University of California
Los Angeles

Hypervisor Side Cache for Virtual Desktop Infrastructure

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Sumedh Vivek Sakdeo

2012

Abstract of the Thesis

Hypervisor Side Cache for Virtual Desktop Infrastructure

by

Sumedh Vivek Sakdeo

Master of Science in Computer Science

University of California, Los Angeles, 2012

Professor Yuval Tamir, Chair

Virtual Desktop Infrastructure (VDI) deployments run large numbers of desktops in a virtualized environment to increase flexibility and address cost. One of the major challenges VDI faces today is the cost of high bandwidth interconnection networks to shared storage. VDI storage workloads have a number of unique characteristics which make them a target for optimization. For example, VDI workloads exhibit high amount of redundant data transfers (from shared OS images), highly bursty behavior (from daily work patterns), and a common storage format (virtual disks).

This thesis performs a detailed study of VDI workload and evaluates effectiveness of four hypervisor side optimization techniques. To eliminate network read requests and serve data from locally cached blocks, we evaluate two read caches, namely, location-addressed and content-addressed. We also compare these read cache with a simple mechanism which stores shared read-only virtual disks on hypervisor side local media. To eliminate transfer of redundant data that is written to the storage server, we evaluate the effectiveness of inline write deduplication. All the experiments are carried out in two setting, for full clone virtual desktops and linked clone virtual desktops.

A detailed trace-driven simulation study of the mechanisms with a realistic VDI workload shows up to 75% reduction in the total network I/O traffic. We propose some recommended setting for choosing the right optimizations, for example, for full clone virtual desktops

content-addressed cache outperforms location-addressed cache by 50%.

The thesis of Sumedh Vivek Sakdeo is approved.

Glenn Reinman

Songwu Lu

Yuval Tamir, Committee Chair

University of California, Los Angeles

2012

To my parents . . .

who always motivated me to . . .

keep learning

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Prof. Yuval Tamir for his continued interest in my thesis and timely help. He has been a constant source of inspiration and guidance, and this work would have been impossible without him. His ability to clearly identify the problem at a high level and then moving down the stack into minute details has proven to be an invaluable contribution to this work. His understanding of software systems and virtualization has helped me evolve a small idea into a big, useful system.

I am also grateful to Prof. Songwu Lu and Prof. Glenn Reinman for agreeing to be on my thesis committee. Their suggestions and detailed feedback helped me polish this work even more.

This work started as my internship project at Tintri, Inc. under Dr. Brandon Salmon. He has been instrumental from the initial days and has always provided me a platform to bounce my ideas. His feedback through every stage of the project and code reviews has helped me maintain high coding standard in developing this system. All the experiments were carried out on systems provided by Tintri for research use, and I am very thankful to each and every employee of this start-up. I thank Edward Lee, principal architect at Tintri, for his motivation towards solving challenges faced in virtual desktop infrastructure. I would also like to thank Josh Yuan for efficiently time-slicing my use of the hardware at Tintri.

I would again like to thank Prof. Yuval Tamir and Dr. Brandon Salmon for the numerous hour long discussions we had over last eight months.

Finally, none of this work would have been possible without support of my family and friends. My parents and my brother have always encouraged me and supported all my decisions. My friends at UCLA, especially my flat-mates have been an encyclopedia for technical discussions like algorithm designs, coding standards, system programming, etc. and of course, having fun.

CHAPTER 1

Introduction

Organizations have seen clear advantages of moving the computing power to high-end centralized servers. This evolution of thin-client or server-based computing replaces bulky physical desktop machines by cheap, energy efficient thin clients. In the past, centralized servers running multi-user operating system(OS) was one of the popular implementation of client-server model, providing remote access to the clients over the network. With the advent of virtualization, soon these servers were virtualized running multiple virtual machines(VMs) per physical server. Virtualization brings benefits like server consolidation, energy efficiency, centralized administration, and high data integrity.

Virtual desktop is a virtual machine which runs a desktop operating system and user applications. The practice of centralizing a farm of virtual desktops constitutes of a virtual desktop infrastructure(VDI). VDI deployments can be seen in call centers, universities, hospitals, banks, where thousand of users use thin clients to access their virtual desktops running on remote centralized servers. Gartner predicts upto 49 million virtual desktops being deployed by the end of 2013 [PS09]. As shown in figure 1.1, VDI deployment comprises of following components: multiple client machines, VDI management solution, centralized virtual desktops, hypervisor, shared storage, and interconnection network. Multiple client machines access centralized virtual desktops over remote desktop protocols. VDI management solutions help automate virtual desktop management by providing a centralized interface for configuring the lifecycle and client assignments for virtual desktops. Hypervisor is a thin layer of software which efficiently multiplexes the physical hardware resources between multiple virtual desktops. Typically, all the virtual desktop data is stored on a remote storage server which is shared between multiple hypervisors. High bandwidth interconnection net-

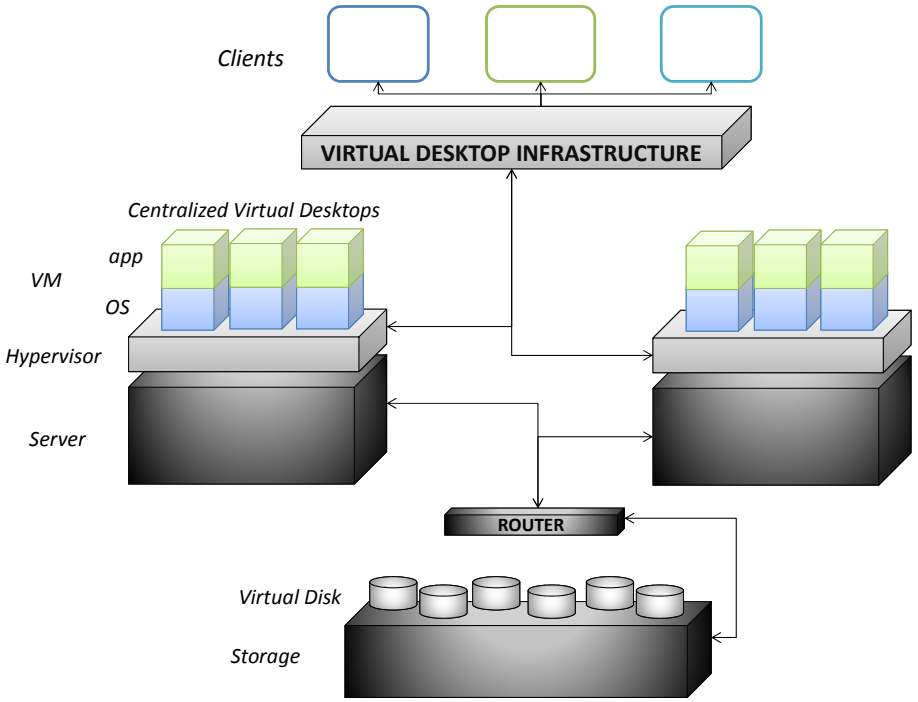


Figure 1.1: Virtual Desktop Infrastructure

works are used between hypervisor and shared storage servers, and hypervisor and multiple client machines.

