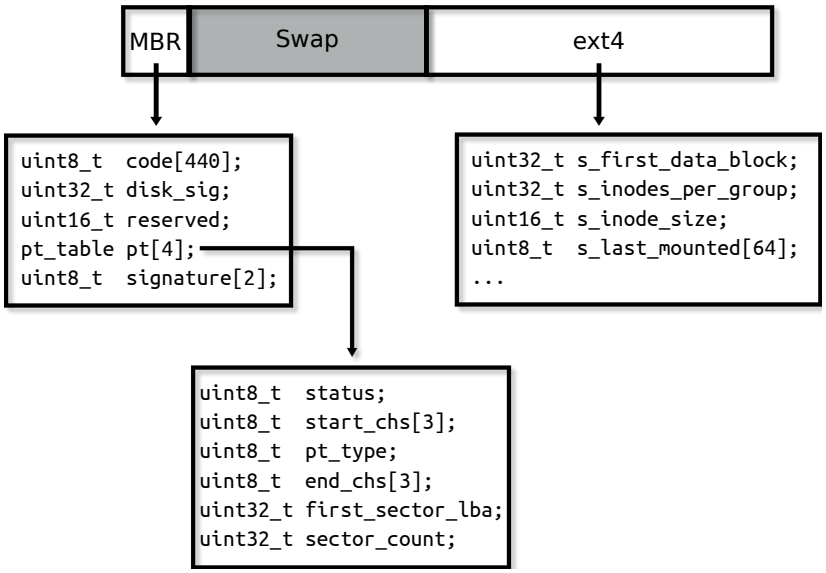


Figure 2.6: View of a raw disk split into partitions by the partition table within an MBR at the start of the disk.

2.3.3 An Example Journaling File System: `ext4`

A deeper look into the `ext4` partition is shown in Figure 2.7. `ext4` file systems are divided into block groups, which are listed in the Block Group Descriptor (BGD) Table. Each block group is associated with a group of inodes in the Inode Table and a group of data blocks on disk. Our static analysis serializes all of this information, because it could change during runtime and affect the emitted stream of file-level updates.

As shown in Figure 2.7, the Inode Table captures a wealth of information about each file. In `ext4` the "`i_block`" field is typically the header of an *extent tree*. Immediately following the header are `eh_entries`—pointers to extents—each of which in turn points to a set of data blocks for the file. The inode shown in Figure 2.7 describes a directory; its data blocks contain filenames and inode numbers for each file in the directory. By walking the inode table and directory entries, starting from the root inode, the disk analyzer collects and serializes the necessary metadata from all allocated files within the file system.

2.3.4 An Example Closed Source File System: `NTFS`

Out of the three file systems considered in this chapter, `NTFS` has the most complex on-disk layout and data structures. Implementing introspection for `NTFS` was further exacerbated because `NTFS` has no open specification, and thus the least amount of documentation out of the three file systems we support in our DS-VMI prototype. This made the initial parsing of `NTFS` difficult and excruciatingly

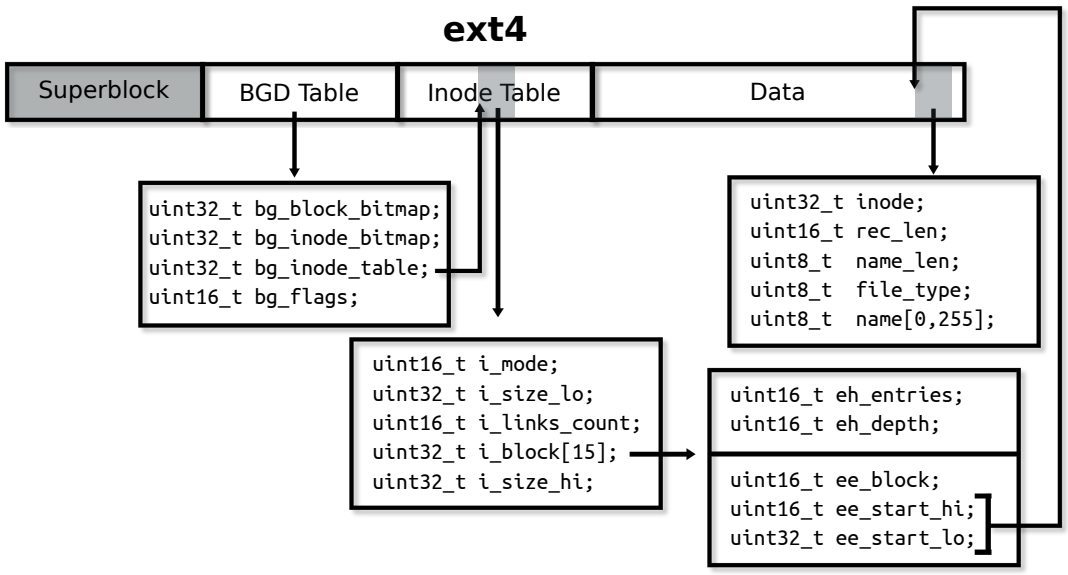


Figure 2.7: View of an ext4 partition and critical metadata on-disk structures. The structure in the data section is a directory entry.

tedious. What little documentation that exists openly about NTFS comes from reverse engineering efforts, and none of it is complete.

Figure 2.8 shows the general layout at a high level of a NTFS partition. Similar to FAT32, NTFS keeps all critical metadata in a single location—the special Master File Table (MFT) file. Everything in NTFS is considered a file, including all portions of the disk dedicated to metadata. This is a very clean design, which allows the file system to manage even metadata as just another file entry. However, this early design decision led to an accumulation of complex metadata headers and types. For example, as shown in Figure 2.8 the metadata for a single file, called a File Record, is split amongst several different data structures. Most File Records consist of at least 3-4 metadata structures, each with their own header.

Bootstrapping an NTFS file system involves parsing values in the first sector of the partition. This portion of the disk is also a file and appears in the MFT with the special name of $Boot. These special files, which includes the MFT itself with the name $MFT, do not appear in the normal directory hierarchy visible to users. But, they are manageable by NTFS as simple files. Critical fields of the $Boot file are shown in Figure 2.8 and include an offset to the $MFT, the size of a sector in bytes, the amount of sectors making up a cluster, the number of clusters per File Record in the $MFT. When crawling files, many, if not all, of their attributes need parsing. Three key attribute data structures for a File Record, pertinent to DS-VMI, are shown in Figure 2.8.

