Capo: Recapitulating Storage for Virtual Desktops

Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova,
Norman C. Hutchinson and Andrew Warfield
Department of Computer Science
University of British Columbia

Abstract

Shared storage underlies most enterprise VM deployments because it is an established technology that administrators are familiar with and because it good job of protecting data. However, shared storage is also very expensive to scale. This paper describes *Capo*[1], a transparent and persistent block request proxy for virtual machine disk images. Capo reduces the load on shared storage by using local disks as persistent caches, using multicast-based preloading to broadcast read results across a cluster, and by imposing *differential durability* – dividing a VM's file system into regions of varying writeback frequency. We motivate the system's design through the analysis of a week-long trace of 55 production virtual desktops and then describe and evaluate our implementation. Capo is particularly well suited for virtual desktop deployments, in which large numbers of VMs boot from a small number of "gold master" images and are refreshed on a periodic basis.

1 Introduction

The storage we trust is expensive. Fast and reliable data storage is something that organizations are prepared to pay a premium for, both in the capital costs of enterprise storage hardware and the operational costs of ensuring that important data is written to it.

Interestingly, the deployment of virtualization has inverted the historical imperative that systems be configured to "opt-in" to storing data on appropriate network shares instead of on less reliable locations such as local disks. While administrators used to have to work to configure applications to use enterprise storage, virtual environments simply store *everything* on it.

[1]The name of our system is borrowed from the phrase *"Da Capo al coda"*, which is used in sheet music to indicate a brief return to the beginning of a piece, followed by a jump to the Coda, or conclusion. In sonatas, this "recapitulation" involves revisiting a similar, but sometimes different version of the main theme of the arrangement.

As such, these environments present the opposite problem: The requirement that virtual machine images be universally accessible, with high performance, to all physical hosts in a cluster has necessitated the deployment of SAN hardware in even modest virtualization deployments. The improved density and utilization afforded by virtualization allows systems to scale to large numbers of VMs; shared storage must scale proportionately to provide for them. This symptom is especially problematic for virtual desktops, where infrastructure is being deployed to host literally thousands of nearly identical VM images. A number of commercial virtual desktop systems now exist, and deployments suffer from a significant, if not dominant, cost for enterprise storage.

This paper argues that shared, central, storage *is* the correct approach for scalable virtual environments. It is trustworthy, relatively easy to manage, and simple to reason about. However, we believe that for applications such as virtual desktops, which involve large numbers of image clones, the majority of request load is redundant and can be effectively serviced by local, commodity disks within individual servers. Furthermore, we believe that the levels of durability provided by enterprise storage in these environments are in excess of what is necessary for large portions of desktop OS disk images.

The contributions of this paper are twofold: First, we validate our hypothesis through the analysis of a week-long trace of all storage traffic from a production deployment of 55 Windows Vista desktops in an executive and administrative office of a large public organization. Our results examine the opportunities that exist for caching data both within and across virtual desktop images. They also examine the breakdown of request workload within desktop filesystems.

Second, based on the analysis of this trace, we describe the design and implementation of *Capo*, a distributed persistent cache that aims to reduce aggregate load on shared storage in virtual desktop environments. Capo uses local server disks to provide persistent caching

of VM images, and includes mechanisms to share and pre-load caches of gold master images across VMs and across hosts. Finally, Capo introduces a facility for *differential durability*, which allows administrators to selectively "opt out" of enterprise storage guarantees by relaxing the durability properties of subsets of a desktop's file system.

Virtual desktop systems have already taken advantage of several approaches to scale storage to large numbers of desktop machines. We begin in Section 2 by providing some brief initial background on these systems.

2 VDI Background

Virtual desktops represent the latest round in a decades-long oscillation between thin- and thick-client computing models. So-called Virtual Desktop Infrastructure (VDI) systems have emerged as a means of serving desktop computers from central, virtualized hardware. VDIs are being touted as a new compromise in a history of largely unsuccessful attempts to migrate desktop users onto thin clients, and the approach does provide a number of benefits. Giving users private virtual machines preserves their ability to customize their environment and interact with the system as they would a normal desktop computer. From the administration perspective, consolidating VMs onto central compute resources has the potential to reduce power consumption, allow location-transparent access, better protect private data, and ease software upgrades and maintenance.

Commercial VDI systems appear to be experiencing a degree of success: Gartner predicts that forty percent of all worldwide desktops—49 Million in total—will be virtualized by 2013 [17]. Today, the two major vendors of VDI systems, Citrix and VMWare, individually describe numerous case studies of active virtual desktop deployments of over 10,000 users. From a storage perspective, VDI systems have faced immediate challenges around space overheads and the ability to deploy and upgrade desktops over time. As background, this section describes how these problems are typically solved in existing architectures, as illustrated in Figure 1.

2.1 Copy-on-Write and Linked Clones

Operating system images are entire virtual disks, often tens of gigabytes each. A naive approach to supporting hundreds or thousands of virtual machines results in two immediate storage scalability problems. First, VMs must have isolated disk images, but maintaining individual copies of every single disk is impractical and consumes an enormous amount of space. Second, adding new users requires that images can be quickly duplicated without necessitating a complete copy.

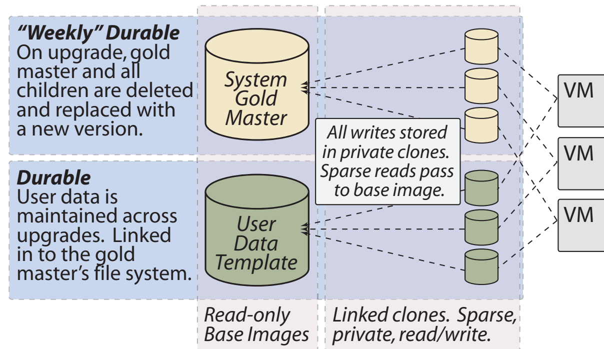


Figure 1: Typical image management in VDI systems.

This observation is not new; it has been a recurring challenge in virtualization. Existing VDI systems make use of VM-specific file formats such as Microsoft's VHD [14] and VMware's VMDK [22]. Both allow a sparse overlay image to be "chained" to a read-only base image (or gold master). As shown in Figure 1, modifications are written to private, per-VM overlays, and any data not in the overlay is read from the base image. In this manner, large numbers of virtual disks may share a single gold master. This approach consolidates common initial image data, and new images may be quickly cloned from a single gold master.

2.2 Image Updates and Periodic Rollback

Image chaining saves space and allows new images to be cloned from a gold master almost instantaneously. It is not a panacea though. Chained images immediately begin to diverge from the master version as VMs issue writes to them. One immediate problem with this divergence is the consumption of independent extra storage on a per-image basis. This divergence problem for storage consumption is typically addressed through the use of data deduplication [24, 6, 4].

For VDI, wasted storage is not the most pressing concern: block-level chaining means that patches and upgrades cannot be applied to the base image in a manner that merges and reconciles with the diverged clones. This means the ability to deploy new software or upgrades to a large number of VMs, which *was* initially provided from the single gold master is immediately lost.