Depending upon the organizational needs large subsets of virtual desktops in a VDI deployment are provisioned with similar operating system and applications. Each virtual desktop stores its data in virtual disk files on shared storage servers. A virtual disk is a special file which is mapped to a block device in the guest OS in VM. The use of shared storage as a backing store for virtual disk files help organizations address flexibility, manageability, scalability, and reliability issues. Two popular choices for shared storage adoption are: block level data storage (Storage Area Network - SAN, using Fibre Channel, iSCSI, or FCOE), and file level data storage (Network Attached Storage - NAS, using NFS, AFS, or CIFS). Block level data storage exposes a block interface to the client over the network, hence the client uses the same volume management and disk I/O facility as for local disk blocks. File level data storage exposes a file system interface to the client, and hence the client must implement a protocol (i.e. NFS, AFS, or CIFS) to access data over the network. VMware ESX server supports both block level data storage (using a proprietary virtual machine file

system, VMFS [Vag10]) and network-attached storage (using a NFS client implementation in the hypervisor [Man09]).

With sophisticated resource management techniques implemented in VMware [Wal02] and Xen [BDF03] hypervisors, number of virtual desktops provisioned per physical server has been gradually increasing. However, this has dramatically increased memory, storage, and network demands. To cope up with this increasing demand, administrators must continuously upgrade their storage and network infrastructure. Ever increasing reliability of distributed file systems like NFS, AFS, CIFS, and a lower cost-per-port for IP-based storage is making NAS a popular choice for small to medium businesses that are deploying virtualization. Although the file server implementations on the NAS appliances have evolved over last three decades, one of the major bottleneck that still remains is the capacity of network used for data transfer, i.e. network bandwidth.

Extensive research has been done to reduce the overall network bandwidth requirement. One of the most popular software optimization technique to address this cost issue is to aggressively cache RPC lookups, attributes, and read data, on either client side(NFS, AFS, or CIFS client) or on intermediary network routers [Tol03, MCM01, GNT07, VAB04, Bad98, DWA94, MAC08, SMW11]. Froese and Bunt[FB96] have investigated the effect of client caching on the workload presented to a file server in a distributed file system. They argue that with the increase in the size of client cache, the temporal locality of I/O access pattern may no longer be present at the server.

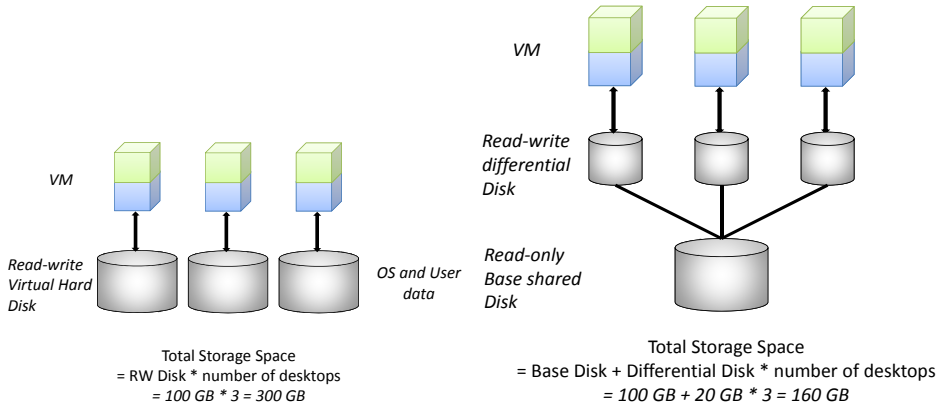
Most of the above client-side cache management studies have shown that, the nature of I/O requests to the file server exhibit properties of temporal and spatial locality. Typically, these I/O requests are generated by a variety of applications running in multiple client workstations accessing client specific data. A traditional cache which is addressed by block locations, and uses a replacement policy which favors recently used blocks, has worked well for the workloads studied above.

VDI workload is marked by large amount of redundant data transferred from/to desktop VMs, and a highly bursty behavior. A large subset of desktop VMs are copies provisioned

from a parent VM template. All these desktop VMs run similar OS and applications, thus resulting in a significant data commonality between multiple virtual desktop images. The I/O request patterns, therefore, exhibit a high degree of content locality i.e. access to the same content. Moreover, I/O traffic in VDI suffers from frequent bursty activities very common in typical VDI deployments like call centers, hospitals, universities. Some of these bursty activities are virtual desktop provisioning, boot storms, login storms, scheduled anti-virus scans, backups, etc. Virtual desktop provisioning is the process of creating new virtual desktop VMs cloned from a given parent template. Boot and login storms are high spikes in I/O traffic generated when number of desktop VMs are powered-on simultaneously. Scheduled anti-virus scans generate a high read traffic at periodic intervals. These bursty I/O activities also involve a lot of similar data contents being transferred over the network. All these characteristics of VDI workload makes it an interesting topic of study for analysis of client side caches. Hence, in this study we evaluate client side caches which are either temporal, spatial, or content locality-aware, for VDI workload.

VDI workload also exposes a high potential to eliminate transfer of data to be written, because it was previously written to the server by other or same virtual desktops [MCM01]. This technique is called write deduplication, and benefits from write activity to similar application and OS logs, downloading email attachments sent to a mailing list, Microsoft Office operations like sorting and saving column data, page files, internet explorer temporary files, etc. Therefore, we also make an effort towards evaluating benefits of this write optimization technique, for VDI workload.

Virtual desktops are classified into two types, mainly depending upon the way in which the virtual disks are organized. In the initial days, each newly created virtual desktop was a naive full copy of the parent virtual desktop template. Soon it was realized that this organization of virtual disk files resulted in the increased consumption of space from duplicate data across multiple virtual desktops. Hence, a new organization was proposed which stores the common data between multiple virtual desktops in a separate shared read-only virtual disk. Borrowing from VMware terminologies [Und], following are the two types of virtual desktops,



(a) Full Clone (b) Linked Clone

Figure 1.2: Types of Virtual Desktops

- Full clone virtual desktop: Every virtual desktop provisioned from the parent virtual machine template is a full copy of the parent virtual machine. This kind of virtual desktop deployment results is wastage of disk space because lot of read-only data is being duplicated for each virtual desktop. Figure 1.2a describes three fully cloned virtual desktops and their storage requirements.