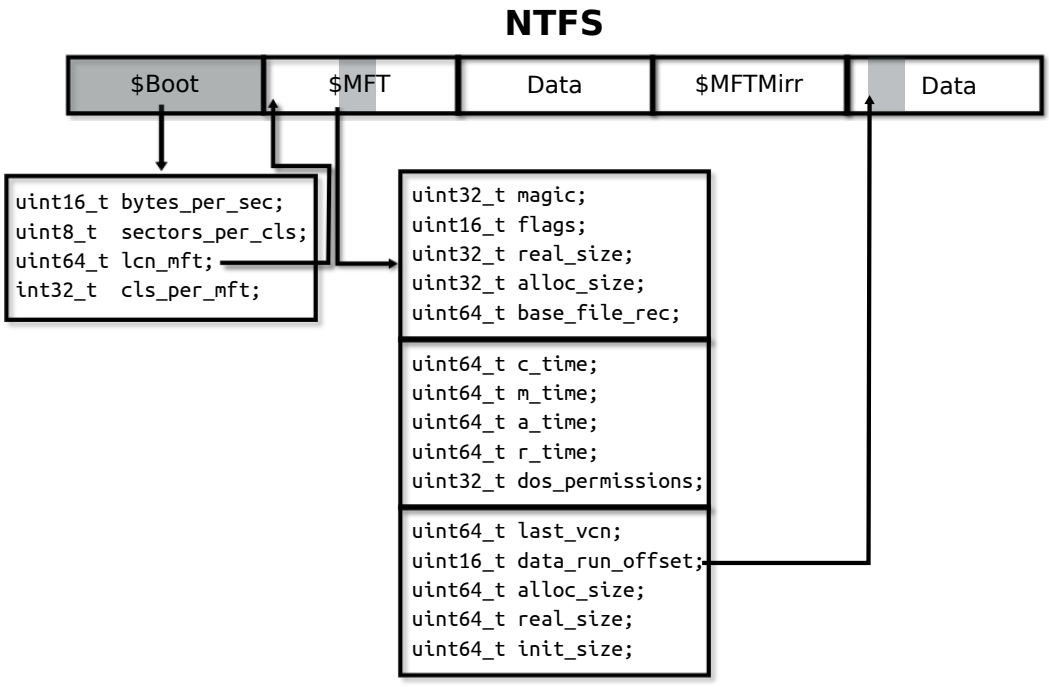


Figure 2.8: View of a NTFS partition showing the complexity of reading a single file from the Master File Table. Although all in one location, NTFS has the most complex on-disk layout out of all the considered file systems.

The first attribute contains general file information such as whether or not the referenced File Record is active, flags on the file, and the size of the file. The next structure describing a file is called the File Name and contains important information such as timestamps. The last attribute indicates a non-resident—not within the $MFT—pointer to the potions of disk making up the byte stream for the file associated with this File Record. Directories further complicate the picture, and are not shown in Figure 2.8.

NTFS was the most difficult file system to implement introspection for because its on-disk specification is not open. We worked almost entirely from reverse-engineering documentation and even with that had to guess occasionally about the data structures laid out on disk when portions of them were not described in the reverse-engineering documentation. NTFS keeps all metadata in a contiguous region of the disk called the Master File Table (MFT). This is in contrast with ext4, which splits inodes across block groups. Thus, crawling a NTFS file system is a more sequential and efficient workload, although it requires more complex knowledge. This added complexity may outweigh the costly seeks incurred by ext4 across the disk as inodes are being processed. Our NTFS implementation serves proof that implementing introspection for a complex closed-source file system without guest modifications is possible although it may be undesirable without support from

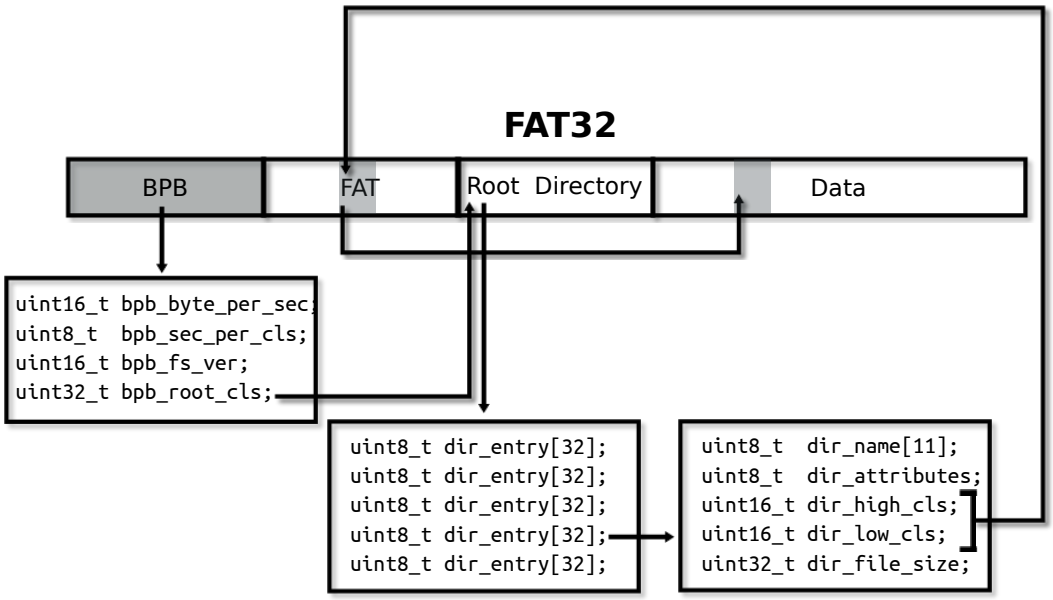


Figure 2.9: View of a FAT32 partition showing traversing the root directory and looking up the start of a file via a singly linked list in the FAT data structure. The FAT portion of the disk is an array of linked lists defining the clusters assigned to a file byte stream.

the vendor of the closed-source file system.

2.3.5 An Example Non-Journaling File System: FAT32

FAT32 is a closed-source, but open specification file system as of 2000 [74]. The availability of the on-disk format specification meant that implementing introspection for FAT32 was much easier than for NTFS, which is only known openly via reverse-engineered data structures. FAT32 is organized around a single large lookup table called the File Allocation Table (FAT). This giant table contains entries to files, which are lists of clusters on the disk. In FAT32 parlance, as in NTFS, a cluster is some number of disk sectors and equivalent to the concept of a block in ext4. The order of these lists is determined by lookups inside the FAT32 table.

Figure 2.9 shows a typical FAT32 partition layout. It starts with the BIOS Boot Parameters (BPB) section which contains bootstrapping variables and is similar to ext4's superblock or NTFS's $Boot file. Technically, there is an additional Reserved Region before the BPB, but that is not important for this discussion of FAT32-specific on-disk structures. The BPB provides details for indexing into the Root Directory. Directories are arrays of 32-byte directory entry structures. A directory entry pointing to a directory has the ATTR_DIRECTORY attribute set in the dir_attributes field. The

map of disk clusters to the byte stream making up a directory or a normal file is stored as a singly linked list within the FAT portion of the disk.