The leading VDI offerings all solve this problem in a very similar way: They disallow users from persisting long term changes to the system image. When gold master images are first created and clones are deployed, the VDI system arranges images to isolate private user data (documents, settings, etc.) on separate storage from the system disk itself. As suggested initially in the Collective project [3], this approach allows a new gold master

with updated software to be prepared and deployed to VMs simply by replacing the gold master, creating new (empty) clones, and throwing away the old version of the system disk along with all changes. This approach effectively "freshens" the underlying system image of all users periodically and ensures that all users are using a similar well-configured desktop. For the most part, it also means that users are unable to install additional, long-lived software within VDI images without support from administrators.

3 Virtual Desktop Trace

To better understand VDI workloads, we arranged to measure all block and file level activity from the a production VDI deployment for a one week period during the Summer of 2010. The deployment being studied is an office in a large public organization containing executive and administrative support staff. The deployment had been in production use for six months and includes 55 Windows Vista desktops, the organization is in the process of rolling out another 300 desktops this fall.

3.1 Methodology

We installed a Windows storage class driver into the base system image of the virtual desktop machines. The driver was written to record block read and write events to the virtual disks using the Microsoft Windows Software Trace Preprocessor (WPP). It recorded request size and virtual disk address. In 93% of cases we were also able to determine the file on which the access originated by following the OriginalFileObject pointer in the Windows I/O Request Packet (IRP) structure. To better contextualize this information, we also installed a driver at the filesystem level and recorded cache accesses, the application making each request, and the file flags for each file accessed. Our disk-level driver is written in 515 lines of C, while our file-level driver is 82 lines of C.

Logs from these drivers were written to a network share and collected on the Thursday following a full week of logging. In total we collected 75GB of logs in a compressed binary format. We then checked for corruption, missing logs, or missing events. Out of over 300 million entries we found a single anomalous write to a clearly invalid block address, which we removed. We could find no explanation for the event. In the rest of this section we present our analysis of this data. Unless otherwise specified, we will refer to block level accesses to a virtual disk and measure aggregate workloads in I/O operations per second (IOPS).

3.2 Our Virtual Desktop Environment

The environment we are studying is structured very much like the one described in Section 2. At the time our measurements were gathered it hosted 55 Microsoft Windows Vista virtual desktops with VMWare View, of which roughly 27 are in dedicated day-to-day use as the primary desktop. This small size is the primary limitation of our study, but we expect to measure considerably more as the installation grows. Furthermore, even at the current size it is possible to see considerable self-similarity among machines, as we will discuss.

End users work from Dell FX100 Zero thin clients, while VMs are served from HP BL490c G6 Blades running ESX Server. These servers connect to a Network Appliance 3170s over fiber channel, for booting from the SAN, and 10GigE, for VM disk images. System images are hosted via NFS on a 14 drive RAID group with 2 parity disks. The operating systems and applications are optimized for the virtual environment [20] and are pre-loaded with Firefox, Microsoft Office Enterprise, and Sophos Anti-Virus among other software. At the end of every Wednesday, a new system image is published to all users exactly as discussed in Section 2.2.

3.3 Analysis

We begin by asking, ***What are the day-to-day characteristics of VDI storage workloads?*** Figure 2 shows the entire study in I/O operations-per-second for the 24-hour period of each of the 7 full days recorded. There is a distinct peak load period between 8:30 and 9:30 every morning, as employees arrive at work. Three peaks in this period are highlighted in the figure and presented for expanded analysis in Table 1. The right-most column shows the applications responsible for the most disk I/O, excluding the system and services. Most days, Firefox and the virus scanner are very active in this period, we also see Thunderbird, Pidgin, and Microsoft Outlook frequently. We were surprised to see the Search Indexer active as well, because we were told its background scanning task had been disabled to reduce I/O consumption. Our best guess is that it was invoked manually.

We measured the write to read percentages for both IOps and throughput, which is useful in characterizing the workload. Our workload is write-heavy in IOps, and read-heavy in throughput, both by approximately two-to-one. We then measured the percentage of VMs which contributed at least 5% of the peak workload, to determine if peaks were caused by multiple VMs or by a few outliers. In most cases, it is the former; however, the peak in slice 4 was caused primarily by 4 VMs. The column titled "Dup. reads" illustrates the potential for caching. We present two numbers. The left-most is the percent-

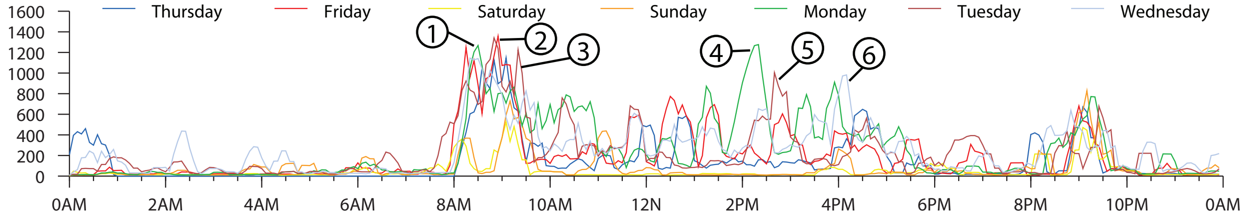


Figure 2: Activity measured in I/O operations over each of the 7 days.

	Time Period	Write % (IOps / Thpt)	% of VMs ($\geq$ 5%)	Dup. reads (VM / Clust.)	Top Applications by IOps (excludes system and scvhost)
1	Mon. 8:30am-8:45am	50% / 22%	26%	81% / 91%	Search Indexer, Firefox, Sophos
2	Fri. 8:30am-9:00am	52% / 22%	29%	88% / 97%	Firefox, Search Indexer, Sophos
3	Tue. 9:15am-9:30am	64% / 43%	29%	78% / 99%	Defrag, Firefox, Search Indexer
4	Mon. 2:00pm-2:30pm	62% / 41%	7%	59% / 99%	Firefox, Pidgin, Sophos
5	Tue. 2:40pm-3:00pm	69% / 52%	26%	77% / 97%	Firefox, Defrag, Pidgin
6	Wed. 4:00pm-4:15pm	60% / 37%	26%	99% / >99%	Firefox, Pidgin, Sophos

Table 1: Points of interest in Figure 2.

age of reads that have been previously seen by that VM over the course of the trace. With a large enough cache, we could potentially absorb *all* these reads. The right-hand column presents the same measure, but imagines that caching could be shared across all VMs in the cluster. Slice 4 stands out for having an unusually low duplicate read rate for VMs, but a very high rate across the cluster as a whole. We investigated and found that two very active VMs had duplicate read rates of 26% and 30%. By including the least beneficial 38%, 15% and 4% of VMs, we could reach duplicate read rates of 40%, 60% and 90% respectively. From this we conclude that you can achieve significant improvements with caching, possibly even by sharing caches, but that some benefits may require careful selection of the VMs in question.

Lunch, dinner, and late nights are periods of relative inactivity, as are the weekends. Late afternoon peaks are sporadic, but reach loads nearly as high as the morning. One such peak, marked 4 in Figure 2 and Table 1, was caused by a relatively small number of machines engaged in heavy browsing activity. This is not the norm, as all other peaks occur when more than a quarter of the VMs are significantly active. This is clear in Figure 3, which shows a CDF of VMs by their contribution to the total workload for each peak. These peak load periods are particularly important, because they define the hardware necessary to service the workload without disruption. We conclude that, ***VDI workloads are defined by their peaks, and those peaks usually occur at times of common activity among many VMs.*** In Sections 4.1 and 4.2 we take advantage of this fact to improve performance in VDI environments.