- Linked clone virtual desktop: Every virtual desktop provisioned from the parent virtual machine shares the read-only part of the virtual desktop image with the parent and other cloned virtual desktops. A differential disk is provided for handling any future read-write operations. Although, this sharing conserves disk space, it results in performance degradation. Figure 1.2b describes three linked cloned virtual desktop and their storage requirements.

Although, linked clone virtual desktops are becoming popular in today's VDI deployments, full clone still prevail due to their simplicity, and high performance. We observed the caching behavior to be different in the two types, mainly because of the organization of virtual disks.

This thesis presents an analysis of the use of read caching and write deduplication to reduce the network and file server capacity required to support virtual desktops. This involves a client cache which resides on the hypervisor running virtual desktops. All the bookkeeping information to manage the cache is stored on the hypervisor.

This is a trace-driven cache simulation study, where the block-level I/Os generated by real desktop applications are traced and studied. Applications like Microsoft Office, Video Player, Archiving Software, PDF Reader, Web Browser, Antivirus, etc. running in Windows 7 virtual desktop VMs are used to simulate most common VDI deployments.

For both the full clone and linked clone virtual desktops, we analyze four mechanisms for reducing network and file server load:

- Location Addressed Read Cache: This cache serves data for read requests that hit in the cache, addressed by location. A location is defined by virtual disk file name used for I/O and block number computed from the read offset. This is a conventional block addressed cache often seen in processor caches. Capo[SMW11] has implemented this cache and seen significant reduction in load on storage for virtual desktop workload.

- Content Addressed Read Cache: *Content Fingerprint* is a unique SHA-1 digest of block data content. This cache is addressed by content fingerprint thus eliminating redundant copies of data in the cache and utilizing the cache space in an efficient way. This *Content Addressable Storage (CAS)* scheme has been widely explored in client-server distributed file systems [Tol03, MCM01, SS02]. We were motivated by this technique because of high degree of similarity between virtual desktop images.

- Local store of Linked Clone Read Only Disk: Linked cloned virtual desktops shares a read-only part of the virtual desktop image between multiple virtual desktops in a separate shared virtual disk. This optimization involves storing the shared virtual disk on local disk of the virtualized servers, thus eliminating any network read traffic to the shared disk.

- Fingerprint cache for Inline Write deduplication: This cache eliminates sending data over the network if the same data is found to be present on the remote network file server. We maintain a list of content fingerprint in our cache to determine if the data is already present on the remote server[MCM01]. This technique of eliminating writes of redundant data to the disk is called write deduplication.

As part of our analysis, we evaluate the impact of cache size and replacement policy. The evaluation metric used for our study is the traffic ratio (both, read and write traffic ratio), as described in chapter 5. Some of our observations show as high as 75% reduction in the total network traffic with the use of above mentioned optimization techniques. Our recommendations can help virtual infrasutructure or storage vendors develop caching modules which fits their customer's infrastructural demands.

Chapter 2 discusses the related work for this research. Chapter 3 describes hypervisor side caching. Chapter 4 describes the design and implementation of different caching techniques in details. Chapter 5 discusses the IO trace collection setup, VDI benchmarks and evaluation results. Chapter 6 discusses some plans on the future work. Chapter 7 summarizes the findings and concludes.

CHAPTER 2

Related Work

While VDI-specific storage optimization is relatively new in the research community, there is a wealth of related work on client-side caching techniques in distributed file systems over the last three decades.

The ITC distributed file system[SHN85], used local disk on client workstations to cache entire files with their status and custodianship information. The main design objectives behind introducing this file cache on client side was to improve performance, mobility and scalability. As client caches entires files, server is contacted only on file opens and closes, and not on individual reads and writes. Moreover, total network protocol overhead in transmitting a file is lower when it is sent *en masses* rather than in a series of responses and requests for individual pages. Finally, implemetation on file servers is simplified because now the request is at a whole file granularity. We leverage these benefits in one of our cache optimization techniques which stores the entire read-only shared virtual disk file on local disk of virtualized server.

Sprite network file system[NWO88] implemented an in-memory client side cache to improve performance on diskless client workstations. To avoid excessive usage of physical memory, this file cache negotiated with the virtual memory system and dynamically changed its size. This client file cache reduced the server load by 50% and network traffic by 75%.

One of the main design goals for each version of Andrew File System(AFS) was to increase scalability. AFS-1 cache files and location information as discussed in ITC distributed file system[SHN85]. AFS-2, AFS-3, and Coda expanded the scope of this caching to directories. AFS-2 performed approximately 30% and 46% better with cold cache and warm cache respectively as against NFS on 20 client instances running Andrew benchmark.

Dahlin[DWA94] performed a trace-driven simulation study of four client side caching techniques which cooperate in a distributed environment. The use of in-memory caches in this work resulted in 50% reduction of the number of disk accesses and improved file system read response time by as much as 73%.

Performance improvements to the NFS client implementation in the linux kernel was done by Badros[Bad98]. This paper described numerous improvements like asynchronous write buffering, caching of *lookup* RPC results, and local disk caching of RPC reads. This enhanced NFS client outperformed the standard NFS client implementation by a factor of 14 when reading cached files and almost by a factor of 100 for small writes for tests like andrew benchmarks, untar big package, repeated ls-ing, read big files, copy big files, small reads and writes.

A Low-bandwidth Network File System(LBFS)[MCM01] is a network file system which exploits similarities between files or versions of the same file to save network bandwidth. It eliminates transfer of data over the network when the same data is found to be present on the servers file system or the clients cache. LBFS ran the following benchmark workloads: Microsoft Word edits, gcc - recompilation of emacs, and changes to the perl 5.6.0 source tree, and compared their results with CIFS, NFS, AFS, Leases + Gzip. LBFS outperformed all other file systems in terms of reduction in network bandwidth reduction and round trip latency. LBFS work is suitable in VDI environment where their is a high degree of commonality between multiple virtual desktop images and divergence from the initial base image is not very high as observed by Capo[SMW11]. This technique of eliminating coarse-grained redundant data is called deduplication. This work motivated us to explore write deduplication as one of caching technique.

