Virtual Machine File System

Satyam B. Vaghani
VMware, Inc.
svaghani@vmware.com

ABSTRACT

The Virtual Machine File System (VMFS) is a scalable and high performance symmetric clustered file system for hosting virtual machines (VMs) on shared block storage. It implements a clustered locking protocol exclusively using storage links, and does not require network-based inter-node communication between hosts participating in a VMFS cluster. VMFS layout and IO algorithms are optimized towards providing raw device speed IO throughput to VMs. An adaptive IO mechanism masks errors on the physical fabric using contextual information from the fabric. The VMFS lock service forms the basis of VMware's clustered applications such as vMotion, Storage vMotion, Distributed Resource Scheduling, High Availability, and Fault Tolerance. Virtual machine metadata is serialized to files and VMFS provides a POSIX interface for cluster-safe virtual machine management operations. It also contains a pipelined data mover for bulk data initialization and movement. In recent years, VMFS has inspired changes to diskarray firmware and the SCSI protocol. These changes enable the file system to implement a hardware accelerated data mover and lock manager, among other things. In this paper, we present the VMFS architecture and its evolution over the years.

Categories and Subject Descriptors

D.4.3 – *Distributed File Systems.*

General Terms

Design, reliability, performance, measurement.

Keywords

Clustered file system, virtual machine, storage virtualization, scalability, storage hardware acceleration, SAN.

1. INTRODUCTION

VMware ESX Server [1] is VMware's flagship hypervisor for server class machines. Figure 1 shows 2 physical machines (servers) running the ESX hypervisor. The labels in regular font are hardware components and the ones in italics are software concepts. Physical machines access file system volumes, also known as datastores, hosted on a shared storage system and accessed via a fibre channel, iSCSI or NFS based storage area network (SAN). VMFS is the underlying file system volume for all block based storage systems that ESX hosts access. For the purpose of this paper, we define an ESX server cluster, or simply a cluster, as a group of ESX servers sharing one or more VMFS volumes via a SAN. This cluster is managed by a central entity known as vCenter (not shown). vCenter [17] may invoke operations that are contained in a single ESX server (such as powering on a VM) or operations that span multiple ESX servers (such as vMotion [4]) or multiple storage devices (such as Storage vMotion [14]).

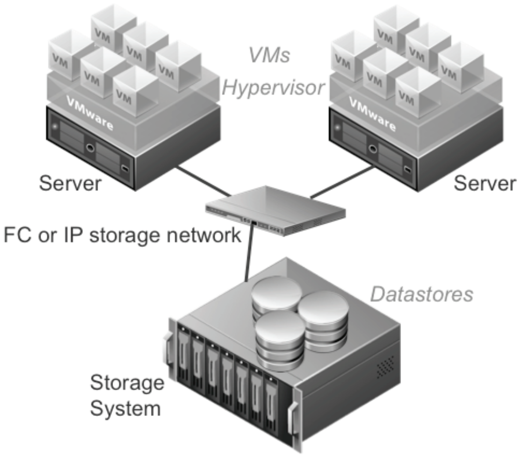


Figure 1: Physical server and storage system setup for VMware ESX Server

Figure 2 illustrates storage virtualization components inside ESX Server. The virtual machine monitor emulates a SCSI host bus adapter, and the ESX Server kernel, also known as VMkernel, populates this virtual adapter's SCSI bus with virtual direct attach SCSI disks. The guest operating system will issue SCSI commands to these virtual SCSI disks, which enter the VMkernel at the SCSI virtualization layer. This layer exports blobs upwards as virtual SCSI disks. These blobs could be a monolithic file or a set of files on VMFS datastore(s), or raw SCSI devices from the storage system of figure 1. On the way in from the VM, the SCSI virtualization layer coverts SCSI commands [15][16] sent by the guest OS into file system operations. For example, a SCSI READ CDB will be converted to an asynchronous file read request, a GET CAPACITY CDB will be converted to a file stat request, and so on. If the file system is accessed via NFS, the file system layer will forward the request to the storage system. If the file system is VMFS on a block device, the VMFS driver will remap virtual SCSI device offsets (file offsets) into volume offsets and forward the request to a volume manager. The volume manager aggregates one or more physical storage devices into a VMFS volume, and remaps VMFS volume offsets to physical device offsets. In this paper, we treat VMFS and its volume manager as a single unit. Finally, the request is forwarded to the device access layer (also known in VMware literature as Pluggable Storage Architecture or PSA [9]). This layer generally handles command queuing, task management, disk scheduling, and multi-path access, policies and failover to storage systems. This is a high throughput and CPU efficient IO path, because the average IO request just involves some offset remapping. In certain cases, VMFS, volume manager or PSA may choose to split a single IO request into smaller fragments because of discontiguity of file system blocks, volume extents or devices; or due to physical host bus adapter and device driver limits.

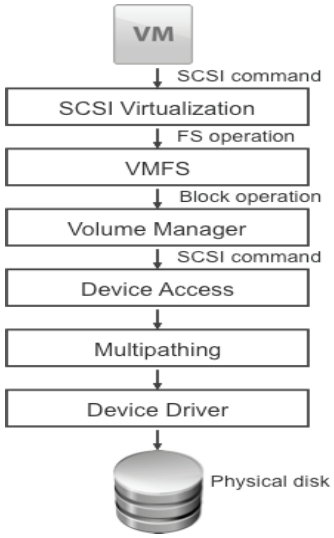


Figure 2: VM IO path through the VMKernel storage virtualization stack

This IO path has been capable of driving wire rate IO throughput at wire level latency range, and this is one of the original design goals of VMFS – to provide file system class services and dynamism without sacrificing raw device level IO performance.

The core virtual machine IO path described above can be extended via file system or block filter drivers. For example, the change block tracker is a filter driver that tracks all writes to a virtual disk since a known epoch. This is useful for creating incremental backups of a virtual disk or for efficiently selecting virtual disk data to move during Storage vMotion. Another example is the deltadisk driver, which creates snapshots [12] of virtual disk files in a file system agnostic manner.

The ESX storage virtualization stack was born out of our realization that the number of virtual machines and ease of their storage operations (such as provisioning, snapshotting, etc) will overwhelm the most capable storage systems. For example, a cluster of ESX Server 4.1 machines may store up to 10000 VMs. Each VM could use up to 60 virtual disks, and each virtual disk may have up to 32 snapshots [2]. Even at a much scaled down estimation of 2 virtual disks and 2 snapshots per VM, this amounts to 40000 distinct disks for the storage system to support if VMs were directly connected to physical disks. Now that virtualization is popular, these scenarios are well understood and storage system architecture is evolving to support such numbers. However, storage device and topology management at this scale remains an acute challenge. Some example management issues include configuring connectivity, zones and multi-path access to such a large number of disks, negotiating ownership, naming/inventory management, etc.

In summary, the hypervisor's storage virtualization stack creates a clustered virtual storage system that hosts virtual disks and disk transformations (such as snapshots, clones, etc) on behalf of VMs. It does so in a more manageable, dynamic and usable manner than a physical storage system otherwise would. VMFS is a core component of this virtualization stack and it makes it possible to host virtual machines on block storage.

This paper presents VMFS architecture and its applicability to a virtual machine workload and a SAN-based block storage system. Section 2 describes the motivation and design goals of this file system. We describe core file system features and algorithms in section 3. Section 4 describes the influence of SAN storage systems on VMFS and vice versa via examples. We conclude in section 5.

2. VMFS DESIGN GOALS

A virtual machine uses storage to store its configuration metadata and guest operating system data. This typically results in a small number of files, approximately 30 to 100 per virtual machine. These files are either very small (few hundred bytes to a few KB) text files or very large (hundreds of MB to many GB) virtual disk files and swap files. Hence, one of the VMFS design goals was to handle a relatively small number of very large files. In particular, the file system metadata overhead per gigabyte of file data should be very low so that metadata IO and latency overhead is very low in the performance critical VM IO path. In fact, IO throughput and latency as observed by a guest operating system to its virtual disk on a VMFS volume should be as good as if the virtual disk were a raw SCSI device directly attached to the guest. At the same time, this metadata design should not make it impractical to host small metadata files.

The file system assumes SAN storage systems as its underlying substrate. These systems have a few important consequences on the storage they present:

- The SCSI device that is presented to ESX servers (initiators) is a virtual entity and different logical block addresses (LBAs) of this SCSI device map to physical spindles in the storage system in unknown ways. In particular, the file system does not assume that contiguous extents on the SCSI device map to contiguous extents on a physical spindle.
- Due to the large non-volatile cache sizes and sophisticated volume managers in the storage system, the effects of sequential and random IO on a disk presented by this storage system are different from the effects of the same workload on a direct-attached disk.
- The presence of various data services change the space and performance effect of a read or write operation to this device. For example, if the disk is thin provisioned or snapshotted, the initiator benefits by not writing to previously unwritten parts of the file system.
- A given LBA could be hosted on a different physical storage tier (e.g., FC, SATA and SSD) at different points in time. The file system may affect this choice to some extent.