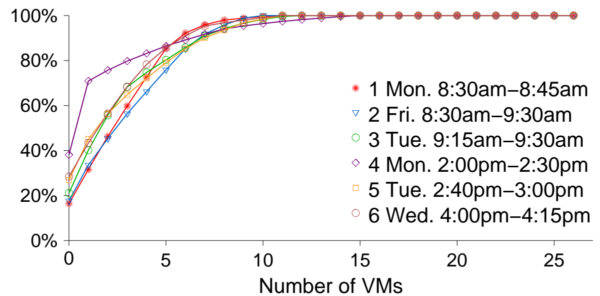


Figure 3: Contributors to each of the major workload peaks.

Next, we ask, ***How can we characterize the I/O requests?*** In Figure 4 we show the entire workload by both request count and size. We differentiate reads from writes, and also isolate each request by its target in the file system namespace. The workload is 65% writes, which account for 35% of the throughput, versus 35% of reads accounting for 65% of the throughput. Metadata operations account for large portion of the requests; unfortunately we cannot determine how these modifications relate to the namespace. Directories typically managed by the operating system, such as `\Windows` and `\Program files` are also frequently accessed. There are fewer accesses to user directories and temporary files; most of the latter are to `\Temporary Internet Files`, as opposed to `\Windows\Temp`. These findings contrast those of Vogels who's study showed that 93% of *file-level* modification occurred in `\User` directories [23]. We conclude that, ***while a wide range of***

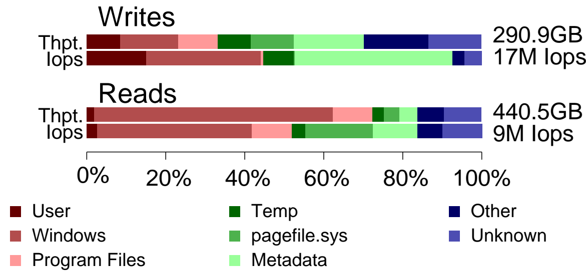


Figure 4: Size and amount of block level writes by file system path.

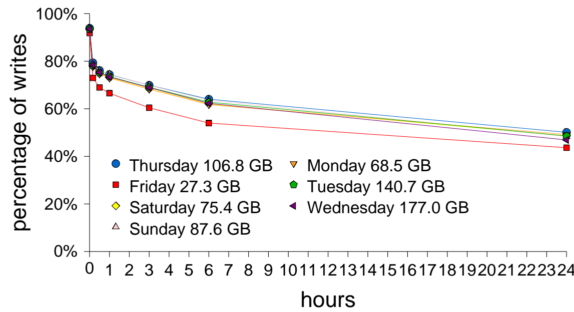


Figure 5: Percentage of bytes that need to be written to the server if writes are held back for different time periods. This is lower than the original volume of writes due to the elimination of rewrites.

the namespace is accessed, it is not accessed uniformly, and access to data directly managed by users is rare. We will revisit this observation in Section 4.3.4.

Since our workload is write-heavy, we next ask, ***how are these writes organized in time?*** Figure 5 shows the percentage of disk writes that overwrite recently written data, for time intervals ranging from 10 seconds to a whole day. We have included results from each of the seven days to underscore how consistent the results are. In a short time span, just 10 seconds, 8% of bytes that are written are written again. This rate increases to 20%-30% in 10 minute periods and ranges between 50%-55% for twenty-four hour periods. From this we conclude that, ***Considerable system-wide effort is spent on data with a high modification rate.*** We show how this can used to our advantage in Section 4.3.

Since VMs typically use disk images chained from a gold master, we are interested in the rate at which the overlay image diverges from the original image. We therefore ask, ***At what frequency do we observe the first write to a sector?*** Figure 6 plots this data for the average VM, as well as the most and least divergent VM, over the entire study. Within 24 hours, most VMs hit a

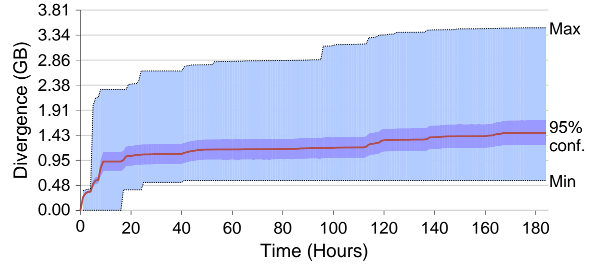


Figure 6: Bytes of disk diverging from the gold master.

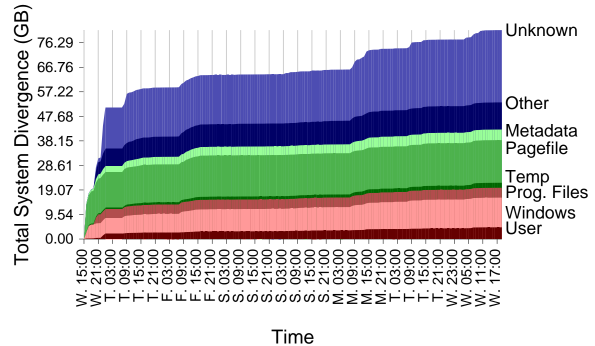


Figure 7: Total divergence versus time for each namespace category.

near plateau in their divergence, around 1GB. Over time this does increase, but slowly. A smaller set of VMs do diverge more quickly and significantly, but they are far from the 95% confidence interval. We conclude that, ***there is significant shared data between VMs, even after several days of divergence.***

Naturally, we do not expect divergent writes to occur uniformly, so we pose a question: ***Where in the namespace do divergent writes occur, and does this change over time?*** Figure 7 plots the cumulative divergence for each VM in the cluster, and divides that total among various components of the namespace. One can observe that the pagefile diverges immediately, then remains a constant size over time, as does the system metadata. Both these files are bounded in size. Meanwhile writes to `\Windows` and areas of the disk we cannot associate with any file continue to grow over the full week of the study. We conclude that, ***While writes occur everywhere in the namespace, they exhibit significant trends when categorized according to the destination.***

3.4 Summary

While there is more to say about this workload and those of VDI environments in general, the observations in this section are valuable. Summarizing our observa-

tions from the trace data:

- VDI workloads are defined by their peaks, and those peaks usually occur at times of common activity among many VMs
- While a wide range of the namespace is accessed, it is not accessed uniformly, and access to data directly managed by users is rare
- Considerable system-wide effort is spent on data with a high modification rate
- There is significant shared data between VMs, even after several days of divergence
- While writes occur everywhere in the namespace, they exhibit significant trends when categorized according to the destination

These observations taken together suggest that additional caching, combined with an awareness of namespace organization might resolve the performance challenges that we have observed. The following section builds on the observations and analysis presented here, and describes the architecture of Capo.