Learning from content addressable storage in Distributed Hash Tables, Casper [Tol03], maintains recipe which is a description of each constituent content addressable block belonging to the file. A Casper client opportunistically fetches cached blocks from nearby client providers to improve its performance when the connection to a file server traverse a low-bandwidth path. Casper uses a cooperative caching methodology where clients can provide data to other clients on the LAN. Casper maintains recipes which contain a mapping of file

location and fingerprint. Each I/O request for a file location uses an associated fingerprint to address into the cache directory in same or different client and fetches corresponding data. This principle of content addressable storage is very useful in eliminating read traffic from I/O request to similar data, very common in VDI workload. Casper evaluates the commonality among different releases of linux source tree, and Mozilla binary nightly releases, thus, providing a clear evidence of benefits of content addressable storage in distributed file systems. Capser uses benchmarks like mozilla install on clients where installer rpm is stored on remote server over WAN, trace from virtual machine migration running common desktop applications, and modified andrew benchmark. In virtual machine migration trace workload, CASPER gained 44% reduction in the execution time on a 1Mb/s link and cache with 33% of pre-populated data. Looking at the relevance of content addressable caching in VDI environments, we use a content-addressed read cache similar to Casper, and discuss its benefits over a traditional location-addressed cache.

Varanasi[VAB04] implemented a cache in an NFS proxy on an intermediary router between client and the server. This read-only cache is implemented on the proxy which stores recently accessed file data packets. This cache observed as high as 71% improvement in the performance as compared to typical NFS configuration at low bandwidth network settings.

Nache[GNT07] is a fairly recent working which implements a caching proxy for NFSv4 protocol in linux 2.6 kernel. Nache demonstrated performance benfits using Filebench benchmark and observed reduction in the number of NFS operations seen at the server by 10-50%.

Parallax[MAC08] employed a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switches are relocated into a federation of storage VMs, sharing the same physical hosts as the VMs that they serve. Parallax implements these storage features like low-overhead snapshot of virtual disks, and persistent caching on local media, in an isolated driver domain VM in Xen. Although implementation of local cache was still in early phases, Parallax observed an aggregate increase in the throughput because each local disk was accessed without interference from competing clients. Unlike parallax, which focuses on general-purpose VMs, we primarily focus on the VDI environments.

Capo[SMW11], is the most recent study on optimizing storage for virtual desktops. 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 VMs file system into regions of varying write-back frequencies. Unlike Capo, one of the limitation of our work is using IO traces collected from a synthetic VDI workload generator and duration of trace collection. However, we argue that the randomization introduced in execution of user operations by VDI workload generator used brings it very close to real workload, and one day of trace collected should suffice our evaluation. On the other hand, the scale of active virtual desktops being traced in our work is more than Capo. We also study the behavior of four different cache algorithms on two different types of virtual desktops, namely, full cloned and linked cloned virtual desktop.

Capo starts with an analysis of a weeklong trace collected from a real production VDI deployment in an executive and administrative office. Their trace collection module resides in the guest operating system, and hence they can record trace entries at desktop file system, block and file level. They use the trace analysis results to design a distributed persistent cache that aims to reduce aggregate load on shared storage in virtual desktop environments. The multi-host cache preload helps clients broadcast data fetched from the server to multiple host, hence eliminating reads to similar locations by multiple hosts. We plan to extend our current work to similar kind of cooperative caching in future. Capo also benefits from differential durability by using write-through, write-back and no-write-back write policies depending upon the file being written into. For example, all user and system data is critical and hence should be write-through, logs can tolerate write-back policy and temporary files do not need any write-back. Although, this concept dramatically improves write I/O traffic, it would require modification to the guest operating system for the caching module to detemine the file information.

Compared to Capo, our work explores various combinations of content addressable read, write deduplication, shared disk caching, and location-addressed caching. In addition, our trace represents workload from a wider set of applications being simulated and very high write IO traffic workload generated during virtual desktop provisioning.

Deduplication is the process of eliminating duplicate data blocks thus resulting an efficient usage of underlying storage space. As deduplication was CPU and memory intensive task, it was generally used in latency insensitive use cases, for example, archivals or backups. However, potential benefits gained from space reduction has led to an increased interest in use of deduplication technology on primary workload. iDedup[SBG12] is an inline deduplication solution, for primary workload on the actual I/O path. iDedup achieves 60-70% of the maximum deduplication with less than a 5% CPU overhead and a 2-4% latency impact. Two of our cache optimization techniques namely, content addressable read cache and write deduplication are based on this inline deduplication technology.

Hypervisor side caching has been widely explored in industry as a possible storage optimization technique. Few companies successfully playing these optimization strategies are, virtualization vendors like VMware, Citrix, enterprise storage vendors like Fusion-IO, EMC, Dell, Nutanix and VDI software solution vendors like Atlantis Computing. Fusion-IO's uses a fast SSD device called ioMemory in the hypervisor for storing the virtual desktop images locally FusionIO. Atlantis computing implements software tricks such as inline deduplication, sequentializing I/O and caching, on the I/O path from VM to shared storage AtlantisComputing. Citrix XenServer IntelliCache performs local caching of disk blocks and is based on Capo[SMW11].

CHAPTER 3

The Characteristics of I/O Workload Generated in VDI Environments

This chapter explains the characteristics of the I/O workload generated by VDI environments. These characteristics motivate the use of read caching and write deduplication, introduced in chapter 1, to reduce network and file server load.

3.1 Cloned Virtual Desktop Images

Virtual desktops are copies provisioned from a master virtual desktop template image. At the birth all virtual desktop images are exactly identical, however, with age each virtual desktop image diverges from their master and sibling virtual desktops. However, this divergence rate is low in VDI envrionment as observed by Capo[SMW11]. Hence, this high degree of similarity between multiple virtual desktop image, results in a I/O traffic which is flooded with redundant data being transferred over network. Hypervisor side read caches can eliminate any read I/O to the data content which was previously accessed by same or any other virtual desktop, as discussed in section 4.2.1, section 4.2.2 and section 4.2.3. Applications running in multiple virtual desktops also write a lot of data which was previously written to the storage server, for example, application logs, page files, e-mail attachments, application specific files, temporary internet explorer files, etc. The technique of using a fingerprint cache to eliminate such redundant writes is called write deduplication, and is discussed in section 4.2.4.