The FAT portion is a large array of 32-bit integers. Each position corresponds to a cluster on the disk. The contents of each position, if not zero, indicate a pointer back into the array for the next cluster position in a file. Thus, these singly linked lists are null terminated indicating the end of a file or directory. For example, imagine a file starting at cluster 3. To obtain the next cluster in the byte stream, we index into the FAT at position 3 and get 7. This means the next cluster in the sequence is cluster 7, and this process continues until a 0 is encountered, which represents no more clusters for any particular file. Directories are just files with a special attribute set, as mentioned, and a structured list of directory entries as their contents.

The complicating factor in tracking information for introspection with `FAT32` is that `FAT32` has no unique data structure separating a file from its path. Thus, files are uniquely defined by their containing directory, which makes their metadata tied directly to their position in the directory hierarchy. To account for this problem and assign each file a unique numeric identifier, similar to an inode number, DS-VMI crawls the FAT file system tree in a depth-first scan. We use the implicit depth-first post-order file system tree position of a file to derive its unique identifier. This value can be calculated easily when doing an initial crawl of the file system tree. Unfortunately, as the tree mutates over time, this can cause large portions of the file system tree to change their unique identifiers. To prevent this, a hash of the absolute path could be used which is invariant to post-order positioning of a file in the file system tree. However, when new directories are introduced into the file system hierarchy, potentially large numbers of hashes also require updating. Our DS-VMI prototype currently only implements the post-order file system tree identifier. It is difficult to create a better identifier because nothing else uniquely identifies a file, neither implicitly nor explicitly.

2.4 Asynchronous Queuing of VM Writes

Remember that KVM sends emulated I/O to QEMU, which is a userspace emulator. We copy this write stream from QEMU to our DS-VMI prototype over a TCP socket using a modified version of QEMU's `drive-backup` command. As Figure 2.10 shows, our modifications are located within QEMU's core, between the layers that communicate with the guest VM and those that communicate with the backing storage. This lets our DS-VMI prototype operate independently from virtual disk formats, such as KVM's `qcow2` or VMware's `VMDK`, and I/O protocols, such as IDE or the paravirtualized VirtIO. Our prototype thus supports any virtual disk format and any disk I/O protocol supported by QEMU.

We anticipate that similar implementations are possible with other hypervisors—that is, implementations divorced from the specifics of the virtual storage setup. The format of the raw writes we obtain from QEMU is very simple. It consists of a small header structure detailing the position on disk of the write, and the number of bytes in the write. The last part of this structure is a pointer to the actual bytes. We serialize this structure again using BSON. We have our own implementation of BSON in C, which we released open source. This BSON-serialized version of the write stream

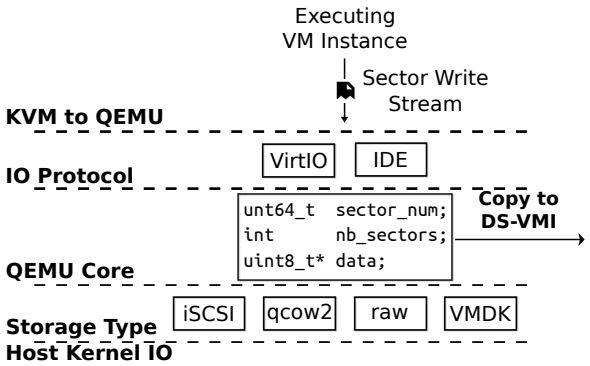


Figure 2.10: Connecting QEMU to DS-VMI.

gets converted into NBD write messages for transport to and consumption by DS-VMI. The TCP socket could route to the local host, or to another host on the network for further processing.

The other end of the NBD TCP socket is connected to the *Async Queuer*, shown much earlier in this chapter, in Figure 2.1 (b), which collects the write events and copies them uninterpreted into an in-memory queue for further processing. In our case, the Async Queuer is little more than a buffer and a translator between the NBD protocol, and our in-memory queue. Our custom NBD implementation, which is also open source, does not need to implement reads as we only need writes duplicated for DS-VMI, although we implemented reads for testing purposes. Limiting the amount of duplicate operations reduces overhead. Our implementation is built upon the `libevent` library, and we believe it to be performant. In tests, our implementation can saturate gigabit Ethernet network links. The challenge is to minimize I/O stalls on the write path of the introspected VM. In order to minimize or eliminate stalls, the Async Queuer empties the socket buffer quicker than the incoming stream of writes. To accomplish this the Async Queuer processes events as quickly as possible, and uses double-buffering during flushes to further minimize stalls. Although in the worst case it is not possible to keep up, and writes wait for sometime in the TCP socket buffer.

2.5 Introspecting Live Virtual Disk Writes

The *Inference Engine*, introduced early on in this chapter in Figure 2.1 (c), first retrieves the BSON-serialized metadata associated with the virtual disk being monitored, decompresses it, parses it, and stores it in a Metadata Store either all at once or lazily (lazy loading optimization described in Section 2.8.6). Remember that this metadata was obtained via crawling the disk as described in Section 2.3. The Metadata Store queues sector writes awaiting translation, and also stores metadata in translation tables for fast lookup. We use Redis [101], an efficient in-memory key-value store, as our Metadata Store. Redis also doubles as an implementation of a publish-subscribe message broker, which we use to implement `cloud-inotify` as described in Chapter 3.

Once virtual disk metadata is loaded and the Async Queuer starts copying raw write events into the Metadata Store, our DS-VMI prototype begins processing the write events by translating the received virtual disk sector writes into actual file system updates. To achieve this, each VM instance has an associated DS-VMI process on its host started alongside the VM. Or, if necessary, DS-VMI may run over the network instead of at the same host as the executing VM. Since DS-VMI runs as a set of separate Linux processes, it can benefit from multiple cores on the host.

At runtime, disk addresses are reverse-mapped using the lookup tables in the Metadata Store in order to determine which file or directory is modified by a disk sector write. Creations and deletions of files and directories are detected via inference based on metadata manipulations; this is file-system-specific and may require monitoring of a journal, inodes, or other file system data structures. Our DS-VMI prototype stores and maintains metadata in a file-system agnostic format, implemented via multiple Redis keyspaces. The benefit of a file-system agnostic format is that it enables easy implementation of generalized tools which work independent of the monitored file system type.

Given a write to an arbitrary position on disk, DS-VMI begins by first identifying if the write is data or metadata. This is done based on mappings maintained in the Metadata Store. If a write is data, DS-VMI only needs to determine which file and which bytes within that file were modified. To reverse-map a write operation to a data block of a file, DS-VMI queries the `sector` keyspace. To retrieve the file pathname, DS-VMI queries the `path` keyspace which maintains an index from file metadata to full path. If any process registers interest in a path, both metadata updates, and full data writes are passed on. In the case of metadata, the write is additionally deeply inspected by DS-VMI and appropriate Metadata Store data structures are updated to maintain correct mappings. For example, the metadata might indicate creation of a new directory, or truncation of a file.