4 Architecture

The trace analysis in Section 3 suggests that caching below the individual VMs may be effective in resolving the demand peaks that we observed. In this section we describe Capo, including its three major components:

1. A single-host cache which eliminates redundant reads and writes from virtual desktops hosted on the same server.
2. A multi-host cache preloader which eliminates redundant reads from virtual desktops hosted on different servers.
3. A component that supports *differential durability*, which modifies cache coherency based on the location in the namespace of the affected file.

Figure 8 shows the overall architecture of Capo. Capo exists as a layer within the virtual machine monitor (VMM) which supports the individual desktop VMs. The figure depicts each host including a Local Persistent Cache which is stored on the local disk of the host machine and is described in Section 4.1. Spanning all of the hosts is the Transparent Multi-host Prefetch component which optimistically preloads data accessed by one host into the local caches of the other hosts. It is described in Section 4.2. The Durability Map component supports the wide variation in the durability requirements of the various components of the file system. It is described in Section 4.3.

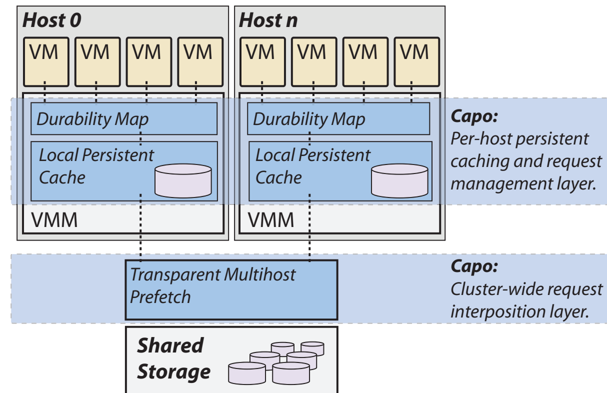


Figure 8: The Major Architectural Components of Capo.

4.1 Local Persistent Cache

All VDI deployments rely on central enterprise storage that provides high availability, durability, and reliability. The servers that host virtual desktops are also configured with local disk storage which consists of cheap COTS disk drives with comparably lower reliability but higher aggregate I/O bandwidth.

The trace-based analysis of our local VDI deployment suggests that a cache shared between multiple virtual desktops might be very effective. As shown in Figure 3, there is significant overlap between the top applications executed on different virtual machines. Table 1 refines this and indicates that aggressive caching can yield very high read hit rates. Also, as shown in Figure 5, a significant fraction of data is overwritten very quickly. Therefore, as depicted in Figure 8, each server machine that hosts virtual desktops uses its local disk as a persistent cache. A key goal for Capo is to provide an appropriate level of durability for all data while taking advantage of the higher aggregate bandwidth available to local disk. The level of durability achieved depends on the caching policy in place.

4.1.1 Caching Policies

The cache supports two consistency policies: write-through and write-back. These policies are enforced at disk image granularity. Each of these policies represents a different tradeoff between virtual disk consistency and overall system performance.

The *write-through* policy provides the highest level of consistency guarantees a machine would expect from a block device. In this policy the cache replicates writes to both the centralized storage and the local cache. Write requests are not acknowledged until they hit both disks. This policy relieves the centralized storage from serving reads to blocks that have been previously read from or written to. The drawback of this policy is that write re-

quests must be sent across the network, consuming network bandwidth and increasing both the load on the centralized storage and the client's perceived write request latency.

The *write-back* policy delays pushing updates to disk blocks by caching writes locally in the write cache. Updates are pushed to the central storage in a crash consistent manner at a per-virtual-disk configurable frequency. The choice of write-back frequency trades off system performance and durability of disk contents in case of a failure. A high update frequency minimizes the amount of data loss in case of the failure of the local disk, while a lower frequency enhances overall system performance by coalescing writes in the local cache.

4.1.2 Design and Implementation

The local persistent cache is implemented as an extension to the publicly available XenServer 5.6 release. It runs in Xen's "domain 0" VM and interposes on the block request path below virtual machines. Cached data is stored as sparse image files in a Linux file system. Each virtual disk's cache consists of either two or three components, shown in Figure 9: a read store, write store, and possibly a snapshot store. Each of these components is represented using a data file and a bitmap in the persistent cache. The bitmap's purpose is to identify which sectors of the corresponding data file are valid. Writing a sector to a cache component involves writing the sector's data to the data file and setting the sector's corresponding bits in the on-disk bitmap.

Write requests are satisfied by writing their data sectors to the cache's write store. When the cache is set to write-through, the sectors are also written concurrently to the centralized storage. Read requests are satisfied by first checking the write store, then the snapshot if it exists, and finally the read store. At each layer, if a sector is valid as specified in the bitmap, the data can be returned immediately from that layer. If none of the stores contain valid data, the sectors are read from the centralized storage, written to the read store, and returned to the client VM.

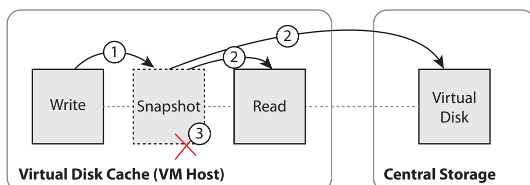


Figure 9: Capo's virtual disk cache components and snapshot procedure.

The snapshot mechanism in the cache works in tandem with a transactional update mechanism in the backend storage to ensure crash consistent updates to remote disk images when operating in write-back mode. Pushing updates to the backend storage involves three steps, as shown in Figure 9. First, a write cache snapshot is created by pausing the request stream momentarily and moving the contents of the cache's write store to the snapshot store. Secondly, the snapshot contents are applied transactionally to the centralized storage and to the read cache concurrently. Finally, after the snapshot updates have been applied, the snapshot store is cleared.

The cache is implemented as a user-level shared library that interposes on I/O calls, specifically the glibc and libaio I/O and file management operations. Due to the relative sizes of the disks in virtual desktops (around 10GB) and the disks in the physical machine which hosts them (greater than 1TB) and the amount of sharing between virtual desktops (see Figure 6), we can easily support hundreds of virtual desktops on a server without worrying about overfilling the cache. In our implementation, when the cache does fill, we simply throw it away and start again.

4.2 Multi-host Cache Preload

Capo's local persistent cache goes a long way towards eliminating redundant read requests on individual machines. But as growing VM deployments lead to larger numbers of physical hosts, redundant reads across these hosts place additional burden on central servers. Further scalability improvements can be attained in this case by multicasting common data to all hosts simultaneously rather than to each host individually.

To this end, we have developed a multicast cache preloader for local caches. The preloader is completely lock-free and requires no modifications to existing centralized servers. It consists of a service which observes network traffic to and from the central storage server. Clients on each host contact the service and register watches for files which are determined to be good candidates for preloading. The service captures any reads made to these files and distributes the results to all subscribed clients via multicast. In this way, the first host to read common data essentially prefetches it for all other hosts.

4.2.1 Design and Implementation

Our initial design for the preload server was to use a mirror port on the central storage server to monitor network traffic. As in *Ditto* [5], our server captured raw network packets and reconstructed TCP flows to extract relevant data (in our case, NFS requests and responses). When deploying this solution, however, we observed signifi-

cant packet loss between the mirror port on the filer and our server, and since a single packet loss is enough to corrupt an entire NFS request or response, we were missing many opportunities to preload data.