VMFS further assumes that all storage exported by these storage systems is shared among all ESX servers. To make this sharing possible, it needs to provide a clustered lock manager for moderating access to file system metadata and to file data. The clustered lock manager becomes the basis of various VM solutions that essentially require an atomic switchover of virtual machine execution from one ESX server to another, without having to move its storage.

One of the drawbacks of SANs is that the server (initiator) will typically observe a much bigger set of transient and non-transient error conditions on the fabric. These range from information events due to on-going fabric reconfiguration, transient errors such as temporary congestion in one or more elements, and non-transient errors such as interconnect failures, etc. ESX is intended to run a wide variety of guest operating systems, including legacy versions that are not capable of dealing with SANs. VMFS and the storage virtualization stack are therefore designed to help the guest OS deterministically react to various SAN events and error conditions. It is also in-line with one of our core virtualization principles: a guest OS should not bear the burden of managing physical devices; instead it should focus on providing an execution environment for applications.

We were aware of research on spatially optimizing file system layouts for rotating media [18] [19], but these storage systems and the special nature of our workload inspired us to design to a newer set of goals – for one VMFS metadata is optimized to result in efficient fetches and stores to/from the storage system cache from/to the back-end physical spindles. The resource manager prefers to establish a localized storage working set among many VMs, as opposed to spatially localizing parts of a single VMDK. The resource manager is also biased towards segregating the storage footprint of different physical machines to different parts of the address space to reduce cross host contention. Finally, the file system layout and algorithms are designed for co-existing with hardware functions like thin provisioning, replication, snapshotting and storage tiering. The subsequent sections describe these in detail.

That said, storage systems were initially designed with physical machines in mind. VMFS is a bridge between how a storage system works at any given point in time, and how we envision it ought to work for the sake of virtual machines. It follows that some VMFS functions were designed to make our customers change their assumptions about storage while making the transition from physical to virtual machines. For example, one of the best storage systems for enterprises that we know of, provisioned new storage at the rate of 16 GB per minute for new disk creation in the year 2007. VMFS on the aforementioned storage system provisioned at 1200GB per minute. This level of dynamism was necessary because we envisioned virtual machine provisioning to be much more frequent and convenient than what precedence dictated. This is not to say that storage systems are less capable, but they are often bound by the use cases they were originally designed for. Virtualization changes this. We are also happy to report that the storage system in the above example provisions storage at almost the same rate as VMFS at the time of writing this paper.

VMFS has also been an interesting exercise in software engineering. The file system hosts guest operating systems which, in turn, use very stable and well performing file systems. Hence, the stability and correctness of VMFS algorithms should be at par or higher than the combined stability of the various file systems it hosts inside virtual disks. Corner case algorithms of a file system, such as crash recovery procedures, split brain and IO fencing procedures generally prove to be the Achilles heel [20]. One of our goals was to make corner case code paths the same as regular code paths.

Finally, we wanted VMFS to address the usability challenge with clustered file systems. Typically, one needs to configure a low latency network, IP networking, routing, primary and secondary metadata servers, etc for a clustered file system. We started work on the first version of clustered VMFS at a time when we had only a small number of ESX customers. It was unthinkable to burden these early adopters who were trying to fathom virtualization with yet another challenge – managing a clustered file system. We wanted to make it such that the only thing one needs to use VMFS is to plug in the SAN cable.

3. VMFS ARCHITECTURE

VMFS models a volume as an aggregation of resources and on-disk locks. A resource could be a file descriptor (also known as an inode), a file block, a sub-block or a pointer block (also known as an indirection block). Each lock moderates access to a subset of resources and hosts participating in a VMFS cluster negotiate access to resources by acquiring the relevant lock. The file system driver contains a clustered lock manager, a resource manager, a journaling module to execute redo transactions, a data mover for bulk data movement, a VM IO manager, and a frontend to service VM management operations via POSIX system calls.

3.1 On-disk lock

A disk lock is a single sector (disk block) data structure. This size makes it easier to design against write tearing while trying to atomically update a lock data structure on disk. It is also the case that the total size of all the fields comprising a lock is much smaller than a sector. Each VMFS volume contains tens of thousands of on disk locks. They appear in clusters and are co-located with the specific resources they govern. The disk lock data structure contains four main fields of interest:

- HostID: This is a 128-bit unique identifier that identifies the ESX host that owns the lock at a given point in time. A value of 0 indicates that the lock is not owned by any host[1].
- Mode: A set of non-zero values to indicate whether a lock is free, held exclusively, held by multiple hosts for shared read access, or held by multiple hosts for shared read and write access.
- Generation: A monotonically increasing counter, updates every time a lock is acquired, released or broken. While the hostID field sufficiently disambiguates operations on a lock from different hosts, this field disambiguates multiple operations on a lock by the same host.
- HBregion: For each valid hostID (if any) currently using the lock, a pointer to the on disk heartbeat region of the host.
- HBgen: A generation number to validate the HBregion reference as being current or stale. It disambiguates locks held by a given host before and after a host crash and before and after a storage outage.

We will revisit these fields in the context of locking, heartbeating, and transactions later. For now, we will simply state that the VMFS lock manager is capable of acquiring a lock on disk by atomically updating the hostID field with the host's unique ID,

[1] We have intentionally not stated that a value of 0 indicates that a lock is free. We rely on a special non-zero value in the lock 'mode' field to make that determination. This gives VMFS a better chance to detect disk corruption, as opposed to interpreting corrupt regions on disk as 'free' locks and causing further damage. As a corollary, our experience indicates that disk corruption (due to whatever reason, hardware or software) usually puts more zeroes on disk than anything else.

and setting the other fields appropriately. This acquisition is valid till the host releases the lock, crashes, or fences itself from the cluster.

3.2 Host liveness via on-disk heartbeat

Every host accessing a VMFS volume acquires a heartbeat on disk to declare liveness to other hosts. A heartbeat is a single sector data structure. All heartbeats are allocated from a 1MB reserved region on a VMFS volume, thus making it possible for up to 2048 hosts to concurrently access a VMFS volume. The size of the heartbeat region is not a significant factor in determining the practical number of hosts in a VMFS cluster. This number largely depends on the IOPS capability of the underlying disk and the number and complexity of concurrent metadata transactions to the volume.

The on-disk heartbeat data structure contains the heartbeat state ID and host ID. A generation number distinguishes multiple acquisitions of the same heartbeat slot by a given host at different times. Hosts decide on a heartbeat slot based on the hash value of their respective hostIDs. In case a slot is already in use, the host will check neighboring slots until a free one is found. It follows that a locks reference to a given HBregion and HBgen is valid if the heartbeat slot referenced by HBregion contains a heatbeat in active state by the same hostID and at the same generation number as HBgen. Finally, the heartbeat contains a stamp, which the owner host is responsible to write to with a randomly changing value or a monotonically increasing counter every 3 seconds.

Hosts interested in locks moderated by a given heartbeat are free to break the locks (i.e. reset their hostID and mode) if said heartbeat's stamp does not change during any continuous 20-second observation interval. When taking ownership of a stale lock, a host is required to replay the journal referred to by the stale heartbeat to make sure that any transactions in flight that updated metadata owned by the lock are rolled forward and metadata is restored to a consistent state.

Conversely, if a given host fails to update its heartbeat slot with a new stamp in five heartbeat periods, each period with 8 IO tries each (approximately 15 seconds and 40 heartbeat IO tries), it will fence itself from accessing the VMFS volume. As part of the fencing procedure, the host will use task management commands to abort all IOs in flight. In case of IO errors that do not indicate a permanent device loss, a fenced VMFS lock manager attempts to rejoin the cluster by re-acquiring its original heartbeat slot. This is possible, if after the storage outage, the host can confirm that its previously owned heartbeat slot is in the same state that it was before the storage outage. This indicates that no host attempted to steal the locks it owned prior to the outage (else the journal would have been replayed and heartbeat slot cleared). At this point all applications on the fenced hosts can continue their operations to files opened before the outage. In case the storage outage results in a remote host replaying the fenced host's heartbeat, the fenced host acquires a new heartbeat slot with a newer heartbeat generation number on rejoining the cluster. However, all files that were opened under the previous heartbeat generation, i.e. prior to the outage, are now declared as stale file handles and IO to these stale handles is disallowed. The stale handles may be closed and reopened for resuming IO.

In summary, each host can join a cluster and hold thousands of on-disk locks contingent on successful updates to an on-disk heartbeat. On-disk locks can be acquired, validated, lost, revalidated, or broken based on the state of the on-disk heartbeat region. IO to files is contingent on lock ownership, which is contingent on a healthy heartbeat.