Naïvely, disk mappings could be maintained at the level of disk sectors, the smallest unit addressable on disk. However, it is much more efficient to match the granularity of file system blocks, because file system block sizes are typically 8-16 times larger (4-8 kibibytes) than disk sectors (512 bytes). Thus, our prototype implementation of DS-VMI maintains mappings at the granularity of file system blocks. In `ext4` the block size is derived from the superblock. For `NTFS`, cluster size comes from the "Boot File"—`$Boot`. And for `FAT32`, the cluster size is derived from the BIOS Parameter Block (BPB) data structure.

2.6 Live Attachment and Detachment

Up until this section, we considered DS-VMI as an always on type of monitoring mechanism. But what about already running VMs? What if the overhead of DS-VMI proves too costly for a new workload? These types of questions led to developing the capability of DS-VMI to live attach to a running VM instance, and to also detach on command. Adding these two capabilities also makes DS-VMI fit into a cloud ecosystem easier, as we will see in Section 2.7, because it does not actually require changing the check-in, check-out process for virtual disk images if it can live attach.

We describe our approach to live attachment via live crawling in Section 2.6.1. We describe our approach to detaching in Section 2.6.2.

2.6.1 Live Crawling and Attaching

As described in Section 2.3.1, crawling a practical virtual disk image takes generally less than 60 seconds. Because this time is so short, when we attach to an executing VM we just live crawl its attached virtual disks. The key difficulty with this approach is that the file system is mutating at the same time. Hence, it is possible for our crawl to produce inconsistent metadata. For example, consider a virtual machine in the middle of deleting a file as we crawl through the file's metadata. Initially we begin with a valid file. As we crawl imagine the file system eclipsing us and mangling an important data structure we are about to read. As we continue parsing what we believed was a valid file, we eventually begin parsing invalid metadata. This can happen even with crash consistent file systems. However, as we describe below, by buffering outstanding writes during our crawl, we can catch up and fix our inconsistencies.

Once the online crawl completes, we must fix up the metadata to represent the actual consistent live state of the crawled file systems. In order to do this, we need to know every change which occurred while we crawled. By understanding the changes which occurred during the crawl, we create a consistent view caught up to the current state of the virtual disk and its file systems. When the live crawl starts we begin buffering writes that occur during the crawl. This buffer only grows until the crawl finishes. Then, we run all of the writes through the normal introspection logic for that respective file system. We let introspection catch up normally with the in-memory metadata, which results in a consistent state.

If we are not careful, this catching up phase leads to incorrect file-level events being generated in the file-level update stream. For example, imagine the creation of a new file and how it affects directory entries. Further imagine that the crawl has already recorded the existence of this new file, but the write buffer contains writes to this directory both before and after the existence of the new file. While DS-VMI catches up to these writes in the buffer it makes erroneous inferences. When it introspects a write to the directory not containing the new file, DS-VMI believes the file has been removed from the directory and emits that update. Later, when DS-VMI sees a write to the directory containing the file it believes the file has been added to that directory again. Unfortunately, both inferred file-level updates are wrong.

To counteract these inconsistent file-level update streams while live attaching, we disable emitting of the file-level update stream until we ensure a consistent view by crawling and introspecting each write in our buffer. Once fully caught up, we re-enable emitting of file-level updates. This ensures that we never emit a false update while live attaching.

Alternatively, we could crawl a snapshot of a storage device which provides safer semantics during the crawl and obviates the need to disable emitting the file-level update stream while introspecting the buffered writes. We must still buffer writes for introspection post-crawl, but will no longer need to worry about erroneous inference of file-level updates. Scanning a snapshot of a

virtual disk drive is equivalent to taking a very long time to process a single write to the device. Thus, just as with normal DS-VMI, we can guarantee a consistent file-level update stream from the moment we finish the crawl and begin introspecting buffered writes.

2.6.2 Detaching Introspection

Turning off DS-VMI is a much simpler process than attaching it to a VM. In theory, DS-VMI stops immediately with the cessation of duplicating writes. Detaching is basically effortless. In our prototype implementation, detachment occurs in three simple steps. We anticipate similar steps for most other hypervisors, as there is no reason why more complication would be required. The steps to detach are:

1. Issue a cancel of the `drive-backup` block device job to QEMU
2. Stop the Async Queuer process
3. Stop the Inference Engine process

Canceling the `drive-backup` command causes an immediate end to the primary source of overhead to a monitored system: duplication of writes. Once the job is canceled, no more NBD writes come over the TCP socket. Thus, we can safely terminate the Async Queuer process without causing any exceptions within the hypervisor. Finally we teardown the Inference Engine, letting it finish introspecting the last few outstanding writes. This whole process generally takes less than 10 seconds, thus detaching is a much more rapid operation than attaching.

2.7 Integration with Existing Clouds

Although designed for integration into cloud infrastructure, none of this chapter explicitly dealt with how to structure such an integration. In this section, we deal with this problem directly by describing an API extension to OpenStack, and providing an open source reference implementation [127, 128]. The implementation is more generic than just disk-based introspection techniques, and designed to also support other forms of introspection such as memory introspection.

There are two paths towards implementing DS-VMI within cloud infrastructure. The first path we assumed for most of this chapter: crawl virtual disks on check-in, upon execution of a VM from a virtual disk load the crawled metadata and activate DS-VMI. This path brings every single executing VM and its associated virtual disks under the purview of DS-VMI. It would work well for a fully managed cloud, such as an enterprise cloud, but this path requires invasive changes to the storage of virtual disks, and the execution of instances from those disks. The second path we just described in Section 2.6. This path decouples DS-VMI's implementation from the life cycle of a VM instance, letting DS-VMI activate on-demand for cloud users and operators. This second path is the most promising path for acceptance into a cloud infrastructure, because it only requires an

extension API, not invasive changes to existing APIs. For the remainder of this section, we explore taking this second path over the first one.

2.7.1 Designing an API within OpenStack

We have taken the DS-VMI prototype implementation as described in this chapter and outfitted a research cloud with DS-VMI to better understand how DS-VMI fits into real cloud work flows. We used a research OpenStack cluster running on Ubuntu 14.04 LTS server hosts using the KVM hypervisor with QEMU emulating hardware. This is a common, and in fact the default, setup for OpenStack deployments. We deploy a customized QEMU, patched to add the special `drive-backup` command to duplicate writes to our DS-VMI prototype. This modified hypervisor has been in production use across hundreds of VM launches for research purposes and course projects, and also used to host VMs with no downtime over the course of a full year.