Our second, and current, design employs a user-level NFS proxy that sits between the clients and the filer. NFS requests and responses are routed through the proxy, and the proxy identifies data that should be preloaded into other local persistent caches. This increases the latency of filer requests somewhat, but avoids all of the issues with packet capture.

In the current implementation, NFS clients are left unmodified. Instead, a single preload client runs on each physical host. On startup, these processes register watches with the server for files known to be shared across hosts. Because this data is predominately read-only, no synchronization is required when multicast clients update the local caches. When the preload server observes reads to files being watched by clients, it multicasts the responses to all clients.

Because NFS clients are unmodified, reads of shared files result in two responses: the unicast response to the original requester, and a second multicast response to all subscribed clients. This leads to an increase in overall read bandwidth consumption from the proxy to the clients, but reduces the load on the storage server. The redundant unicast response could easily be avoided by making NFS clients aware of the multicast service.

We also currently prioritize unicast responses over multicast responses. This limits the latency overhead seen by NFS clients while delaying preloading on other clients, making it slightly more likely that they will submit unicast requests for the same data. With modified NFS clients, we could more viably prioritize multicast responses, improving the efficacy of preloading.

The preload server sends a significant amount of traffic over a number of multicast sessions, and has exposed problems with the support for multicast in some modern switches. On some of the switches that we have experimented against, multicast packets appear to consume a disproportionately large amount of resources. As a result, even relatively low-throughput multicast traffic has resulted in packet drops with detrimental consequences for concurrent TCP connections. The results can be dramatic: early experiments with completely unthrottled multicast traffic resulted in NFS throughput drops from 100MB/sec to 3MB/sec.

To address this, we have implemented a rudimentary, adaptive flow-control protocol, similar to one described in *SnowFlock* [13]. Each packet sent by the server is associated with an epoch. The server periodically updates the epoch number, and when clients notice a new epoch, they send a message indicating the number of packets successfully received during the previous epoch. In this way the server gets feedback about packet drop rates and is able to vary transmission rates accordingly. An additive increase/multiplicative decrease scheme with aggressive back-off has produced reasonable results in our benchmarks.

This flow-control protocol – and preloading in general – is strictly best-effort: no work is wasted trying to retransmit dropped multicast packets. If the preload clients fail to receive multicast updates for required data, it will eventually be fetched via the conventional unicast path.

The client logic for deciding which files to preload is simplified by a few basic design principles. We assume that, given a number of VMs derived from a common master image, reads of the base image made by any individual VM will likely be duplicated by all VMs. That is, while the disks belonging to derived VMs will tend to diverge as the VMs age, the common portion of these disks will likely be read by all or none of the VMs. Thus if the multicast server observes any read of a common file, it is worth sending this data to all hosts on which Capo is caching this file. By the same assumption, multicast clients do not pro-actively request data from the server, as they are not in a position to know which portions of files will be read by VMs.

4.3 Differential Durability

Major VDI providers have all adopted the software update strategy proposed in The Collective [3], where user directories are isolated from the rest of the file system. Modifications made to files in the user directories must be durable; users depend on these changes. Capo therefore uses write-through caching on these directories, propagating all changed blocks immediately to the centralized storage servers. Any modifications to the system image can then be performed on all VMs in one step by completely replacing the system images in the entire pool, leaving the user's data unmodified. This impacts durability—any writes to the system portion (e.g., by updating the registry or installing software) will be lost. In this section we use and extend this notion to optimize for our write-heavy workload.

4.3.1 Write-Back Period

As mentioned in Section 2.2, VDI deployments manage system data centrally, regularly replacing the system data seen by each virtual desktop with a clean updated version. While users are allowed to make changes to their system data, these changes are not guaranteed to be durable. Writes to the `\Program Files` directory as part of an application install process, for example, represent work done by a user, but a software installation could easily be repeated if a failure caused this to be nec-

Path	Policy
\Program Files\	write-back
\WINDOWS\	write-back
\Users\ProgramData\VMware\VDM\logs	write-back
\Users\$USER$\ntuser.dat	write-back
\Users\$USER$\AppData\local	write-back
\Users\$USER$\AppData\roaming	write-back
\pagefile.sys	no-write-back
\ProgramData\Sophos	no-write-back
\Temp\	no-write-back
\Users\$USER$\AppData\Local\Microsoft\Windows\Temporary Internet Files	no-write-back
Everything else, including user data and FileSystem metadata	write-through

Table 2: Sample cache-coherency policies applied as part of durability optimization.

essary. It might be acceptable if the loss of such effort was limited to, for example, an hour or even a day. We can set our write-back period for such partially durable files to a corresponding length of time.

4.3.2 Extending Partial Durability to User files

While much of the data on the User volume is important to the user and must have maximum durability, Windows, in particular, places some files containing system data in the User volume. Examples include log files, the user portion of the Windows registry, and the local and roaming profiles containing per-application configuration settings. Table 2 shows some paths on User volumes in Windows that can reasonably be cached with a write-back policy and a relatively long write-back period.

4.3.3 Eliminating Write-Back

There are some system files that need not be durably stored at all. These include files that are discarded on system restarts or can easily be reconstructed if lost. Writes to the pagefile, for example, represent nearly a tenth of the total throughput to centralized storage in our workload. These writes consume valuable storage and network bandwidth, but since the pagefile is discarded on system restart, durably storing this data provides no benefit. The additional durability obtained by transmitting these writes over a congested network to store them on highly redundant centralized storage provides no value because this data fate-shares with the local host machine and its disk. Many temporary files are used in the same way, requiring persistent storage only as long as the VM is running.

We store this data to local disk only, assigning it a write-back cache policy with an infinitely long write-back period. In the event of a hardware crash on a physical host, the VM will be forced to reboot, and the data can be discarded.

4.3.4 Design

Initially, we approached the problem of mapping these policies to write requests as one of request *tagging*, in which a driver installed on each virtual desktop would provide hints to the local cache about each write. While this approach is flexible and powerful, maintaining the correct consistency between file and filesystem metadata (much of which appears as opaque writes to the Master File Table in NTFS) under different policies is challenging. Instead, we have developed a simpler and better performing approach using existing filesystem features.

The path-based policies we use in our experiments can be seen in Table 2; naturally, these may be customized by an administrator. We provide these policies to a disk optimization tool that we run when creating a virtual machine image. The optimization tool also takes a populated and configured base disk image. For each of the two less-durable policies, it takes the given path and moves the existing data to one of two newly-created NTFS file systems dedicated to that policy. It then replaces the path in the original file system with a reparse point (Window's analogue of a symbolic link) to the migrated data. This transforms the single file system into three file systems with the same original logical view. Each of the three file systems are placed on a volume with the appropriate policy provided by the local disk cache. This technique is similar to the view synthesis in Ventana [18], though we are the first to apply the technique with a local cache to optimize performance.

We appreciate that applying different consistency policies to files in a single logical file system may be controversial. The risk in doing so is that a crash or hardware failure results in a dependency between a file that is preserved and a file that is lost. Such a state could lead to instability; however, we are aware of no dependencies

crossing from files with high durability requirements to those with lower durability requirements in practice. Further, we observe that this threat already exists in the production environment we studied, which overwrites system images with a common shared image on a weekly basis.