3.3 Resources and file system layout

VMFS contains four types of resources. File blocks are linearly arranged from the beginning of the volume to the end. Some file blocks are aggregated into a separate sub-block address space, some are aggregated into a file descriptor address space, and finally, some form a pointer block address space. All file system resources occur in clusters, and some metadata describes the cluster. A single lock governs this metadata. Figure 3 shows an example VMFS layout, which comprises of 4 resources per cluster, some metadata for the resource cluster and a lock governing this metadata. Resource cluster metadata mostly contains the number of total and free resources in a cluster and an allocation bitmap for resources. Some optional data structures include self-references such as cluster number for consistency checking, resource refcounts for block sharing, and some usage hints for the VMFS resource allocator. Resource metadata is 1 sector wide.

Since allocation and deallocation of all resources in a given cluster is contingent on a single lock, it should be clear that the number of resources in a cluster influences the number of locks that need to be acquired in the context of a given file system operation. If the resources were 1MB file blocks in figure 3, the allocator would need to acquire at least 25600 locks on a best case non-fragmented file system to service a 100GB preallocated virtual disk creation request. On the other hand, if the layout was changed to 40 resources per cluster, the number of lock acquisitions will decrease to 2560. It should be noted that a greater number of resources per cluster increases the chances of multiple hosts clashing on the same resource cluster at a given point in time. In summary, a smaller number of resources per cluster increases transaction complexity while a higher number of resources per cluster increases cross-host contention.

A number of resource clusters of the same resource type form a cluster group. As shown in figure 3, locks and metadata for all resource clusters in a cluster group are co-located at the beginning of the cluster group. This provides locality of reference at resource (de)allocation time to the resource allocator, which is exclusively interested in (reading and writing) resource locks and metadata. As stated earlier, most VMFS volumes reside on high-end SCSI devices with significant non-volatile caches. These devices are efficient at servicing large IO requests, as opposed to a larger number of small requests that amount to lesser total IO than the aforementioned large request. Due to the layout, the resource manager is able to efficiently read hundreds of locks and resource metadata regions in a single read request. An example real world VMFS layout for file block resources uses 200 resources per cluster and 64 clusters per group. This layout would enable a given host to read in locks and metadata for 12800MB of file block address space via a single 64KB IO request. This layout also provides locality of reference to file and directory IO requests, which are exclusively bound to resources. Cluster groups are repeated to create a file system. An existing VMFS volume grows over unused space on the disk or spans new disks by laying out new cluster groups that refer to the newly added space. We will elaborate on resource cluster and cluster group sizing principles in section 3.8.

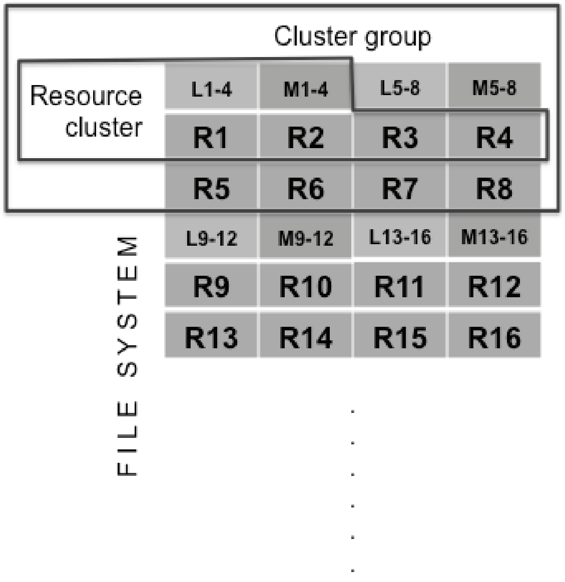


Figure 3: VMFS layout on disk

3.4 System files

VMFS creates system files, one per resource type, to contain all cluster groups for the given resource. Hence, the sequential layout of resource cluster groups for a given resource, as shown in figure 3, is contained in the system file. Since the system files use file blocks themselves, the physical layout of system file blocks, and hence the metadata of the file system, is controlled by the same allocation policies that apply to regular files in the file system. We bootstrap the file block resource system file at format time with just enough hard coded file block addresses to lay out the first cluster group of file block resources inside the file block resource system file.

As a result of system files, the methods to read and write VMFS metadata (which happens to be system file data) is the same as the methods to read and write VMFS regular file data. This gives us great testing coverage because every test that checks for file data consistency invariably produces proof towards file system metadata consistency.

The format of each system file is the same regardless of the resource it contains. Each system file contains a header to describe the layout of locks, metadata and resources inside the file. The header contains constant quantities (set at volume format time) such as resources per cluster, clusters per group, zeroeth cluster group offset in the file, resource size, and cluster group size. In addition, it contains 2 variable quantities: total number of resources in the file and total number of cluster groups. The variable header quantities are updated at format time or during a volume grow operation to use new space on a device or a new device. An example VMFS layout on the author's machine contains 4 system files: 1MB file block resources, 8KB sub-block resources, 4KB pointer block resources, and 2KB file descriptor resources. Sub-block size of 64KB and file block sizes greater than 1MB are common too. These self-describing system files allow VMFS to choose an optimal resource size and layout at format time. We will describe this dynamic resource layout process in section 3.8.

Many hosts may concurrently allocate and de-allocate resources from a given system file on a shared VMFS volume. One of the basic principles used by the VMFS resource manager is to make these hosts operate on different and distant cluster groups within a system file. This reduces the possibility of mutiple hosts contending on the same lock(s) and increases the efficiency of the clustered lock manager. We will also describe an optimistic locking system based on this layout characteristic in section 3.7.1. Figure 4 shows the physical location of files on a 250GB VMFS volume shared by 2 hosts. The X-axis is physical block offset on the VMFS volume and the Y-axis plots used blocks on the volume. The volume contains approximately 35 files and blocks used by different files are plotted in different colors. The figure shows that the 2 hosts distance themselves to reduce contention. They still strive for spatial locality of objects they individually manage (indicated by contiguous colors in each cluster).

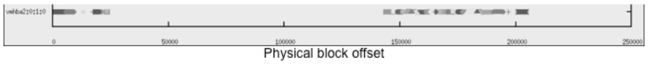


Figure 4: Physical location of files on a shared volume

3.5 Types and properties of data blocks

Virtual machines are represented as a collection of configuration and data files (and their derivatives such as snapshots) on disk. VM file sizes are bimodal – there are a few extremely small files and a few extremely large files per VM. VMFS optimizes the storage of both these file classes by using large block size for large files and sub blocks for small files. A newly created file starts its growth by allocating sub-blocks. Once the file passes a certain size criteria (equal to a small number of sub-blocks), its addressing scheme is switched to file blocks. File data from sub blocks is migrated to file blocks at the time of transition. Newer versions of VMFS optimize small files by packing the contents of files smaller than 1KB inside the file descriptor in the place typically occupied by block or pointer block addresses. These packed inodes can possibly migrate to sub-block-, file-block- or pointer-block-addressing if needed.

Data block addresses occurring inside a file descriptor contain some bits reserved for additional block attributes. For example, VMFS can pre-allocate files to a given size. This is required to make virtual disks for mission critical VMs that do not prefer to run the risk of running out of space due to future block allocation failures to sparse regions of the file. Pre-allocated blocks need to be zeroed for data isolation reasons, but zeroing virtual disks that are many GB in size is an expensive proposition. Instead, file block addresses in a preallocated file contain a 'to be zeroed' (TBZ) bit, which indicates whether the given file block was ever written to. If not, read requests to the block are zeroed by the file system driver and not forwarded to disk. A TBZ block is actually zeroed and the TBZ bit is unset on receiving a valid write to the block. In practice, large portions of preallocated VMDKs retain their TBZ bits for many years.

Resource cluster metadata for data blocks contain a 16-bit refcount, thus enabling a given data block to be shared by as many as 65535 distinct files. This forms the basis of file level snapshots in VMFS. A snapshot of a file descriptor is another file descriptor that points to the same data blocks as the original. Resource cluster metadata block refcounts record this degree of sharing. All shared block addresses in all files that contain them, are marked as such with a 'copy-on-write' (COW) bit. Writes to a COW block within a file result in new block allocation, copying the COW block contents to the newly allocated block, and patching the new block address in place of the COW block address in said file. The resource cluster metadata refcount of the COW block is decremented and the write is forwarded to the new block address.

A file descriptor has a fixed number of address slots for data blocks. Once the file size grows beyond what this fixed number of slots can address, VMFS switches the file descriptor to use pointer blocks and indirect addressing. As mentioned earlier, each pointer block is 4KB in size, and can hold up to 1024 addresses. With a default file block size of 1MB, a single pointer block can give the host access to 1GB worth of file address space. This low metadata overhead results in physical device level throughput that VMFS is typically able to achieve while servicing VM IO.