Although the hypervisor needs modification, we required no modification of the supported guest: Ubuntu 14.04 LTS Server. The default cloud image provided by Canonical works out of the box with DS-VMI. Thus, DS-VMI delivers on its promise in a real cloud on unmodified production VM images. With zero modifications we are able to capture writes and stream them as an interpreted file-level update stream for any executing Ubuntu guest across our research cloud. In addition, to illustrate the near-real-time nature of our file-update stream, we built a WebSockets-based front-end which has the capability of showing updates in near-real-time. The updates are streamed as BSON-serialized messages from a back-end compute node, converted into JSON-serialized messages via a BSON-to-JSON proxy, and transported to a client web browser over a WebSocket. We provide a minimal JavaScript library for subscribing to updates via `cloud-inotify`.

What is the API we extended OpenStack with implementing DS-VMI? As with most clouds, our API extension is a set of REST endpoints. We have deliberately left the DS-VMI OpenStack API generic to allow for more forms of introspection other than disk-centric introspection. We integrated our implementation into OpenStack as an extension API for the `nova` compute project. The core representation of state in our implementation is the `IntrospectionEntity`. An `IntrospectionEntity` is an abstract representation of an arbitrary underlying introspection primitive. Primarily we envision both memory- and disk-based introspection being unified under this single API; however, any form of introspection falls under the `IntrospectionEntity` umbrella. Also, note that this API makes no assumptions about when an introspection begins, or when it ends. By default we live attach to executing VM images and crawl their disks in real-time. This lets our framework only affect the VMs that our users want included in DS-VMI.

As Table 2.3 shows, the API extensions follow a simple pattern provided by OpenStack. We have deliberately left the API very generic and not directly tied it to the virtual disks serving an instance. Although that is the only form of introspection supported by our implementation, we wanted an API which could subsume future introspection efforts. For example, memory introspection of live, volatile state can benefit from this same API. This API is designed around the capability of live attaching to a running VM instance. The general workflow is to activate introspection for a VM as

Type	Endpoint	Returns
GET	/servers/<instance_id>/os-introspection	Lists active introspections
GET	/servers/<instance_id>/os-introspection/<introspection_id>	Retrieve details about a specific introspection
POST	/servers/<instance_id>/os-introspection	Success at starting introspection or error code
DELETE	/servers/<uuid>/os-introspection/<introspection_id>	Teardown introspection for an instance

Table 2.3: These are the REST API calls extending OpenStack for Introspection.

identified by an OpenStack UUID, ensure that it is operating correctly, and then access one of the two interfaces described in the next two chapters. The next chapter discusses `cloud-inotify`, and we directly implemented a front-end for `cloud-inotify` within OpenStack in the form of a proxy web application. Our front-end is a proxy designed to proxy `cloud-inotify` messages via WebSockets to a client's web browser or another listening system. We chose WebSockets because this model fits directly with DS-VMI's file-level update stream. We also provide a matching JavaScript example application managing subscriptions to upstream virtual instances and virtual storage devices. Our proxy currently resides on a known TCP port, and there is no API yet for starting or finding the proxy.

2.8 Evaluating Overall Overhead

This section provides an answer to the question: what is the overhead on the write path of virtualized storage with DS-VMI monitoring its stream of writes? We find worst case behavior in line with expectation—we have a write amplification of 2, and expected worst case slowdown of 2x. We find overhead on realistic workloads to be either negligible, or at least within reason.

We begin this section by describing the experimental setup, used throughout, in Section 2.8.1. We then explore overhead via four realistic workloads in Section 2.8.3 through Section 2.8.5. We finish this section with two optimizations for DS-VMI to reduce overheads in the worst case. We explore lazily loading metadata to reduce memory pressure in Section 2.8.6. We explore dropping writes as early as possible in Section 2.8.7 to reduce costs associated with copying them out even just to another process.

2.8.1 Experimental Setup

All host nodes are identically configured throughout all of the following experiments. Each machine has a 3.00GHz Intel Core 2 Duo E8400 CPU and 4 GB RAM and runs an up-to-date version of Ubuntu 12.04 LTS AMD64 Server. We base all of our work off of a recent QEMU source tree, git commit `71ea2e01`. We use Redis server version 2.2.12 and `libhiredis` version 0.10.1. For BSON, we have our own custom implementation in C. Each host has two hard drives: a main 250 GB drive (Seagate ST3250310AS) and a secondary 1.5 TB (Seagate ST31500341A). The secondary drive was used to write and store log files, and the main drive was hosting the VM virtual disks. This

setup minimizes I/O contention while collecting results from experiments. Unless otherwise stated, timing experiments were run 20 times and both their average and standard deviation are reported.

When running a VM we follow IBM's KVM best practices [52]. Both the guest VM and host OS are configured to use the 'deadline' elevator algorithm for disk I/O scheduling, the VirtIO paravirtualization solution for I/O communication, and the asynchronous I/O back end native to their host. Before running a VM, we `sync` the host and drop all file system caches. Once the guest VM is booted, we repeat the procedure inside the guest. We configure and begin executing an experiment via `ssh`. VMs are only run for a single experiment, then discarded by deleting their hard drive and replacing it with a pristine copy. When an experiment begins within a VM guest we use a simple Python script to send a single UDP packet to a host daemon process. This process records a timestamp for the UDP packet and acts as the timer for experiments within VM guests. When an experiment finishes inside a guest VM, the Python script sends a final UDP packet to the host daemon process and shuts down. By using an external clock tied to the host VM we reduce the risk of invalid timing data due to unreliable VM clocks.

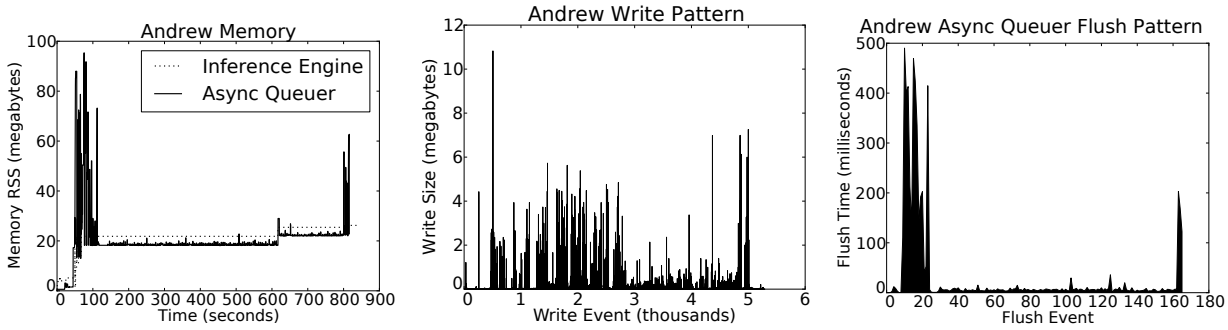
For all VM experiments we used a single VM guest pre-loaded with all software needed to perform the experiments. The guest is an Ubuntu 12.04 LTS AMD64 Server, with 1 CPU, 1 GB RAM, 20 GB disk, and a single partition containing an `ext4` file system with default file system options and 2.6 GB of used space.