5 Evaluation

To evaluate the effectiveness of Capo, we first consider how effective differential durability is at reducing write load from unimportant regions of disk. Next, we show the storage reduction achieved by Capo with eleven concurrent users synthesizing active desktop workloads. Finally we show the storage reduction achieved by Capo by replaying I/O logs gathered from a production system (see Section 3) under different caching policies.

5.1 Differential Durability

This section describes several microbenchmarks that evaluate the effectiveness of differential durability in isolation of other features and provide a clearer mapping of end-user activity to observed writes. We applied the policies in Table 2 to several realistic desktop workloads. For each, we measured the portion of write requests that would fall under each policy category.

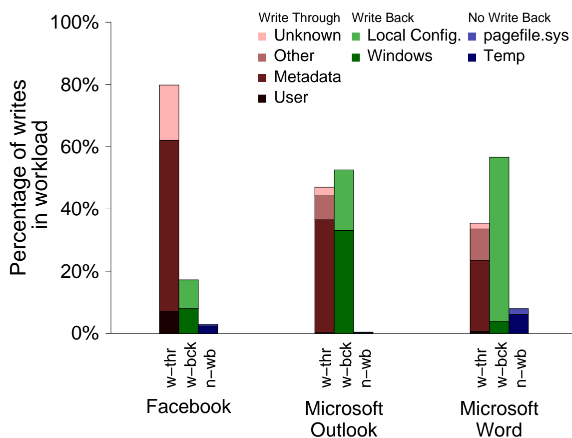


Figure 10: Percentage of writes in three microbenchmarks organized by governing cache-coherence policy.

5.1.1 Web Workload

Our web workload is intended to capture a short burst of web activity. The user opens www.facebook.com with Microsoft Internet Explorer, logs in, and posts a brief message to their account. They then log off and close the browser. The entire task lasts less than a minute. The workload consists of 8MB (43.6% by count) of writes and 25.3MB (56.4% by count) of reads.

A breakdown of writes by their associated policy for each workload is shown in Figure 10. In this short workload only a small, but non-trivial improvement can be made. Local configuration changes such as registry, temp file, and cache updates need not be written immediately, removing or delaying just over 20% of the operations.

5.1.2 Email Workload

Our email workload is based on Microsoft Outlook. The user sends emails to a server we have configured to automatically reply to every message by sending back an identical message. Ten emails are sent and received in succession before the test ends. The workload consists of 63MB (39% by count) of writes and 148MB (61% by count) of reads.

Here the improvement is much more substantial. Although very few writes can be stored to local disk indefinitely, over half can be delayed in writing to centralized storage. This is due to Outlook's caching behavior, which makes heavy use of the system and application data folders. Emails in the .pst file are included in the user category. It is worth noting that many files in the windows and application data are obvious temp files, but did not match our current policies. With more careful tuning, the policies could be further optimized for this workload.

5.1.3 Application Workload

Our application workload is intended to simulate a simple editing task. We open Microsoft Word and create a new document. We also open www.wikipedia.org in Microsoft Internet Explorer. We then proceed to navigate to 10 random Wikipedia pages in turn, and copy the first paragraph of each into our word document, saving the document each time. Finally, we close both programs. The workload consists of 120MB (20.0% by count) of writes and 406MB (80.0% by count) of reads.

Viewing many small pages creates a large number of small writes to temporary files and memory pressure[2] increases the pagefile usage. Both programs write significantly to system folders, leaving less than 36% of the workload to be issued as write-through.

5.2 Multi-host Cache Preload

To evaluate the effectiveness of Capo's multi-host prefetching we boot three Windows XP VMs on three different hosts. The experiment first fully boots one VM

[2]The guest was running Windows Vista with 1GB of RAM, 25% higher than the XenDesktop recommended minimum.

before booting the other two VMs concurrently, with the intention of demonstrating that the reads triggered by the boot on the first host are sufficient to achieve a savings for the later boots.

Figure 11 shows the read workload observed at the server in three different cache configurations: *No cache*, *Write-through* and *Write-through with multi-host preload*. Notice that the read workload for booting the two VMs is roughly double that of booting a single VM for both the *no cache* and *write-through* configurations. On the other hand *write-through with multi-host prefetching* almost eliminates the workload due to booting the two later machines.

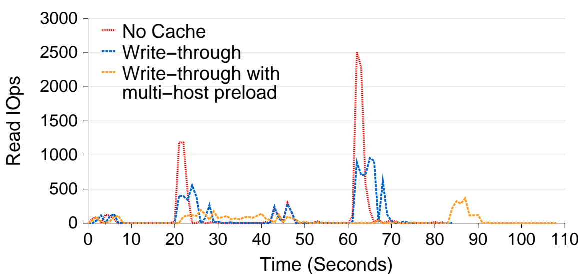


Figure 11: Read IOps per second for booting three VMs on three different hosts.

5.3 Synthetic Workload

To evaluate Capo as a whole, we arranged to simulate a set of (very) active desktop users, performing similar workloads to those seen in the trace. Figure 13 shows the results of request traffic hitting both the local caches (in aggregate across all images) and the filer, while 11 users actively use a variety of office and web-based applications.

First, note that the load in this case is higher than any of the peaks seen in the trace data. This workload represents a higher level of aggregate storage activity than was ever seen in the production environment. Second, observe that despite being configured conservatively for complete write back, Capo reduces all peaks in the storage request load.

5.4 Trace Replay

To evaluate the benefits of deploying Capo in a real world setting, we replay the collected I/O traces (see Section 3) using different disk caching policies. The next sections describe our experimental setting, analyze our replayer fidelity, and present the replay results.

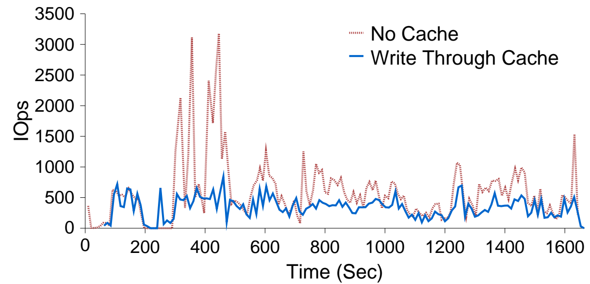


Figure 13: IOps per second for a workload of 11 Windows users on a XenCenter Cluster.

5.4.1 Experimental Setting

The test environment consists of four physical machines which serve as hosts for the virtual machines that replay requests from the recorded trace, and a filer to serve as a backend storage for these virtual machines' disks. The filer runs Linux's default kernel NFS server to host an XFS volume built on top of a RAID 0 consisting of six disks. The host machines run XenServer 5.6 and store their local caches in an ext3 volume on top of a RAID configuration similar to that of the filer. The machines are connected using a 1Gb Ethernet switch.

We replayed the workload of each desktop for which we had collected traces in a distinct virtual machine on one of the XenServer hosts. As it is impractical to replay the entire week's trace for each configuration, we choose to focus on the six peak regions identified in Section 3.

Entirely isolating our analysis to the peak regions would start each replay with an empty cache. Instead, we accurately recreated the state the cache would be in at the start of each region by priming it with the data from whole trace up to that point. This includes any write-back blocks that would have been pending. The write-back interval was set to ten minutes for the write-back and differential durability policies.