3.2 Virtual Desktop Classification

Virtual desktops are classified on the basis of how a new copy/clone of a virtual desktop is being created. Borrowing from VMware terminology, we call the two types of virtual desktops, full-cloned virtual desktop and linked-cloned virtual desktop. As discussed in chapter 1, drawbacks of full clone is high storage space utilization with no sharing of common data between nearly identical virtual desktops. However, in terms of performance full clones are better than linked clone virtual desktops because of an extra level of indirection in linked clones induced by use of a read-write differential disk on the top of shared read-only disk. I/Os are issued to the differential disk first, and if the I/O offset misses in the differential disk an I/O is re-issued to read-only base disk. Hence, a single I/O request can turn into multiple I/O requests increasing the latency for I/O request.

With full clones, the same information (for example, program executables) is read from the server by multiple virtual desktops. With location-addressed caches, each virtual desktop has to obtain its own copy of the data from the server. However, with content-addressed caches, multiple virtual desktops accessing identical disk blocks can obtain copy of the data fetched only once from the server. Thus, with full clones, location-addressed caches is not expected to be as effective as content-addressed caches.

On the other hand, linked clones use a shared virtual disk between multiple virtual desktops, containing OS and application data. Unlike full clones, with location-addressed caches, only the virtual desktop accessing the shared disk data block for the first time has to obtain copy of data from the server. As these shared disks are a target of high read activity, with linked clones, location-addressed caches can be expected to be as effective as content-addressed caches. The mechanism of storing the entire read-only shared disk on the local media on hypervisor side is a popular optimization technique, and is discussed in section 4.2.3.

Both full clones and linked clones are currently in use. As discussed above, these two types of virtual desktop generate different loads on the file server and may benefits differently from different mechanisms to reduce network and file server load. This motivates the evaluation in

Chapter 5, that includes the various configurations of network and file server load reduction mechanisms with the two different types of virtual desktops.

3.3 Bursty Workload Pattern

In a scenario like call center, where all the employees arrive to work between 8am and 9am on weekdays and power-on their virtual desktops, a lot of I/O traffic is generated. This burtsy I/O traffic is often seen in virtual desktop workload, and is called boot storm. Boot storm generates accesses to system and application files used during boot-up of any OS, thus resulting in access to similar content from multiple virtual desktops. Few other highly bursty I/O patterns are virtual desktop provisioning, antivirus scans, login-storms, and backups. These activity can lead to severe network congestion and significantly increased I/O latencies. In such circumstances, hypervisor side read cache can serve data locally and solve these problems.

CHAPTER 4

Design and Implementation

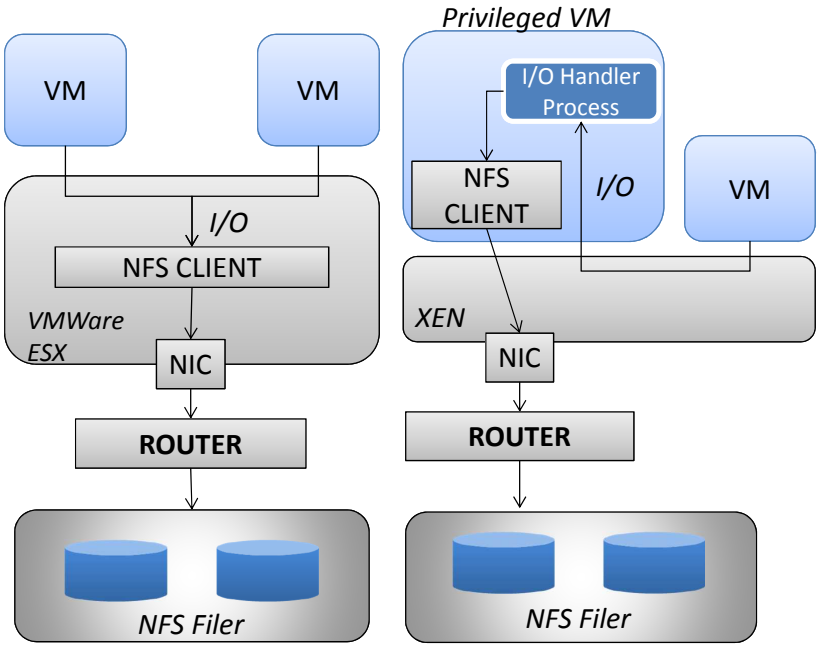
This chapter begins with a brief architectural overview of hypervisor side cache. Later in section 4.2, we provide a detailed explanation of the design and implementation of different hypervisor side read caching and write deduplication techniques, as introduced in chapter 1.

The ultimate goal of such caching is to reduce the overall network traffic between hypervisors and shared storage, here NFS server. This would help VDI deployments overcome the network bandwidth bottlenecks and increase the number of virtual desktop provisioned per physical server. Use of such caching on hypervisor side could help gain benefits of a costly high-bandwidth network while still using a low-bandwidth network between the client (virtualized server) and the NFS server. Our goal is to optimize access to NFS storage targeting common bottlenecks in VDI environments, while keeping up the traditional file system semantics. We do not compromise any consistency guarantees in a network file system to improve performance.

4.1 Architecture

Our study uses a NAS appliance/NFS filer to host virtual disks. A NFS client implementation on the hypervisor side is used to access remote VM data. In VMware ESX, NFS client implementation is part of VMware's proprietary hypervisor as shown in figure 4.1a. Xen uses the linux kernel NFS client implementation in the privileged domain *dom0* as shown in figure 4.1b. The hypervisor side read cache or write deduplication optimization can be sliced into the I/O stack at different levels as described below,

- Modified NFS Client: This approach involves enhancing NFS client implementation



(a) VMware NFS (b) Xen NFS

Figure 4.1: NFS I/O path in VMware and Xen

to leverage an on-disk cache. Such a modified NFS client would reduce the number of remote procedure calls thus reducing overall network traffic. This approach can use the consistency guarantees already implemented in NFS Client thus making the implementation less complicated. Badros [Bad98] modified the linux NFS client implementation to reduce the number of remote procedure calls through aggressive caching of lookups, attributes, and data pages.

- Proxy-based: This approach involves use of a proxy between NFS client and NFS server. Such a proxy can be implemented in a modular router or on the VM I/O path in the hypervisor. Although, this approach is more modular than modifying a NFS client, it incurs an extra overhead of parsing the network packets and understanding the RPC requests. Varanasi [VAB04] used *Click modular* router to implement a virtualized NFS proxy to implement functionalities like querying of file system meta-data, caching file system data, and encrypting file data.

- Filesystem in userspace (FUSE): This approach involves use of a userspace file sys-

tem process intercepting all the disk I/O requests to a mounted VM disk folder. This userspace program implements the caching functionality even before I/O request reaches NFS client. In Xen, this FUSE program can run in the privileged domain, which is responsible for handling all the VM I/O traffic. However, use of FUSE incurs potential performance overheads because of additional user to kernel boundary crossings.