Table 1 shows results from an experiment with a uniprocessor Windows Server 2008 virtual machine running Iometer configured for 100% sequential writes at varying block sizes (4KB and 64KB). We chose a sequential write workload because it is known to produce the best results on physical disks, and hence it is the most challenging for VMFS to match. Iometer wrote to a drive inside Windows, which either mapped directly to a physical SCSI disk on the SAN, or to a virtual disk file on a VMFS volume on top of a physical SCSI disk from the same SAN. We used two types of physical interconnects – 8Gbps fibre channel (8G_FC) and software iSCSI over 10 Gbps Ethernet. Table 1 lists our results as throughput, latency and CPU cost ratios. VMFS result is the numerator and raw device result is the denominator. A ratio of 1 means that VMFS result matched exactly to the raw disk result. The reader might observe that VMFS drives slightly higher throughput at slightly lower latencies compared to raw devices. We have observed this on various devices and our customers have occasionally reported it on their setups too. This phenomenon continues to puzzle us; but we attribute this to slightly different TLB behavior and IO queuing characteristic when using VMFS. At any rate, the difference is extremely small.

Table 1: Throughput, latency and CPU cost ratios of VM IO to a VMFS virtual disk (file) and a raw SCSI disk

All ratios ➔ **VMFS/Raw**	**Throughput (MBps) ratio**	**Latency (msec) ratio**	**Cost per IO (MHz/MBps) ratio**
8G_FC-4k_seq_wr	1.013	0.985	1.030
8G_FC-64k_seq_wr	1.003	1	1.082
10G_iSCSI-4k_seq_wr	1.006	0.998	1.047
10G_iSCSI-64k_seq_wr	1.000	0.997	1.031

3.6 Acquiring a disk lock

An ESX host interested in locked access to metadata needs to atomically check the hostID and mode fields in the lock, and if the lock is free, acquire it by writing out its own hostID into the owner field and the mode in which it wants to hold the lock. The HBregion and HBgen fields, which together constitute lock liveness, point to the given hostID's heartbeat region on disk. Figure 5 illustrates a highly simplified set of steps used by the VMFS lock manager to acquire an on-disk lock. The lock manager uses SCSI reservations to establish a critical region of IO to the underlying LUN. This critical region allows the host to check the owner and liveness information of a lock, establish itself as the owner by writing to the relevant lock fields on disk if the lock is free, and release the SCSI reservation.

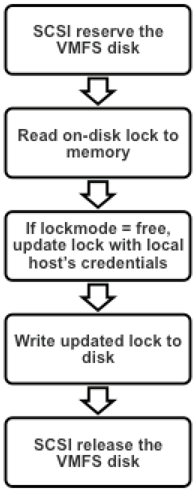

SCSI reserve the VMFS disk
Read on-disk lock to memory
If lockmode = free, update lock with local host's credentials
Write updated lock to disk
SCSI release the VMFS disk

Figure 5: Algorithm to acquire a disk lock (simplified)

Note that the file system driver might need to acquire multiple locks to successfully execute a given task. For example, consider a request to allocate a new data block to an existing file. The driver acquires a lock that governs a free block bitmap, and a lock that governs access to the file descriptor for updating the file length, pointer block list, ctime, etc. Similarly, a file create request will require a directory lock to update directory entries and a lock that governs a free file descriptor bitmap, and so on. Also note that the locking algorithm mainly serves metadata operations such as creating and deleting files, allocating and freeing blocks, and changing length or other attributes of a file. These operations are rarely invoked as part of servicing guest OS IO requests.

Unlike other clustered file systems whose lock managers rely on some form of network communication, the above algorithm relies exclusively on storage links to acquire a lock. This greatly simplifies setup steps needed to use VMFS in the datacenter. In fact, the clustered lock manager is operational as soon as an ESX host discovers a storage device containing a VMFS volume.

There are downsides of using standard SCSI commands to implement this algorithm. A SCSI reservation command locks out other hosts from doing IO to the entire disk, while the host issuing the reservation was really interested in locking out other hosts from doing IO to a particular sector on the disk. These other hosts witness IO failures with 'reservation conflicts' for the duration of the reservation. The duration of this disruption is limited to the amount of time the reservation is held, which is the time it takes for the steps of figure 5 to complete. We have observed this time to vary from approximately 10 milliseconds to 100 milliseconds for a single lock acquisition on fibre channel and iSCSI networks respectively.

Using a specific form of SCSI-3 persistent group reservation can solve this problem partially. The SCSI-3 write exclusive registrants only persistent group reservation allows remote hosts to concurrently read from but not write to a reserved disk. On an average, this is half as disruptive to IO from remote hosts. However, the latency of a persistent group reservation command is significantly higher than the latency of a regular reservation command because storage systems have multiple service processors and persistent reservations have to be synchronized across all of them. Hence this scheme adds latency to metadata operations. It was more feasible for us to optimize the regular reservation based lock manager than to use a higher latency persistent reservation based lock manager. These optimizations are generally aimed at:

- Reducing the number of SCSI reservations required to do a given transaction.
- Reducing the disruption caused by a SCSI reservation on IO from other hosts.
- Creating a new lock manager that acquires disk locks without using the algorithm from figure 5.

VMFS was always very efficient at hosting virtual machine disks and serving IO requests from guest operating systems (see Table 1). Metadata operations such as creating virtual disks, powering on and snapshotting VMs, etc were infrequent and were sufficiently served by the disk locking algorithm we outlined earlier. Over the years, newer VM solutions and usage patterns resulted in a rapid increase in metadata operations. This made it all the more important for us to optimize core cluster locking algorithms and metadata transaction routines.

3.7 Journaling and transactions

Given the layout and the locking algorithm discussed above, it must be clear that a VMFS transaction may work on several locks and several metadata regions inside the volume. For example, consider a user request to create a 100 MB file. A file system of figure 3 using 1MB file blocks will require 100 blocks to be allocated. At 4 file block resources per cluster as shown in figure 3, this amounts to acquiring at least 25 file block resource cluster locks, updating the corresponding bitmaps, and acquiring a file descriptor lock to update the descriptor with pointers to the newly allocated blocks. A VMFS transaction rolls forward the state of the file system on a replay. In the example above, the VMFS driver will acquire a file block resource cluster lock, read in the corresponding metadata region, and modify it in memory. It may then acquire more locks and modify more metadata. Once all metadata is modified, the entire transaction – a record of all locks acquired and all metadata modified – is written out to the journal.

Each host participating in a VMFS cluster maintains its own journal on the VMFS volume. The host's heartbeat region on disk stores the journal location on the volume. The disk lock acquisition step ensures that all locks and metadata occurring in a given hosts journal at a particular point in time cannot occur in any remote hosts' journal at exactly that point in time. Further, the lock generation number stored for every lock inside a transaction determines the ordering of transactions when multiple transactions in multiple journals happen to contain a reference to the same lock (obviously acquired at different times). It follows that all transactions containing older generation numbers of a given lock need not be replayed. In fact, if a stale lock is detected on disk with a crashed hostID H and generation number G, it is implied that only H's journal contains a transaction that may need to be replayed to bring the stale lock and its metadata back to consistency. Further, all transactions in H's journal that refer to this lock at a generation number lower than G may safely be ignored during replay.

Coming back to our 100 MB file example, once the transaction is committed to the journal, the driver updates the actual metadata regions (bitmap, file descriptor) on disk and releases all locks. This simplified state machine is illustrated in figure 6.

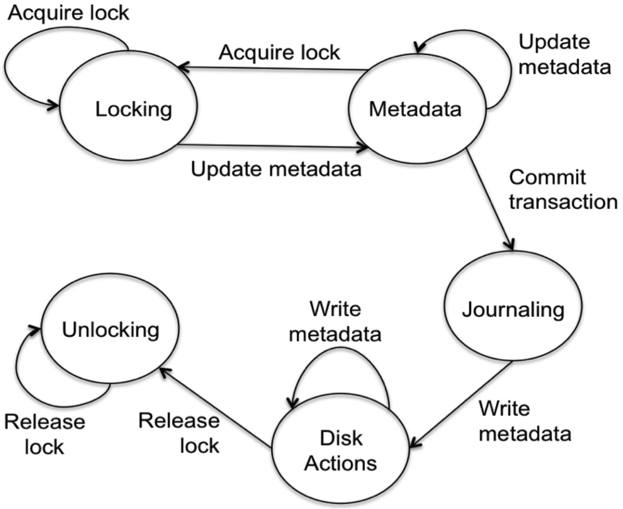


Figure 6: Transaction state machine

The host will invoke the algorithm from figure 5 to acquire locks required during a transaction. Each lock acquisition disrupts IO from other hosts in the cluster, and it follows that the total disruption is proportional to the number of locks a transaction requires. Our 100MB file creation example will cause at least 26 disruptions, and larger file creation will cause more. Most transactions on VMFS involve creating, deleting or modifying large virtual disks, snapshots, and VM swap files. Hence, it was very important to replace the above algorithm with a variant whose locking complexity (and disruption) does not increase linearly with the complexity of the user's request.