2.8.2 DS-VMI Tunables

There are several tunable parameters which we fixed throughout the experiments for DS-VMI. These tunables are shown in Table 2.4. The asynchronous queuer is the first and last point of contact between an executing VM, via QEMU, and DS-VMI. If the asynchronous queuer copies write events slowly, the performance of the executing VM guest will be negatively impacted. Table 2.4 shows the main parameters which affect the behavior of the asynchronous queuer. The "Default" values were used in the experiments. In our experiments the most critical parameters are the "Async Flush Timeout" and "Async Queue Size Limit." The flush timeout helps bound the maximum latency from a disk write to an emitted inferred file-level event on a channel for subscribers. Large monitoring systems such as the one Akamai [24] deploys on $70,000+$ servers require "near real-time" updates on the order of minutes. Our default setting of 5 seconds may be too aggressive; however, this choice explores performance when flushes occur with high frequency. The queue size limit bounds the amount of memory that the asynchronous queue process may consume. The outstanding write limit bounds the number of writes queued for the inference engine—preventing it from falling too far behind the guest VM. The "Unknown Write TTL" defends against denial of service: if a guest writes a large amount of data, but never associates it with live files, the data will not be kept in a queue indefinitely waiting to be assigned a path.

Tunable	Default
Unknown Write TTL	300 seconds
Async Flush Timeout	5 seconds
Async Queue Size Limit	250 MB
Async Outstanding Write Limit	16,384
Redis Max Memory	2 GB

Table 2.4: List of DS-VMI runtime tunables which affect performance.



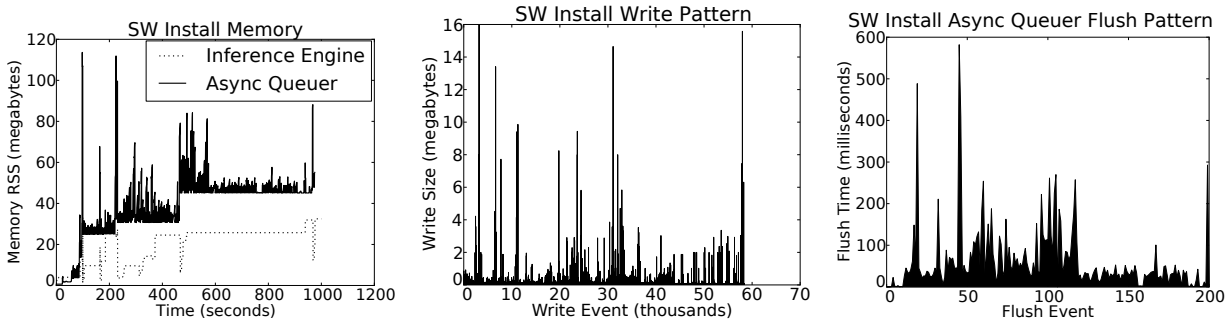
(a) Modified Andrew memory usage (b) Modified Andrew write pattern (c) Modified Andrew flush pattern

Figure 2.11: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the Modified Andrew benchmark.

2.8.3 Light-rate Small Writes: Modified Andrew Benchmark

The Andrew Benchmark is a well known benchmark [49]. It is designed to model common operations on a developer's file system. It operates by compiling a program and performing manipulations to the source tree. We modified the Andrew Benchmark to modernize it. In the `MakeDir` phase, our modified version creates a directory tree mirroring the `linux-3.5.4` kernel tree [65]. In the `Copy` phase, it copies the entire source tree, including files, into this directory tree. The `ScanDir` phase reads file system metadata for all files in the tree. The `ReadAll` phase reads the contents of all files in the tree. Finally, the `Make` phase compiles the Linux kernel using a configuration provided by `make defconfig`.

With the default parameters, the Modified Andrew Benchmark shows negligible overhead with DS-VMI introspecting disk writes. This is because the write traffic was not sufficient to fill the asynchronous queue, and when flushed due to timeout, the write volume was small enough to avoid any performance degradation. The write pattern shown in Figure 2.11 demonstrates minimal write clustering compared to PostMark and bonnie++, two benchmarks described below. The Modified Andrew Benchmark had the fewest writes out of all the benchmarks: 5,293 in total.



(a) Software Install memory usage (b) Software Install write pattern (c) Software Install flush pattern

Figure 2.12: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the SW Install benchmark.

2.8.4 Clustered Large Writes: Installing Software

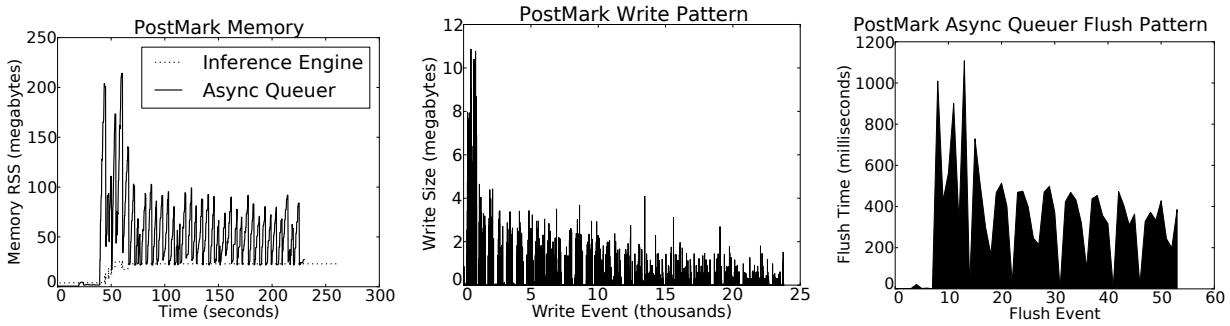
The Software Install benchmark, inspired by a benchmark used in [106], uses Ubuntu's `apt-get` tool within the guest to install a long list of server packages that have been downloaded in advance. The server packages include Apache, MySQL, PHP, Ruby on Rails, Java Application Servers, and many others.

Figure 2.12 shows the results of a run of the Software Install benchmark. The Software Install benchmark has the largest number of writes: $61,694$ in total. This benchmark, although it writes a lot of data, spreads the writes over a long period of time. There were no large bursts of heavy write activity and no wild spikes in asynchronous queuer memory. Even though it does not trigger extra asynchronous queue flushes it is, however, interrupted too frequently by timer-based flushing. Whenever the timer fires and the asynchronous queue is flushed, a few writes pending from the VM receive slightly higher latency. This effect, accumulated over the entire benchmark, was sufficient to significantly slow it down.