4.2 Read Caching and Write Deduplication

The rest of this chapter describes the design and implementation of hypervisor side read cache and write deduplication optimization technique in details.

4.2.1 Location Addressed Read Cache

As discussed in chapter 1, a location-addressed read cache maintains a cache directory addressed by location, and an on-disk flat-file to store the cached data. Each location is a combination of virtual disk file identifier and block number. The block number is computed as below,

$$\frac{Offset}{BlockSize} \tag{4.1}$$

where, offset is the file I/O offset, and block size is the granularity of mapping and the granularity of transfer used by the cache. The cache directory maintains a tuple of location and indexing structure. The indexing structure is used to index into the flat-file. The flat-file is a regular cache data file on a file system local to hypervisor. For every cache hit by a location key in the cache directory, data is served from the cache file to the requesting virtual desktop. Otherwise, RPC request is issued to the NFS server to fetch data over the network, location is inserted into the cache directory, and fetched data is added to cache data file. Figure 4.2 describes the components of a location-addressed read cache.

The location-addressed read cache is fully associative to avoid any collision misses or interference misses. These misses are caused due to block placement strategies in set asso-

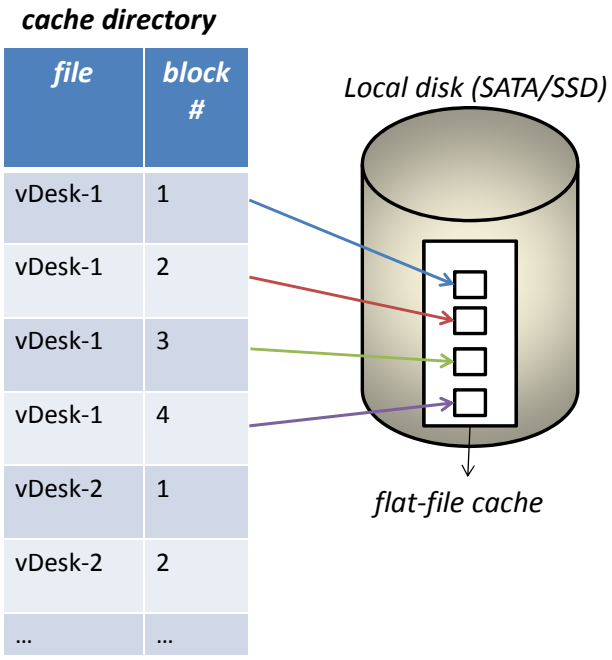


Figure 4.2: Location Addressed Read Cache

ciative or direct mapped caches. This cache implements a write-around and write-back write policy. The use of write-allocate policy in this cache led to approximately 30% decrease in the overall cache hit ratio, this is because the blocks that were written into the cache were less frequently read back by the virtual desktop. Although, the latency of write operation is significantly reduced with a write-back policy, if the client crashes before writing the data to the server, the file system might reach an inconsistent state.

The initial design decision for the cache directory implementation was to use a hash table for faster lookups (i.e. O(1) amortized lookup complexity). A properly designed hashing technique is generally more efficient than any other technique for associative lookup. However, there is a non-zero probability of poor performance in the worst cases (i.e. O(N) worst case time complexity for lookups).

The red-black tree is a balanced binary search tree, which guarantees the lookup time complexity of O(logN) in the best, average and worst case. This data structure is simple to implement unlike hash tables, which involve design of complex hashing functions. However, the memory overhead to maintain a red-black tree structure is more than a hash table

```
// Each node consumes 23 bytes for Random, and 31 bytes for FIFO and LRU
struct node {
      string location_; // 10 bytes
      char color_; // 1 byte
      struct node *parent_, *left_, *right_; // 4 bytes * 3

     // prev_ and next_ pointers are only used for FIFO and LRU
     struct node *prev_, *next_; // 4 bytes * 2
};

// statically allocated array of tree nodes
struct node array [NUM_OF_ENTRIES];
```

Figure 4.3: Data Structures used in Location Addressed Read Cache

structure.

The cache directory is implemented as one red-black tree structure shared across all the virtual desktops running on a hypervisor. These red-black trees are balanced binary search trees ordered alphabetically by key. Every cache directory tree node structure contains the location string used as a lookup key. This unique location string is a concatenation of a unique file identifier, a dash separator (-), and a block number. For example, if file identifier and block number for a location is 151 and 1400 respectively, then the location key string is '151-1400'.

Figure 4.3 shows the structure for a red-black tree node used to implement the cache directory. All the nodes in the red-black tree are stored in a statically allocated array. The size of this array is equal to the number of block frames in the cache data file. The cache data file maintains a pool of free block numbers, which are used whenever a new data block is to be written into the cache data file. Every read I/O issued to the cache, first looks up the disk block location key in the cache directory. If this location key is present in the cache directory, the cache block frame number, which is same as the index of the node in the statically allocated array, represents the block number to be read from the cache data file. If the location key is not present in the cache directory, a new location key is written to one of the free nodes, and data fetched over the network is written to the corresponding

block frame in the cache data file.

The cache directory must also implement a replacement policy for eviction of location keys when the cache directory is full. We study three replacement policies, namely, Random, FIFO, and LRU. A detailed explanation of the implementation of Random, FIFO and LRU replacement policies is given below,

- Random: This is the simplest replacement policy which selects a random tree node for eviction from the cache directory tree. In this eviction algorithm, we generate a pseudo random number to index into the statically allocated array of nodes, shown in figure 4.3. The victim node chosen for eviction is the indexed node of the array, i.e. the node which was randomly chosen.

- FIFO: In addition to using a balanced tree for faster lookups, we maintain a prev_ and next_ node pointer to construct a doubly linked list of all the tree nodes in the cache directory. For FIFO replacement policy, every newly inserted tree node is added to the end of this doubly linked list. The victim node chosen for eviction is the head node of this doubly linked list, i.e. the node which was inserted first is evicted first.

- LRU: Similar to FIFO, we maintain a doubly linked list of all the tree nodes in the cache directory. In addition to appending every newly created tree node to the doubly linked list, every recently looked up tree node is moved to the end of doubly linked list. The victim node chosen for eviction is the head node of this doubly linked list, i.e. the node which was least recently used.

All of the above replacement policies have O(1) time complexity for both determining the victim node, and removing the node from the tree. The time complexity for performing an insert or a lookup operation for a key in the cache directory tree is O(logN).