3.7.1 *Optimistic locking*

We created optimistic locking that exploits the layout characteristic of VMFS resource manager to speculatively execute through a transaction state machine.

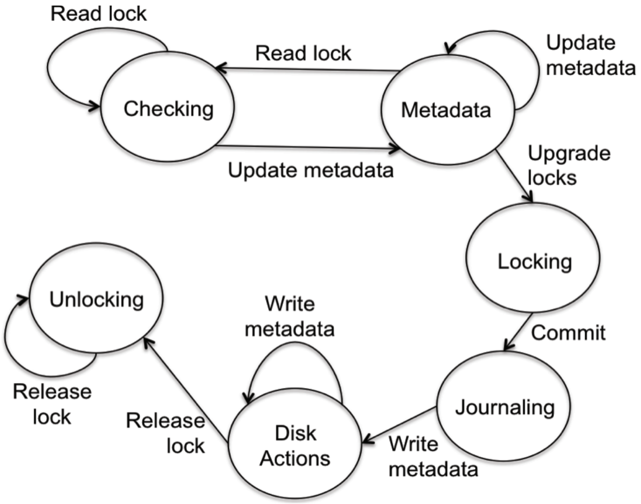


Figure 7: Optimistic locking based transaction state machine

Optimistic locking is based on the premise that all hosts in a VMFS cluster generally operate on mutually exclusive subsets of locks on the volume. Hence, a host that is interested in acquiring a given lock will typically find it to be free on disk. We have observed that most VMFS layouts match the pattern in figure 4 and hence conform to this premise.

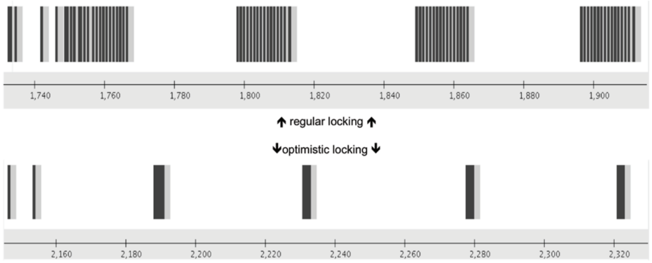
Figure 7 shows how optimistic locking changes the transaction state machine. In particular, the host no longer eagerly acquires locks to all the metadata regions it needs to read and modify in the transaction. Instead, it reads all required locks and if the locks are free, it reads and updates metadata in memory. By the time the transaction is ready to be committed to the journal, the host has built up a list of all locks that the transaction depends on. It then proceeds to acquire all locks using a single SCSI reservation as shown in figure 8 below:

```
1: reserve disk;
2: issue asynchronous (async) reads of all
   required locks;
3: if any lock is acquired by remote host,
   abort and fall back to figure 6;
4: issue async writes of all required locks;
5: wait for all async writes to complete;
6: release disk;
```

Figure 8: Algorithm to acquire optimistic locks (simplified)

The optimistic locking algorithm requires a single SCSI reservation to acquire any number of locks in a transaction. It also has another desirable side effect. Note that steps 2 and 4 in figure 8 pipeline all lock reads and writes to disk. This incurs far less latency compared to the synchronous set of steps from figure 5. In other words, the latency of a $\sum_{1}^{N}(read + write)$ regular locking operation is much higher than the latency of $N*(read + write)$ optimistic locking operation, where the '+' operator means that the current operation is to be executed strictly after its predecessor and the '*' operator means that the operations can be executed in parallel. We now describe some experiments to quantify the differences.

We set up a single ESX host running an extremely metadata intensive workload on the VMFS volume. The workload consisted of concurrent file create, delete, truncate and extend processes in an un-ending loop. Figure 9 shows the characteristic of this workload on the wire when using regular locking and optimistic locking.



Black: reserve, yellow: R-M-W release
Width: duration of operation, #stripes: operation frequency

Figure 9: Regular and optimistic locking event timeline

The X-axis is increasing time and denotes an observation period of approximately 200 milliseconds in both cases. The left edge of the dark stripe indicates the issuance of a reservation command, and the right edge of the light stripe indicates the subsequent release command. The width is the latency of figure 5 in case of regular locking and figure 8 in case of optimistic locking. Optimistic locking creates far lesser chatter on the wire and shows significantly lower operation latencies.

One obvious downside of optimistic locking is that the benefits disappear rapidly in the event of widespread contention on a set of locks. In particular, the optimistic locking algorithm of figure 8 needs to abort the transaction and restart it with regular locking even if a single lock is discovered as being held by a remote host. We ran another set of tests to confirm the benefits of optimistic locking on a cluster of hosts. This experiment ran anywhere from 1 to 4 hosts sharing a VMFS volume and concurrently creating and deleting 32GB virtual disks. As shown in figure 10, optimistic locking significantly reduces the number and latency of reservations and the number of reservation conflicts. In summary, an optimistic locking based clustered lock manager and transaction system is able to sustain a far higher number of transactions – and hence a much higher virtual machine density per VMFS volume.

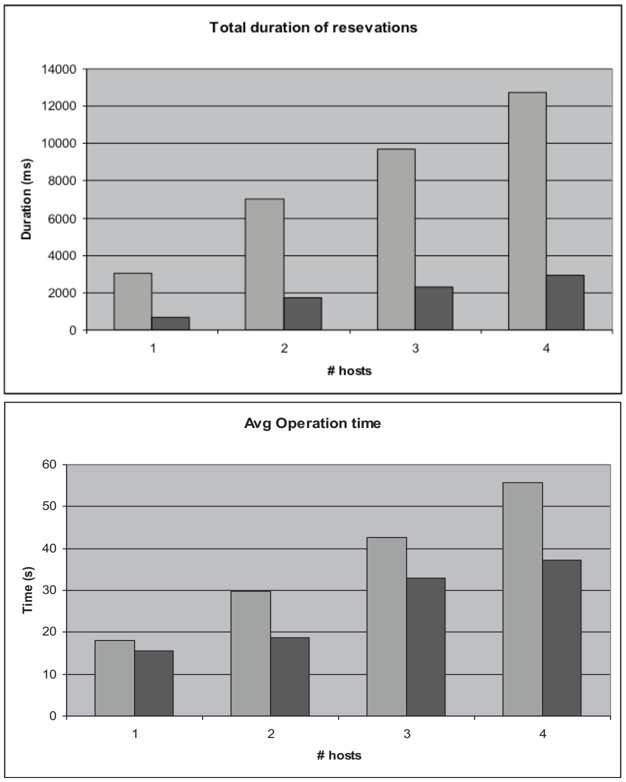


Figure 10: Comparison of optimistic locking (dark shade) and regular locking (light shade) on concurrent transactions from multiple hosts

3.8 Dynamic resource sizing

We alluded to the fact that resource cluster sizing (and hence locking and transaction complexity) is fraught with performance versus contention tradeoffs. Since we have the advantage of dynamic resource sizing by means of changing various values in the system file headers, we decided to implement heuristics to determine optimal values of resources per cluster, clusters per group, etc for various resource types on a VMFS volume at format time. We call this feature 'dynamic resource sizing' and the following sections describe some of the factors the VMFS driver takes into account while running sizing heuristics.

We mentioned earlier that VMFS is designed for SAN storage systems and these systems have larger non-volatile caches. The cache block size varies on different platforms, but it is in the order of a few kilobytes. Figure 3 shows that a resource cluster group's locks and metadata occur at the beginning of the cluster group. Therefore, a good starting point is to size the number of resources per cluster and clusters per group such that the locks and metadata region for a cluster group is (1) aligned to the storage system's cache block boundary, and (2) a multiple of cache pages, and ideally within the storage system's readahead range. The number of resources per cluster can further be refined on similar principles: resources that change often should be aligned to cache block boundaries and locality of reference should resolve to a contiguous region of disk in multiples of cache blocks. Note that the total number of resources per group will determine the alignment of the beginning of the next cluster group (and hence its locks and metadata). Further, since all this is contained in system files, which themselves store data as file blocks, the layout manager considers the possibility that physical discontiguities may arise at file block boundaries. It is preferable to avoid discontiguity in the lock and metadata region of a cluster group. As a corollary, when the VMFS resource manager scans for free resources, it reads in locks and metadata of the entire cluster group in a single operation.