5.4.2 Replay Fidelity

Both the hosts and the storage server in our replay experiment are different from those in the original system from which we collected the traces. We satisfied ourselves that the replay is representative by measuring the observed load during a simple replay without any caching. Figure 12 plots the fourth selected time period's I/O operations per second as observed at a number of different points in the I/O stack of the experimental environment.

The *Trace* line represents the aggregate workload as observed in the original trace. The *Replay* line represents the rate at which the replayer issues I/O requests to the system as observed at the replay clients. These two lines

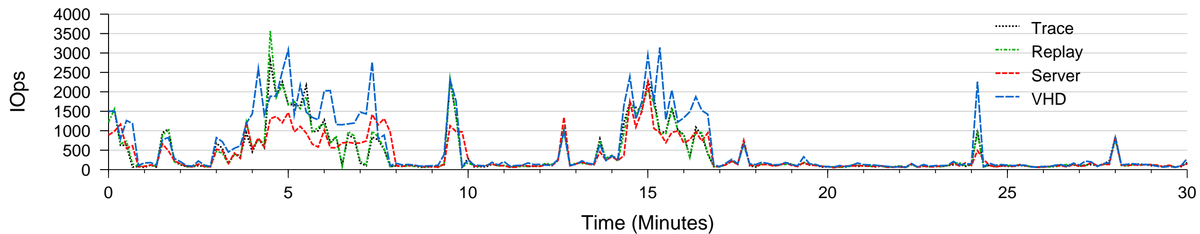


Figure 12: Replay fidelity and resulting load on the server.

Time Period	Peak IOps / Reduction in peak IOps compared to No Cache configuration							
	No Cache		Write Through		Write Back		Differential	
	Peak	Total	Peak	Total	Peak	Total	Peak	Total
1	2307 / 100%	262590 / 100%	893 / 38%	155757 / 59%	670 / 29%	67798 / 25%	712 / 30%	91877 / 34%
2	2516 / 100%	561894 / 100%	937 / 37%	319936 / 56%	671 / 26%	113184 / 20%	903 / 35%	161737 / 28%
3	1302 / 100%	143468 / 100%	876 / 67%	126049 / 87%	595 / 45%	43455 / 30%	802 / 61%	84044 / 58%
4	1887 / 100%	450914 / 100%	910 / 48%	334089 / 74%	595 / 31%	131064 / 29%	849 / 44%	271529 / 60%
5	1214 / 100%	159736 / 100%	890 / 73%	141656 / 88%	704 / 57%	45868 / 28%	841 / 69%	75500 / 47%
6	2185 / 100%	72082 / 100%	1155 / 52%	66668 / 92%	910 / 41%	29086 / 40%	1368 / 62%	42895 / 59%
avg	100%	100%	52.5%	76%	38.1%	28.6%	50.1%	47.6%

Table 3: Peak and Total IOps workload observed at the file server during the replay of time periods of interest under different caching policies. Each peak or total IOps value is followed by its ratio relative to its corresponding value observed with no cache deployed. The last row represents the average reduction of the metric across the six time periods of interest.

are almost indistinguishable in the figure which indicates that the timing of our replayer is accurate.

The *Server* line represents the load observed at the server. Notice that this load is lighter than the aggregate trace load, largely due to coalescing requests in the storage stack of the XenServer hosts. The *VHD* line represents the load observed at the server when image files are stored in the Microsoft VHD format. Notice that VHD adds significant overhead to the workload; most of this overhead is due to meta data management.

We draw two observations from this evaluation. First, our replay client is able to match the request issue rate of the original trace with high fidelity. Second, because of transformations that result from both the XenServer storage stack and the underlying VM image format, the load experienced at the storage target may be dramatically different from that measured at the client. In evaluating our cache under replay in the next subsection, we first replay with no caching involved to establish a baseline load at the filer, and then compare caching configurations to this baseline.

5.4.3 Replay Results

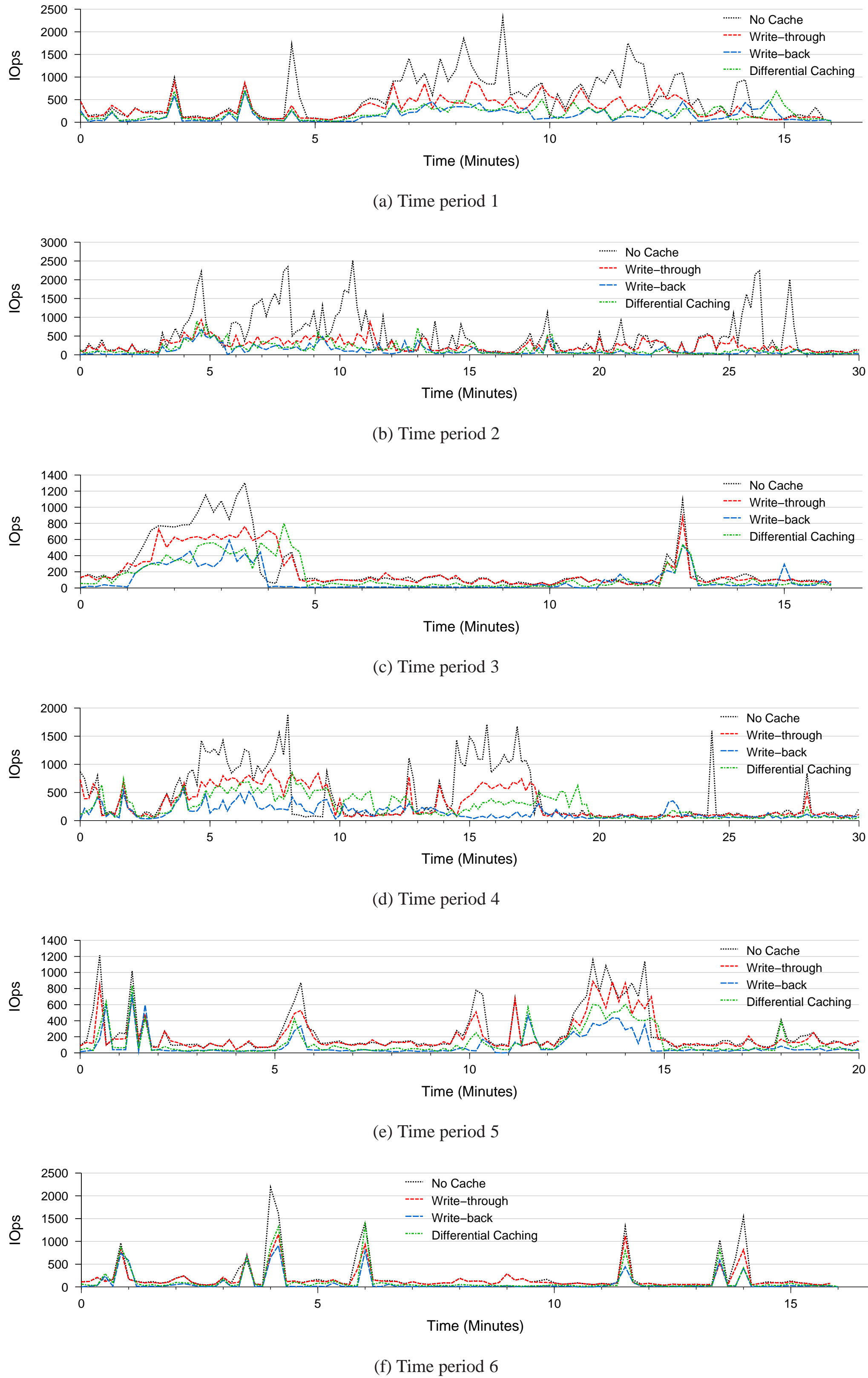
We replayed the 6 periods of intense workload identified in Figure 2 using four different cache configurations. These cache configurations are no cache, write-through, write-back and differential. Figure 14 plots the IOps observed at the server using each configuration. As expected, differential durability represents a compromise between the load reduction realized by write-back and completely protecting important user data.