Table 4.1 shows approximate amount of main memory consumed by a full cache directory using location addressing, for different cache sizes and replacement policies. The total memory consumed by the entire cache directory is a product of, the number of entries in

Cache Size	Replacement Policy	Memory consumed
4GB	Random	184
8GB	Random	368
16GB	Random	736
4GB	FIFO	248
8GB	FIFO	496
16GB	FIFO	992
4GB	LRU	248
8GB	LRU	496
16GB	LRU	992

Table 4.1: Memory consumed by cache directory structure in MB - Location Addressed Cache

cache directory and the size of each tree node structure. As with every cache, the number of directory entries is equal to the number of block frames in the cache data file.

Each replacement policy requires different amount of memory allocations, for example, LRU needs more memory than Random to maintain directory entries which are least recently used. As shown in table 4.1, Random replacement policy has a lower memory overhead than FIFO and LRU. As shown in figure 4.3, the tree node structure consumes 23 bytes for Random replacement policy and 31 bytes for LRU or FIFO. As discussed above, the location key string (location_) is a concatenation of file identifier, dash separator, and block number.

As discussed above, the cache directory structure is maintained in main memory for faster access. As use of volatile memory might result in loss of data in the cache directory across reboots/crashes, use of phase change memory becomes a useful alternative. The cache data file is maintained on non-volatile storage local to the hypervisor. The storage devices may be a conventional hard disk drives, or, for higher performance, solid state drives (SSDs).

We expect a location-addressed cache to perform well in case of linked clone virtual desktop deployment. This is because, linked clones use a shared read-only virtual disk

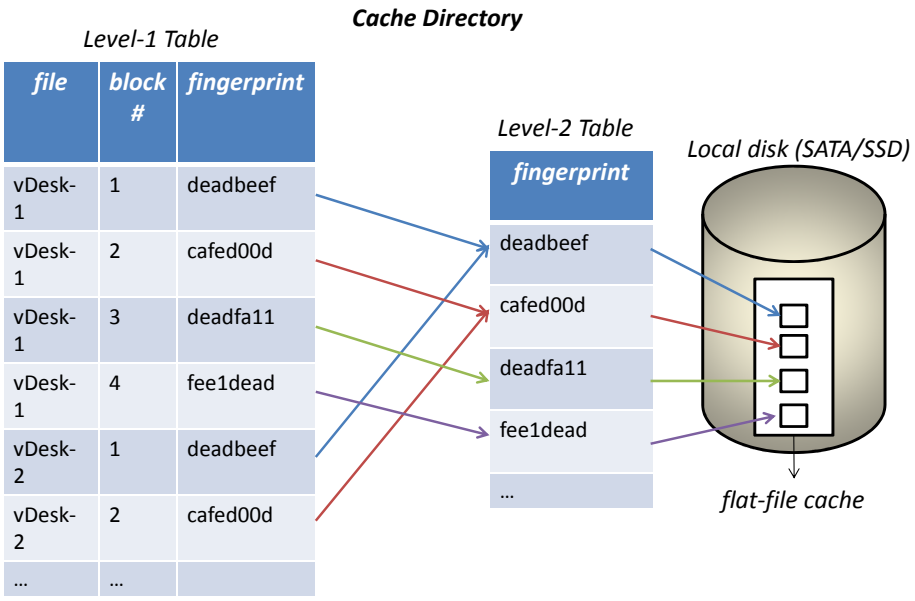


Figure 4.4: Content Addressed Read Cache

between multiple virtual desktops. The shared virtual disk contains OS, application and user data which has a high read access. This results in repeated I/O requests to same blocks of read-only shared disk from multiple virtual dektops sharing the cache. This cache also benefited full clone virtual desktop deployments to some extent.

4.2.2 Content Addressed Read Cache

As mentioned in section 4.2.1, a location-addressed cache can be useful when the I/O traffic is flooded with frequent access to same disk block locations. As discussed in chapter 1, because multiple virtual desktop run similar operating systems and applications, VDI workload is flooded with frequent accesses to identical disk blocks. A content-addressed read cache leverges this high degree of content similarity between virtual desktops and eliminates redundant copies of identical disk blocks cached locally. This Content Addressable Storage(CAS) scheme has been widely explored in distributed hash tables and distributed file system [Tol03, MCM01].

As shown in figure 4.4, a content-addressed read cache maintains a two level cache directory table structure. Similar to a location-addressed read cache, it maintains a cache data

file to store cached data. The level-1 table of the cache directory maintains a key-value pair of disk block locations mapped to disk block's content fingerprint. Whereas, the level-2 table of the cache directory maintains a set of all the content fingerprints. For the first I/O to any disk block location, the client cache is not aware of the disk block's content fingerprint and has to fetch the data from the server. This fetched data is used in fingerprint computation, and a mapping from location to fingerprint is added to the level-1 table for future accesses to the same location. The elimination of redundant copies of identical disk blocks cached locally, takes place in level-2 table. This level-2 table eliminates addition of new data to the cache data file, whose fingerprint matches any of the content fingerprints already present in the level-2 table. Hence, the number of entries in the level-2 table is determined by the number of block frames the cache data file can hold. The number of entries in level-1 tree is greater than number of entries in level-2 tree. This is because multiple disk block locations in level-1 tree can point to a single fingerprint entry in the level-2 tree.

Every I/O request issued to the cache will first use the location key (file identifier and block number) to lookup in the level-1 table. If the location key is present in the level-1 table, the corresponding fingerprint value is retrieved. This fingerprint value is then used as a lookup key in the level-2 table. If the fingerprint is present in the level-2 table, the corresponding cache block frame number is used to read data from cache data file.

A miss in the level-1 table results in following sequence of steps,

- fetch the data over the network,
- insert the location-fingerprint mapping in level-1 table,
- if the fingerprint is not already present in level-2 table, write the data to cache data file, add the fingerprint and associated indexing structure to the level-2 table. However, if the fingerprint is already present in the level-2 table, we do not write a redundant identical block to the cache data file or add fingerprint to the level-2 table.