Another consideration for dynamic resource sizing is the size of the disk itself. This yields the approximate number of objects that will come to live on a given volume. Over-provisioning features in VMFS such as thin provisioning [13] and snapshots skew the estimate. However, there are ways to correct this skew. We consider the average server hardware platforms that may run a given generation of the ESX Server product. Given the CPU, memory and IO capabilities of the server platform, the interconnects used in the fabric, and the average number of hosts in a cluster, we can model the total number of VMs that may reside on a given VMFS volume. We further tune our layout assuming a uniform distribution of VMs across these ESX hosts. Finally, transactions are proportional not to the number of VMs running on a given volume, but to the number of VMs undergoing state changes (power on, snapshot, Storage vMotion, vMotion, HA failover, etc) per host on a given volume. We use the advertized concurrency limits of our various solutions to estimate this number and further tune the layout.

Figure 11 studies the effects of dynamic resource sizing in isolation or in conjunction with optimistic locking based transactions. We ran a cluster of 1, 2, 4, 8, 16 and 32 ESX hosts sharing a VMFS volume. Each host concurrently created and deleted a 30GB virtual disk for a finite number of times.

The 'reg 3.21' bar is the baseline average latency per host, established by running the workload on a VMFS volume that uses static resource sizing (fixed, but reasonable values for resources per cluster, cluster per group, etc) and regular locking based transactions. The 'opt 3.21' bar is the average latency when all hosts use optimistic locking. The reg 3.31 bar is with a volume formatted with dynamic resource sizing, but running regular locking based transactions. Finally, the 'opt 3.31' bar is when all hosts run optimistic locking with dynamic resource sizing. Dynamic resource sizing greatly decreases metadata operation latency independent of optimistic locking. Based on the results, we concluded that a good layout is very important, but the layout principles may be very different for a clustered file system with SAN storage systems in the back end, as compared to local file systems on direct-attach storage.

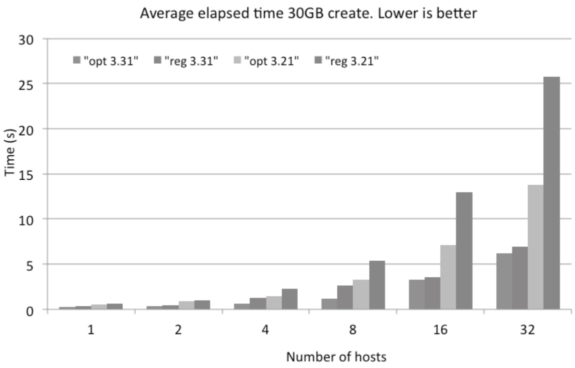


Figure 11: Multi-host performance evaluation of optimistic locking and dynamic resource sizing

3.9 Adaptive SAN-aware retries

We mentioned earlier that one of VMFS' design goals was to free the guest OS from interpreting and reacting to a growing variety of SAN events and errors. In this sense, the virtual disk is an idealized storage device. VMFS and the underlying physical device access layers create this idealized device by monitoring responses to all IO commands issued to the storage system. Some responses, for example, may signal that a (storage area) network path to the device is disconnected. The multipathing module from figure 2 will interpret this response and initiate a failover to a healthier path to the device. New IO requests to the device may be serviced via this new path. At the same time, the virtualization stack will convert the path failure condition to a different error code that indicates temporary disruption when reporting the IO response to the guest OS. The guest OS will choose to retry the failed command according to its internal policies. In effect, the virtual disk was shielded from conditions on the physical fabric. A variety of other events may cause temporary and recoverable physical fabric disruption:

- Queue congestion at the storage system or at the initiator.
- SAN element failures, such as switch port failure, HBA port failure, link removal, storage processor failure, etc.
- Side effects of commands issued by the storage virtualization stack. For example, SCSI reservation conflicts, device on standby path, etc.
- Side effects of task management actions by the storage virtualization stack. For example, IO fencing at the time a clustered host loses its VMFS heartbeat.

- SAN maintenance related. For example, warm-upgrade of storage system firmware, warm-restart of the storage system or switches, etc.
- The consequence of some unrelated (e.g., some other VM's) IO on a given VM's IO. For example, check conditions.

The above list is not exhaustive.

While masking physical fabric conditions from the virtual fabric met our design goals, we could do better. There are two problems in how a guest OS deals with an error. An error condition is observed by VMkernel, possibly transformed to some other error condition, and forwarded to the guest OS. The guest OS needs to be scheduled so that it can receive the IO completion and act on it. The action is typically a delayed retry mechanism to re-issue the failed IO. First, the preset delay is typically based on legacy assumptions about storage behavior. Secondly, the delay is often in response to a transient condition the VMkernel manufactured as opposed to what was really observed on the fabric. In summary, the error condition incurs the latency of delivery to the guest OS and then additional sub-optimal delay before the guest retries the command.

We implemented a mechanism called 'adaptive SAN-aware retries' to enhance this path. The VMkernel categorizes error conditions and other events into various classes. The bulleted list above is one such classification. It keeps a moving average of the duration each event/error class lasts on the physical fabric. If a guest issued IO completes with a known event or error class, the VMkernel now retries the IO after a duration that is a function of the moving average. In doing so, the retried IO has a higher chance of being issued at an optimal time and succeeding. For example, an IO that failed with a reservation conflict may be retried after a duration that is roughly equal to the latency of the 'Locking' step of figure 6 or figure 7. The function used to calculate the moving average and the sample size is determined by the characteristic of the event class it tracks. Our operational experience also taught us that using moving averages on rare but long-lasting error conditions forces the system to waste retry attempts because the moving average that determines the retry interval doesn't adequately sample the impulse event that just occurred. We addressed this by modifying the moving average function to be influenced much more rapidly by high latency samples and less rapidly by low latency samples.

The adaptive retry mechanism has produced good results in the event of widespread SAN events, while causing minimal overhead in the average case. Figure 12 shows results from an experiment involving 2 ESX hosts connected to a shared VMFS volume. Host 1 runs a 1VCPU Windows guest VM with Iometer to a VMFS virtual disk configured as 1 worker, 16 outstanding IOs, 100% sequential write. The first run used 64KB Iometer write size, and the second run used 4KB. These sizes are typical of guest OSes. Host 2 is running a workload that will cause the storage system to reject host 1's IO requests with reservation conflicts and other transient error conditions every few milliseconds. The workload on host 2 causes an error scenario that is representative of worst-case error conditions one would encounter in a real world deployment. We plot IOPS and command latencies observed by Iometer on host 1 as a result of the disruptive actions of host 2. The 'best possible' bar represents the highest IOPS and lowest latencies that host 1 observes with an Iometer run in the absence of host 2. The 'base VMFS' bars are the IOPS and latency readings with adaptive retries turned off. In other words, IO errors are propagated all the way to the VM for the guest OS to retry according to its internal policies. With adaptive retries turned on, one can see that the IOPS and latency readings are very close to the corresponding best case readings.

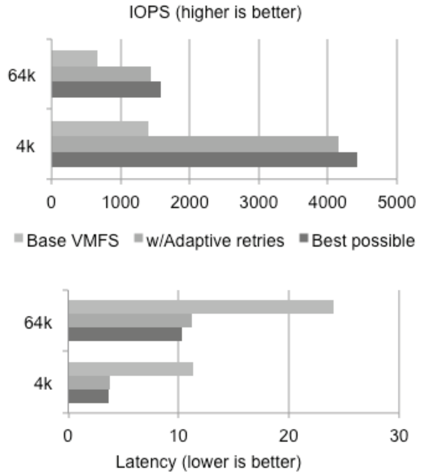


Figure 12: Effect of adaptive retries on VM throughput and latency

4. DESIGNING FOR SANs AND INFLUENCING SAN DESIGN

We mentioned earlier that VMFS design assumes a SAN storage substrate. This section describes some example SAN specific VMFS features that are typically not found in a regular file system.

4.1 Raw Device Mapping

The SCSI virtualization layer from figure 2 changes the outcome of certain SCSI commands received from a VM to a virtual disk, compared to the outcome of the same commands if they were issued from a physical machine to a physical disk. For example, the disk vendor name and model advertised to a VM as part of standard inquiry is `VMware Virtual Disk` instead of the underlying hardware's vendor and model number. This preserves virtual disk portability across hardware and protocol types. However, some applications require the presence of a particular vendor's hardware or support of a set of vendor-specific SCSI commands to function. This use case requires one to directly connect physical disks to virtual machines and passing through all SCSI commands and responses verbatim.

Physical disks are difficult to use in general, and more so in a virtualized and clustered environment that allows a VM to access its storage from different hosts at different times. In particular, we prefer to have cluster-level locked access to a disk resource, a device naming scheme that doesn't change device names based on the scan order, and a place to store extended attributes associated with the device.

VMFS addresses the above problems with physical disks via special file descriptors called Raw Device Mappings (RDMs). While a regular file descriptor stores addresses to file blocks, a RDM stores the hardware ID of the SCSI device it points to. At the time of file open, this hardware ID is used to resolve the request to the correct SCSI device, irrespective of its canonical name in the device file system. Further, the file lock embedded inside the RDM imparts cluster level locking to the device, and moderates concurrent access from multiple ESX hosts to the device. RDM file descriptor metadata stores persistent attributes such as ownership information, mode, time of access, etc.