2.8.5 Moderate-rate Small Writes: PostMark

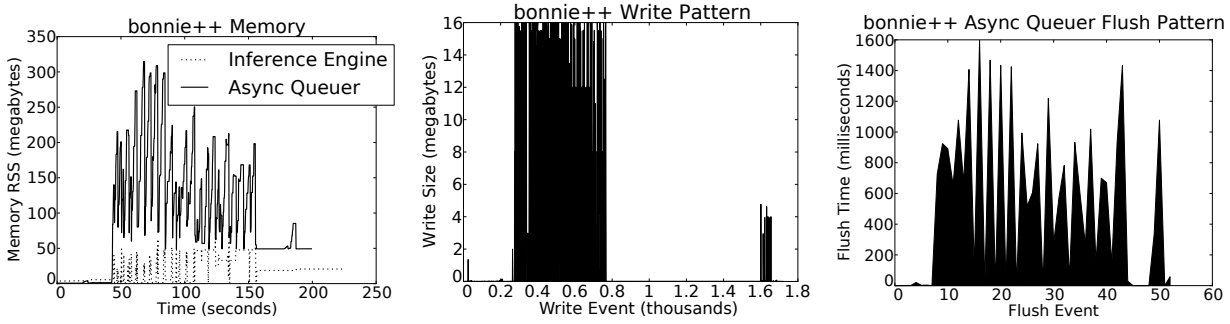
PostMark [57] is a well-known benchmark designed to simulate mail server disk I/O. We used it with a configuration suggested by [117]: file size $[512, 328072]$, read size 4096, write size 4096, number of files 5000, number of transactions $20,000$.

PostMark shows similar behavior as bonnie++; however, its write pattern is more dispersed. Figure 2.13 shows that PostMark has many smaller clusters of writes. Its asynchronous queue memory, although spiking like bonnie++, does not fill as often. These spikes do trigger extra flush events, which incur a performance penalty just as in the bonnie++ case, though on a much smaller scale. In this case the experiment ran for 231 seconds, resulting in 46 expected and 54 actual flush events.



(a) PostMark memory usage (b) PostMark write pattern (c) PostMark flush pattern

Figure 2.13: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the PostMark benchmark.



(a) bonnie++ memory usage (b) bonnie++ write pattern (c) bonnie++ flush pattern

Figure 2.14: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the bonnie++ benchmark.

High-rate Large Writes: bonnie++

bonnie++ [25] is a microbenchmark tool designed to measure the overhead of various file system operations such as create, delete, write, and read. We used its default settings. By default, it attempts to write a dataset at least twice the size of main memory.

A breakdown of memory usage, I/O pattern, and asynchronous queue flushes for bonnie++ is the first row in Figure 2.14. The first graph for bonnie++ shows a breakdown by memory of the various components necessary for DS-VMI. The inference engine, across all experiments, shows very little memory usage. The memory usage of the asynchronous queuer, however, repeatedly spikes up and down. This occurs because bonnie++ is a write-intensive microbenchmark, and the asynchronous queuer, in this case, is regularly hitting its configured queue size limit of 250MB and flushing to Redis. The effect is so pronounced that it slows down the VM guest while the flushing is occurring. In the experiment presented for bonnie++ in Figure 2.14, the VM guest ran for 200 seconds. If

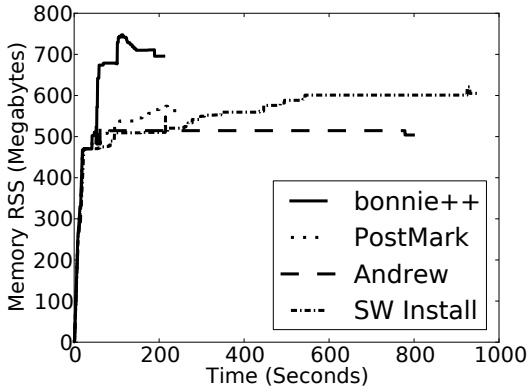


Figure 2.15: Memory usage by Redis for each experiment. Only a single run was examined.

Experiment	Asynchronous Queuer (MB)	Inferencce Engine (MB)	w/ Redis (MB)
bonnie++	250	49	1043
Andrew	88	9	630
PostMark	214	27	739
SW Install	81	26	708

Table 2.5: Peak memory usage of the Async Queuer, inference engine, and Redis combined.

flushes were only triggered by the 5-second writeback timer, this implies a maximum of 40 flushes. However, the right-most graph shows 53 flushes, 13 triggered by the 250 MB ceiling. In this and the other experiments, there were not enough write operations to trigger the outstanding-writes tunable. The middle graph confirms: this experiment was the most write-intensive of all of them—the other experiments have more dispersed write patterns. It is this closely-clustered, intense write pattern that causes the performance degradation. We simply can not hide the overhead of copying writes once buffers fill.

2.8.6 Reducing Memory Footprint: Lazily Loading Metadata

Figure 2.15 shows memory used by Redis during each of the four experiments, and Table 2.5 shows peak memory usage in Resident Set Size (RSS) by the inference engine and Async Queuer combined with Redis. At startup, the Redis database fills with metadata and the Async Queuer awaits writes from a booting VM guest. At this stable point the Async Queuer process uses 652 KB of memory, the inference engine uses 4096 KB of memory, and Redis uses 393.23 MB of memory.

This memory overhead is 15% of used disk space, when Redis is pre-populated with metadata for all files in the guest file system. This is a fairly high overhead, and becomes prohibitive with larger and larger used disk size. However, during benchmarks, as well as in expected day-to-day

Experiment	File Tree Loaded	Old Peak (MB)	Peak (MB)
bonnie++	4.7%	1043	766
Andrew	17.2%	630	357
PostMark	5.2%	739	562
SW Install	11.3%	708	533

Table 2.6: Lazy loading optimization effect on memory.

use, only a small fraction of the file system tree within the guest is modified. Thus, loading and caching metadata only for recently written files promises to greatly reduce the memory overhead of our approach.

The results of implementing lazy loading of metadata are shown in Table 2.6. With this optimization, the startup memory footprint drops to 4% of used disk space (114 MB instead of 392 MB), the loading of metadata takes 73% less time (5 seconds in the base case), and peak memory usage drops. Further optimization seems possible with customized data structures cutting out Redis, but we did not explore this path. We implement lazy loading of metadata by leaving pointers into the serialized metadata file within our in-memory datastructures. For example, we load into memory the information that a given sector is related to a certain inode. But, we don't load the inode's metadata into memory until we encounter a write to that sector.

2.8.7 Dropping Writes

Blocks in a file system can be categorized into metadata and data. A high write throughput implies that large quantities of data blocks are being written. The intuition for this optimization is that dropping data writes should significantly reduce the throughput required, while maintaining proper mappings of disk sectors to files by continuing to consume metadata writes. The only loss of information occurs when blocks transition from data to metadata such as when a data block becomes a listing of files in a directory. We either wait for follow-up writes, or read directly from the virtual disk to catch up from data blocks which may have been dropped before the transition was reflected on-disk. Figure 2.16 shows the effect of data dropping. All of the benchmarks with significant write overhead show improvement. The application-based benchmarks show that worst case overhead is reduced to 39.5% instead of the expected 100% overhead stemming from duplicating every single write. Here we trade-off completeness to improve performance. We implemented the dropping as early as possible within the hypervisor.