Table 3 summarizes the peak and total I/O workload reductions for the time periods of interest. The write-back policy applied to the entire disk was the best in reducing I/O workload. On average it reduced the peak and total I/O workload to 38.1% and 28.6% of that without any caching in place. The differential durability policy goes further to protect user files, and still reduced the peak and total I/O workload down to 50.1% and 47.6% on average. Finally, as expected the write-through policy had the worst average peak and total workload reductions of 52.5% and 76%.

6 Related Work

The caching component of Capo is most closely related to the ITC [19], Andrew [10], and Coda [12] file systems which utilize the local disk as a cache for whole files retrieved from servers. The Cedar file system [21] allows users to share immutable files over the network; by only supporting immutable files Cedar eliminates the need for cache consistency management.

Unlike these distributed file system caches, Capo operates at the block level. Cache consistency management



(a) Time period 1

(b) Time period 2

(c) Time period 3

(d) Time period 4

(e) Time period 5

(f) Time period 6

Figure 14: IOps per second observed at the filer for replays of selected periods of interest under different cache configurations.

is simplified by the fact that each virtual disk has a single writer and copy-on-write is used to prevent updating shared data. As in Cedar, shared data is always immutable.

Fs-cache [11] and iCache [9] are systems that, like Capo, implement block-level caching for remote storage systems: a file system in the case of Fs-cache and an iSCSI target in the case of iCache. Capo extends the basic block caches of these systems using a host cache shared by all the VMs on a host, the multi-host prefetcher, and differential durability for files. All of these features are inspired by our target environment of supporting virtual desktops.

Capo's use of write-back caching reduces the demand placed on the central storage facility in a manner similar to that of Everest [15]. Where Everest replicates offloaded write requests to tolerate disk failures, Capo uses a technique similar to Snapmirror [16] to periodically push self-consistent updates across the network for data that is cached in write-back mode.

Other researchers have studied the performance of storage in virtualized environments. In particular, Gulati et al. [7] study the storage demands of enterprise applications in virtualized environments. In contrast, our study of virtual desktops provides insight into the unique characteristics of this emerging use of virtualization.

SnowFlock [13] provides a fork abstraction to instantaneously replicate stateful virtual machines to scale up computations in the cloud easily. Similar to our multi-host cache preloader, SnowFlock uses multicasting to replicate the persistent (disk) and non-persistent (memory) state of the cloned virtual machines.

Agrawal et al. [1] and Bolosky et al. [2] collect and analyze snapshots of Desktop machine's file system metadata over long periods of time. This kind of analysis restricts I/O workload analysis to mean estimates and doesn't capture the dynamic characteristics of Desktop I/O such as burstiness. In this work we focus on capturing detailed block level I/O operations to better understand the variation of Desktop I/O workloads in time.

Lithium [8] gives up centralization in favor of distribution to provide scalable storage for virtual machines. To improve availability of data, Lithium replicates disk updates to remote hosts either synchronously or lazily (eventual consistency). These two replication policies are synonymous to Capo's write-through and write-back caching policies. However, Lithium's treatment of replication consistency is more complicated due to its distributed nature.

7 Conclusion

Enterprise storage provides considerable benefit to virtual environments. However, for applications such as virtual desktops, which involve large numbers of nearly identical images running concurrently, a large portion of the request load placed on shared storage is unnecessary. After analyzing a one-week trace of a production VDI deployment, we presented Capo, a distributed and persistent cache which reduces the aggregate load placed on shared storage. Capo uses local disks on individual physical servers to cache image contents for the VMs being hosted. It includes mechanisms to share common cached base images across VMs, and to prefetch caches across physical hosts. In addition, Capo supports a configurable degree of differential durability, allowing administrators to relax the durability properties and the associated write load of less-important subsets of a VM's file system.

Acknowledgements

We thank our shepherd Greg Ganger, the anonymous reviewers, and Brendan Cully for their valuable feedback that helped improve this paper. We also thank the University of British Columbia's IT department, particularly Brent Dunington and Lois Cumming, for allowing us to collect I/O traces from one of their VDI deployments. We would also like to thank MITACS, NSERC, Citrix and IBM CAS who have provided support for the students involved in this work.

References

[1] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9, 2007.

[2] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43. ACM, 2000.

[3] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: A cache-based system management architecture. In *In Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI*, pages 259–272, 2005.

[4] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, page 8. USENIX Association, 2009.

[5] F. Dogar, A. Phanishayee, H. Pucha, O. Ruwase, and D. Andersen. Ditto - a system for opportunistic caching in multi-hop wireless mesh networks.

In *Proceedings of ACM MobiCom*, San Francisco, CA, Sept. 2008.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: a scalable secondary storage. In *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[7] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[8] J. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26. ACM, 2010.

[9] X. He. A caching strategy to improve iSCSI performance. *27th Annual IEEE Conference on Local Computer Networks, 2002. Proceedings. LCN 2002.*, pages 278–285, 2002.

[10] J. Howard, M. Kazar, S. Menees, and DA. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[11] D. Howells and R. Ltd. Fs-cache: A network filesystem caching facility. In *Proceedings of the Linux Symposium*, volume 1, 2006.

[12] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[13] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[14] Microsoft. Virtual hard disk image format specification, Feb. 2009. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

[15] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 15–28. USENIX Association, 2008.

[16] R. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 9. USENIX Association, 2002.

[17] C. Pettey and H. Stevens. Gartner says worldwide hosted virtual desktop market to surpass $65 billion in 2013, Mar. 2009. `http://www.gartner.com/it/page.jsp?id=920814`.

[18] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *In 3rd Symposium of Networked Systems Design and Implementation (NSDI*, pages 353–366, 2006.

[19] M. Satyanarayanan, J. H. Howard, D. a. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system. *ACM SIGOPS Operating Systems Review*, 19(5):35–50, Dec. 1985.

[20] E. Scholten. How to: Optimize guests for vmware view, July 2010. `http://www.vmguru.nl/wordpress/2010/07/how-to-optimize-guests-for-vmware-view`.

[21] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *SOSP*, pages 25–34, 1985.

[22] VMWare. Virtual machine disk format (vmdk), 2010. `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.

[23] W. Vogels. File system usage in windows nt 4.0. In *ACM Symposium on Operating System Principles*, pages 93–109. ACM, 1999.

[24] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2008.