4.2 Distributed volume resignaturing

SAN storage systems typically export multiple disks to servers. VMFS assembles these disks into one or more volumes. The storage system can create snapshots, i.e. point-in-time (PiT) copies of said disks. We'll refer to the n^{th} snapshot generation of a disk as PiTn. The storage system may export the original disk (PiT0) and any of its snapshots. The file system needs to identify all PiTp disks of a given volume (a volume may span multiple disks), for a discreet generation number p in {0…n} and resignature these disks with a distinct file system identity that is different from the PiTq identity of the file system. This resignaturing step either doesn't exist or has been manual because the server does not know how to make an association between two or more disks belonging to the same snapshot generation and the same file system volume.

VMFS volume manager includes a distributed algorithm to resignature snapshot volumes. Each disk that makes up a volume stores as volume metadata the volume UUID, the volume generation number, the disk ID at the time of volume creation and its sequence number within the volume. A volume spanning one or more disks is uniquely identified by the {UUID, generation} tuple.

```
input S = {all disks visible to volume manager};
for each disk d in S do {
   lock d;
   if (d.SCSI.diskID <> d.volume.diskID) {
      d.volume.gen = d.volume.gen + 1;
      d.volume.diskID = d.SCSI.diskID;
      commit d.volume.diskID and d.volume.gen
         to d;
   }
   unlock d;
}
```

Figure 13: Resignaturing a single extent volume

Figure 13 illustrates an algorithm that can run on any ESX host to detect a snapshot disk and resignature the volume it contains. We rely on the fact that snapshots of a disk result in a new hardware identifier (`d.SCSI.diskID`) that will differ from one stored in volume metadata (`d.volume.diskID`). A longer variant of this algorithm can make association between snapshots of various disks in a spanned volume and assemble snapshotted disks into a spanned volume snapshot.

4.3 Blocklist based backup and restore

Physical machines are typically backed up by running a backup agent in the operating system, which streams the data to a media server, which, in turn, writes it to archival media. Backup traffic consumes a significant portion of CPU, memory and network bandwidth in a large datacenter. LAN-free and serverless backup techniques address these issues, but we took a newer approach designing backup for virtual machines. We leveraged the fact that VMs are hosted on VMFS, and VMFS is primarily used on shared SAN storage. It is therefore possible for a media server to access VMFS disks and back them up without involving the hypervisor in the backup data path. However, media servers do not run ESX and do not understand the VMFS on-disk format. Porting a clustered file system driver for the sole purpose of reading a set of files was a heavy weight approach. Instead, we created a block list protocol and a virtual LUN driver for media servers running the Windows operating system. The block list protocol uses an ESX host to open a file for read access (and therefore acquire a read lock on the file) and export the file's physical extent layout information, i.e. the blocklist. The blocklist is contingent on a lease, and the lease is valid as long as the ESX host can maintain cluster-wide ownership of the file, as determined by the file lock. A virtual LUN driver running in a media server connected to the disks hosting the VMFS volume exports this block list as a virtual LUN to Windows. Backup software running in Windows can back up the entire virtual LUN, or mount the contents of the virtual LUN as a file system for finer grained backups. The virtual LUN driver periodically renews the lease on the blocklist by communicating with the ESX host via the blocklist protocol. In retrospect, the blocklist protocol is similar to a part of the pNFS protocol [21].

In subsequent versions of ESX, the blocklist protocol allowed writable block lists, thus allowing the media server to restore VMDK images by writing to the virtual LUN. On receiving a write to a non-existent part of a virtual LUN (i.e. a part that is not backed by a physical extent on VMFS), the virtual LUN driver sends a block allocation request for the corresponding VMDK offset to the ESX server in charge of the blocklist lease. The VMFS driver on this ESX server allocates the block and returns physical layout information for the newly allocated block to the virtual LUN driver.

4.4 Data mover

Early versions of VMFS were simply optimized to allow wire-rate throughput for IO from VMs to virtual disks. We have witnessed a significant increase in the frequency of data services on virtual disks since then. For example, provisioning a virtual machine, and hence a virtual disk, is an extremely common operation. The virtual disk is typically provisioned from a template virtual disk, and hence the provisioning operation is akin to copying the contents of one file to another. Since virtual disks are extremely large files, it was important to decrease the latency of a provisioning operation. VMFS employs a component called a data mover for this purpose.

As shown in Figure 14, a management application executes a VMDK provisioning request by sequentially reading the contents of the source template file and writing the contents to the destination VMDK file. Data travels from the source physical disk through the SAN fabric to the hypervisor, is copied from the hypervisor to the application and written out immediately from the application to the hypervisor and then to the destination physical disk via the fabric.

The data mover allows the application to convey this provisioning operation at a meta level to the VMFS driver. This request is of the form:

```
clone(srcFileHandle, srcFileOffset,
      dstFileHandle, dstFileOffset,
      length, policies)
```

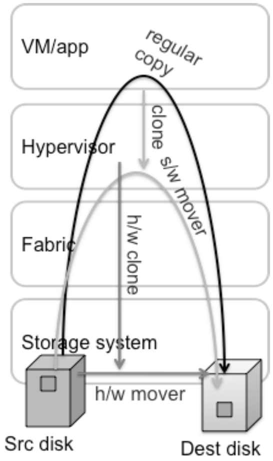


Figure 14: Cloning a file using regular copy, software and hardware data mover

The VMFS data mover breaks the incoming request into a vector of greatest common contiguous physical extents on source and destination disks and issues all requests asynchronously, each asynchronous request being a asynchronous read followed by an asynchronous write. The policies determine request timouts, whether zeroes read should be written out (zero compression), whether null blocks in source file translate to null blocks in destination file, etc. The data mover saves the need to copy data to and from the application and the need for the application to be scheduled to receive and send out data, in addition to the obvious CPU and memory savings.

A newer version of the data mover – a hardware data mover – goes a step further by forwarding the meta-level clone directive to the storage system. Data moves within the storage system, but doesn't use fabric bandwidth and host CPU, memory and IO cycles. Figure 15 shows a performance comparison of the traditional VMFS copy path and the software and hardware data mover path. We cloned a 16GB VMDK across two VMFS volumes.

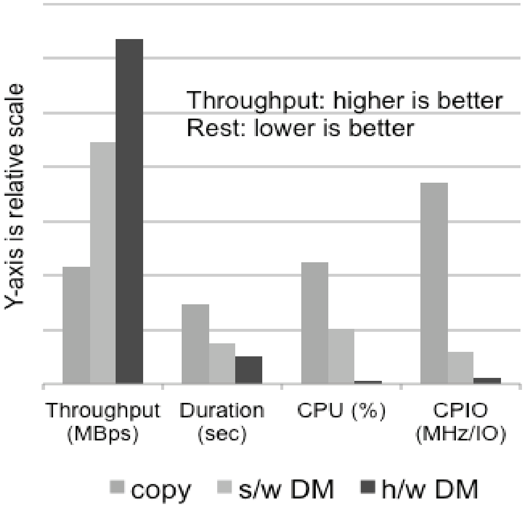


Figure 15: VMDK provisioning performance results with data mover

The hardware data mover is able to achieve approximately 3 times the throughput of regular file copy and uses negligible CPU on the host. The software data mover has a much lower CPU cost than regular copy on account of limiting the data movement to a zero-copy asynchronous kernel IO path.

Since the y-axis is a relative scale, one might mistake the VMFS copy path to be slow given that the data mover exhibits twice the throughput. We'd like to point out that in this particular experiment, the 'copy' throughput bar represents 108.51 MBps, which is sufficiently high in itself.

4.5 Hardware auto-tiering and overprovisioning

In recent years, SAN storage systems have implemented auto-tiering – a single disk, as visible to a server, is composed of a mixture of extents from SSD, SATA and FC spindles. The storage system tries to migrate disk blocks of the storage working set to a higher storage tier. The VMFS dynamic resource layout algorithm tries to size and place resource cluster metadata on storage system extent boundaries so it can be migrated between storage tiers efficiently, if needed.

Our final example of SAN influence on VMFS design is the effect of hardware space overprovisioning. Storage systems use thin provisioning, snapshotting or replication to reduce the space footprint of a workload. These techniques are most effective if the overlying file system (1) refrains from initializing blocks that are allocated but not yet used by the application, and (2) conveys that the contents of a block are no longer important because its corresponding file was just deleted. The aforementioned characteristics also help the storage system reduce the amount of replication traffic, snapshot and replica storage footprint, etc. As discussed earlier, VMFS' use of TBZ blocks delays initialization of newly allocated blocks to the point when the application actually writes to the block. On the file deletion path, VMFS maps the blocks to be deleted to the underlying physical extents and uses the UNMAP SCSI command to convey block deletion to the storage system. The storage system is free to repurpose unmapped physical extents.