Our implementation of dropping writes is based off of a customizable bitmap shared between DS-VMI and the hypervisor. This lets the hypervisor drop writes early and not incur the cost of copying them to DS-VMI. In addition, it lets DS-VMI customize the writes that get passed along dynamically as metadata updates change state over time. The bitmap refers to individual disk sectors, each bit represents a sector. If the bit is a 1, the sector is passed along to DS-VMI. If the bit is a 0,

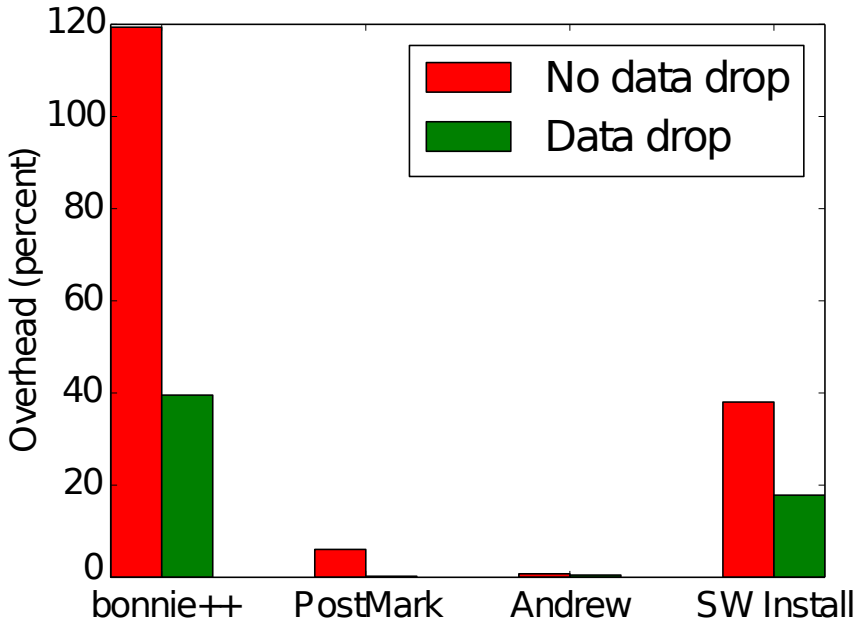


Figure 2.16: Effect of dropping data writes on DS-VMI efficiency in terms of normalized overhead.

the sector is immediately dropped by the hypervisor and not passed along for further analysis. The size of the bitmap is the overall size of a virtual disk divided by 512 bytes. For example, a 20 GiB virtual disk results in a bitmap of size 5 MiB.

2.9 Limitations of DS-VMI

This section discusses the limitations of DS-VMI as a monitoring framework. As previously described, information which resides only in the memory of an executing virtual machine guest and never flushes to persistent storage remains outside the purview of DS-VMI and its interfaces. In addition, modern protection schemes including full-disk encryption (FDE) make file-level information indecipherable to DS-VMI. We describe monitoring limitations in Section 2.9.1, and discuss the implications of modern protection schemes in Section 2.9.2.

2.9.1 Monitoring Limits

DS-VMI does not work well for all types of files and monitoring workloads. Short-lived, transient files typically exist only within the page cache of an executing virtual machine guest. Such files never flush to persistent storage. Thus, they are invisible to DS-VMI. Raw writes directly to persistent storage without using the framing of a file system are not useful to DS-VMI. Although they can be monitored, they do not fit within the structure of the file-centric interfaces layered on top of

DS-VMI. For example, a database writing directly to disk appears as a stream of unstructured writes. Sophisticated attackers hide persistent data within unused file system blocks, or unused portions of a disk. These writes do not affect any real file visible to the guest. Although they are detectable via DS-VMI, they are semantically meaningless without association to higher-level entities.

Reads are not introspected at all by DS-VMI, and certain types of writes are impossible to introspect. Cloud workloads requiring high performance storage typically receive direct access to storage devices. Their writes bypass the underlying hypervisor which makes introspecting them impossible without hardware support. Monitoring applications based on the reads performed by a guest can not use DS-VMI. For example, determining which files are loaded at boot time is not possible with DS-VMI.

The extensibility of DS-VMI mitigates some of these limitations, but some are fundamentally not solvable. Databases writing direct to storage still have an internal structure which provides semantic meaning. Extending DS-VMI to support database on-disk layouts is achievable, although it would eschew all of the architecture built upon the abstraction of a file. By using hardware-assisted introspection, DS-VMI could also introspect writes from a guest which bypass the hypervisor and go direct to persistent storage. Malicious writes to non-files must still contain internal structure usable by future DS-VMI extensions. Introspecting reads, although incurring a performance penalty, is trivial within a hypervisor. Reads and writes serviced by an in-memory page cache are never visible to underlying storage layers. Thus, memory-only structures are fundamentally invisible to DS-VMI.

2.9.2 Technologies Defeating DS-VMI

Security best practices prescribe heavy application of cryptography especially for data at rest [130]. This is a troubling trend for DS-VMI, because if the guest OS uses FDE to encrypt blocks before flushing them to persistent storage, then file system data structures become hidden. File-level encryption exposes file system data structures, but prevents deep monitoring of information within files. Such protection schemes which place no trust in the underlying cloud infrastructure render cloud-implemented services useless. However, this problem is not unique to DS-VMI.

Imagine clouds with inter-VM network bandwidth optimization achieved via packet-level deduplication. Such a technique works by recognizing data within packets that is identical. This can greatly magnify the bandwidth available between VMs in a cloud if their transmitted data has a high degree of duplication. For example, a VM broadcasting configuration state to a set of other VMs typically sends the same message many times to different hosts. By deduplicating this message across the cloud, extra bandwidth becomes available to all VMs. Should the VMs not trust their underlying cloud infrastructure and encrypt every packet, then such optimizations are impossible for the cloud to implement. This places the onus of implementing distributed optimizations firmly on the cloud user, and precludes the possibility of cloud-wide, cross-user optimizations.

The obstacle posed by protection schemes is not fundamental. It is intimately tied to the trust model applied to cloud computing. The choice to defeat such cloud-wide services and optimizations hinges on this trust model, and the tension between guest and host capabilities. If guests trust their

cloud to also protect their data, then implementing FDE and other protection schemes within guest environments is purely redundant. This dissertation argues that the best layer for implementing protection and optimization lies not within the VM guests, but beneath them within the cloud infrastructure. Whether for monitoring, or for optimization, such services benefit from global knowledge, coordination, and economies of scale when implemented in the cloud infrastructure. As mentioned before, decoupling these services from guest-level misconfigurations and compromises is attractive. This is not possible without cooperating with the cloud by exposing raw state.