4.6 Hardware design for VMFS

VMFS has enjoyed tremendous popularity hosting virtual machines on block based SAN hardware. We presented several examples on how the file system is optimized for SAN hardware. However, SAN hardware is then prevented from directly applying some if its physical disk based data services to virtual machines because it does not understand the layout of virtual disks on physical media. In recent years, we published a set of desired SCSI protocol modifications to allow VMFS to leverage hardware data services for virtual machine disks (i.e. files). The first set of modifications allow VMFS to issue directives of the form

```
operator(VMID,
        source_blocklist,
        destination_blocklist)
```

to the underlying storage system. VMID is a context ID for resource management purposes inside the storage system. The source and destination blocklist are vectors of `<diskID, offset, length>` tuples. Some example operators are zero, clone and delete. Destination blocklist is not required for some operators. The zero operation instructs the storage system to write zeroes to source blocklist, the clone operator copies data from source to destination blocklist, and the delete operation causes the physical media backing the source blocklist to be repurposed. Another primitive:

```
        atomic_test_and_set(block_number,
                             old_image, new_image)
```

also known as ATS causes the storage system to atomically change the contents of the given block number if the current contents match a VMFS provided `old_image`. This enables the VMFS lock manager of figure 5 to move from a 4-IO-operation SCSI reservation based locking algorithm to a 2-IO reservation-less locking algorithm. The new algorithm reads a lock image from disk, and if the lock is free, issues an `atomic_test_and_set` with a `new_image` containing host specific hostID, generation and heartbeat information. The algorithm does not disrupt IOs from other hosts sharing the VMFS volume. Our tests with up to 8 ESX hosts concurrently booting 64 VMs each show that the file system micro operation latency is between 5% and 160% higher when using a reservation based lock manager under severe contention, compared to an ATS based lock manager. This is illustrated in Figure 16.

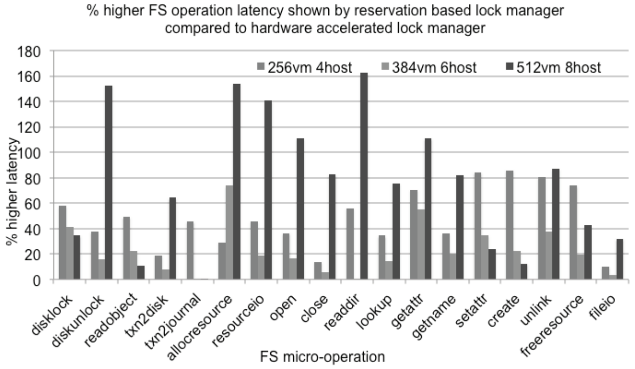


Figure 16: Effect of locking algorithms on micro FS operation latency during a multi-host VM boot storm

These VMFS hardware primitives are implemented by most SAN storage platforms today [8][10][11]. While the primitives are simple enough to justify a storage system firmware change, they are enhanced with host side algorithms to have far reaching impact on various VM use cases. For example, the clone operator forms the basis of hardware data movement which, in turn, results in hardware accelerated VMDK provisioning, Storage vMotion and VMDK snapshotting. Figure 15 summarized the effects of hardware accelerated VMDK provisioning.

While hardware primitives might largely be viewed as a means to increase performance and scale, the end goal of hardware assisted operations in VMFS is to build a high quality storage execution and reporting engine for policy based management of virtual machines. We believe this is best accomplished by blurring the boundary between where VMware storage virtualization ends and the storage hardware layer begins. We will end with an example to our point. File systems report storage utilization via the `stat` system call. However, a block that is considered to be in use and allocated to a file, may be a deduplicated shared block inside the storage system, or a snapshot-shared block or even a non-existent block if the application never wrote to that part of the file. A more comprehensive way to report space utilization for a virtual disk file, then, is to merge file system level block usage and sharing stats with hardware level block usage and sharing stats. This would better cater to a potential raise-alarm-if-utilization-exceeds-X storage policy on a VM.

5. CONCLUSION

We presented VMFS, a symmetric clustered file system custom built to meet the requirements of virtual machine IO and management operations. The founding goal of the file system was to provide physical device level IO performance to virtual machines, while providing file system level manageability for virtual machine management. VMFS clustering capabilities are trivial to set up and almost invisible to ESX administrators. These clustering functions are available to applications via a well-known POSIX API, and it forms the basis of many VMware solutions such as Fault Tolerance, High Availability, and Distributed Resource Scheduling. The requirements of newer VM solutions have resulted in the addition of innovative locking, data movement, error resolution and other algorithms in VMFS.

VMFS is best deployed on enterprise-class block storage systems with differentiated data services, and contains many features to leverage the underlying hardware services. This file system successfully hides the complexity of SAN hardware and clustering from the end user, and is largely responsible for the success of block storage systems in the context of virtualization. In recent years, the file system has inspired storage system and protocol changes across many vendors, such that virtual machine operations can now be conveyed to the storage system at the meta-level instead of typical reads and writes. We have witnessed unprecedented storage efficiency gains due to this change.

VMFS has met the operating requirements of four generations of VMware hypervisor deployments all over the world. Along with ESX storage virtualization stack and abstractions such as exclusive locks, virtual disks and snapshots, it set a standard in hosting and managing virtual machine storage.

6. ACKNOWLEDGEMENTS

VMFS is joint work with my extraordinary colleagues Abhijit Paithankar, Mayank Rawat, Faraz Shaikh, Murali Vilayannur, and Krishna Yadappanavar, who form the development team. I am honored at the opportunity to write this paper on the team's behalf. Thanks to Jinyuan Li, Mike Nelson, Abhishek Rai, Manjunath Rajashekhar, Dan Scales, Dragan Stancevic, and Yuen-Lin Tan who contributed to VMFS in the past. Tejasvi Aswathanarayana, Adam Belay, Sujay Godbole, Alexander Mordkovich, and Ilia Sokolinski for integrating storage hardware acceleration features for VMFS to feast upon. Christian Czezatke for the first version of the blocklist protocol based backup solution. Jim Chow and Ben Verghese for reviewing this paper. And the thousands of VMware customers who trust VMFS with exabytes of data and make all this worthwhile.

7. REFERENCES

[1] VMware vSphere hypervisor http://www.vmware.com/products/vsphere-hypervisor/index.html

[2] VMware vSphere 4.1 Configuration Maximums http://www.vmware.com/pdf/vsphere4/r41/vsp_41_config_max.pdf

[3] VMware vStorage VMFS http://www.vmware.com/products/vmfs/

[4] Nelson, M., Lim, B.H., Hutchins, G. 2005. *Fast Transparent Migration for Virtual Machines.* Proceedings of USENIX ATC.

[5] VMware Distributed Resource Scheduler http://www.vmware.com/products/drs/

[6] VMware Fault Tolerance http://www.vmware.com/products/fault-tolerance/

[7] VMware High Availability http://www.vmware.com/products/high-availability/

[8] Storage Hardware Acceleration – SAN configuration guide http://www.vmware.com/pdf/vsphere4/r41/vsp_41_san_cfg.pdf

[9] VMware Pluggable Storage Architecture and NMP http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1011375

[10] Vaghani, S. 2009. VMware vStorage VMFS-3: Architectural Advances since ESX 3.0 at VMworld 2009 San Francisco. http://www.vmworld.com/docs/DOC-3843

[11] Nguyen, L and Vaghani, S. 2010. Next Generation VM Storage Solutions with vStorage API for Array Integration (VAAI) at VMworld 2010 San Francisco. http://www.vmworld.com/docs/DOC-4667

[12] Understanding virtual machine snapshots in VMware ESX http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1015180

[13] VMware vStorage Thin Provisioning http://www.vmware.com/products/vstorage-thin-provisioning/

[14] VMware Storage vMotion http://www.vmware.com/products/storage-vmotion/

[15] SCSI Primary Commands – 4 http://www.t10.org/cgi-bin/ac.pl?t=f&f=spc4r27.pdf

[16] SCSI Block Commands –3 http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc3r25.pdf

[17] VMware vCenter Server http://www.vmware.com/products/vcenter-server/

[18] Trivedi, K. 1980. *Optimal Selection of CPU Speed, Device Capabilities, and File Assignment*", Journal of the ACM, 27, 3. pp 457-473

[19] McKusick, M. K. et al. 1984. *A fast file system for Unix*

[20] Yang, J et al. 2004. *Using Model Checking to Find Serious File System Errors* at Operating System Design and Implementation (OSDI)

[21] RFC 5663 Parallel NFS (pNFS) Block/Volume Layout http://tools.ietf.org/search/rfc5663