Such an insertion of data which eliminates redundant copies of identical blocks written to the cache data file, results in efficient utilization of the on-disk cache space usage.

```
 string fingerprint_; //  20 bytes

// Level-1 Table: Tree Node Structure
// Each node consumes 27 bytes
struct level1Node {
       string location_;  // 10 bytes
       char *fingerprint_; // 4 bytes pointer to a 20 bytes fingerprint string
       char color_;  // 1 byte
       struct level1Node *parent_, *left_, *right_; // 4 bytes * 3
};
struct level1Node level1Array [NUM_LEV1_ENTRIES];

// Level-2 Table: Tree Node Structure
// Each node consumes 17 bytes for Random, and 25 bytes for FIFO and LRU
struct level2Node {
       char *fingerprint_; // 4 bytes pointer to a 20 bytes fingerprint string
       char color_;  // 1 byte
       struct node *parent_, *left_, *right_; // 4 bytes * 3

      // prev_ and next_ pointers are only used for FIFO and LRU
      struct node *prev_, *next_; // 4 bytes * 2
};

// statically allocated array of level-2 tree nodes
struct level2Node level2Array [NUM_LEV2_ENTRIES];
```

Figure 4.5: Data Structures used in Content Addressed Read Cache

As the size of on-disk cache data file is fixed, there is a need to evict blocks when all the blocks in the file are used. This eviction begins with removing a victim node in level-2 table, and adding the evicted block to the pool of free blocks. Hence, if an I/O location hits in the level-1 table, and the corresponding fingerprint is not present in the level-2 table, it is a cache miss and requires fetching data over the network. The data fetched is then written to the cache data file, and the fingerprint computed is updated in level-1 and level-2 tables.

Similar to location-addressed cache, a write-back, write-around policy is implemented in the content-addressed cache. Every write I/O location that hits in the level-1 table, the corresponding new fingerprint value is updated in level-1 table, data is added to the cache data file, and new fingerprint is added to level-2 table, if not already present.

Both the level-1 and level-2 cache directory tables are implemented as separate red-black trees, shared across all the virtual desktops running on a hypervisor. Figure 4.5 shows the tree node datastructure for both level-1 and level-2 trees. Every tree node of level-1 tree structure contains the location string used as a key. The location string is a concatenation of a unique file identifier, a dash separator (-), and a block number. In addition, this level-1 tree node also maintains a value field used to store the content fingerprint for the corresponding disk block. The content fingerprint is a SHA-1 hash of the contents of the disk block. Every tree node of level-2 tree stucture contains the content fingerprint used as a key. Similar to the location-addressed read cache, we maintain a statically allocated array for level-1 and level-2 tree nodes. All the fingerprint strings cached are dynamically allocated, and consume 20 bytes. Both the tree node structure maintain a character pointer to a single fingerprint string, in order to avoid maintaining duplicate copies of same fingerprint string in the tree node structure. This fingerprint string is dynamically allocated when a new fingerprint is encountered for a disk block location. The character pointers in the level-1 and level-2 tree structures point to this fingerprint. When all the level-1 entries and the level-2 entry are evicted, the fingerprint storage is reclaimed.

The number of entries in the level-1 tree is dependent upon the physical memory size available on the hypervisor. Generally, the number of entries in the level-1 tree is 5 to 10 times the number of entries in the level-2 tree. When the level-1 tree is full, a Random replacement policy is used to evict node from the level-1 tree. This replacement policy is used to minimize overhead. There is no connection between level-1 table evictions and level-2 table evictions. The implementation of this eviction algorithm is similar to the implementation described for random replacement policy, in location-addressed read cache. The random number generated is indexed in the level1Array, shown in figure 4.5, to chose a victim node. The implementation of all the replacement policies in level-2 table is similar to their implementation discussed for location-addressed read cache.

Table 4.2 shows approximate amount of main memory consumed by a full cache directory using content addressing, for different cache sizes and replacement policies. The total memory consumed by level-1 table is a product of number of entries and size of each level-1

Cache Size	Replacement Policy	Memory consumed
4GB	Random	1376
8GB	Random	2752
16GB	Random	5504
4GB	FIFO	1440
8GB	FIFO	2882
16GB	FIFO	5760
4GB	LRU	1440
8GB	LRU	2882
16GB	LRU	5760

Table 4.2: Memory consumed by cache directory structure in MB - Content Addressed Cache

tree node structure. The total memory consumed by level-2 table is a product of number of entries and size of each level-2 tree node structure. For the calculations in table 4.2, the number of entries in level-1 table was five times the number of entries in the level-2 table.

4.2.3 Local Store of Linked Clone Read-only Disk

As described in figure 1.2b, multiple linked clones share a read-only base image virtual disk to store OS data, application installations/data, and user data. This results in a significant portion of data read to be served from the shared base image. One of our VDI deployment used is chapter 5 showed approximately 50% of read request hitting the shared base image.

As described in chapter 1, this technique stores the shared read-only virtual disk on the local hard disk on hypervisor-side thus eliminating a significant chunk of read traffic. All the read traffic to differential disks can still be served from a location-addressed cache or a content-addressed cache. This simple technique of storing shared disks on fast SSD local media on hypervsiors has been exploited by companies like Fusion-IO, Nutanix, etc. Any patches that are applied to the base image would need an invalidation of the current local copy and replacement with a new copy.

4.2.4 Fingerprint Cache for Inline Write Deduplication

Similar operating systems and applications lead to an I/O write workload which involves writing same data again and again by multiple virtual desktops. Some examples are application logs, downloading email attachment sent to a mailing list, microsoft office application like excel performing sort operation on columns and saving it, multiple virtual desktop users browsing same web pages writes similar temporary internet cache files, writing to swap files, and many more. Hence, write IO workload is flooded with a lot of redundant data which was previously written to the server by some other or same virtual desktop.

As discussed in chapter 1, using a fingerprint cache for inline write deduplication is an optimization technique to eliminate transfer of data to be written over the network, if the same data was previously written to the storage server. We were motivated to explore this deduplication technique from our observation of very high write deduplication factor implemented for SSD disks on Tintri NFS server, running a real virtual desktop deployment. Tintri optimizes the use of fast but expensive flash/SSD drives by implementing write deduplication in flash, i.e. eliminating redundant copies of the data. This technique has been vastly researched in Venti [SS02], LBFS [MCM01]. Figure 4.6 plots a graph of write dedup ratio against time from a real VDI deployment in a hospital hosting approximately 300 virtual desktops on Tintri NFS server. The dedup ratio is defined as,

$$\frac{Bytes\ in\ Flash\ without\ dedup}{Bytes\ in\ Flash\ with\ dedup} \tag{4.2}$$

As shown in the figure 4.6, the dedup ratio for VDI write I/O workload is as high as 3x. A similar high dedup ratio of upto 3.5x was observed in a university VDI deployment hosting approximately 140 virtual desktops.

Write deduplication avoids sending the same data over the network when it can determine that the data to be written already exists on the server. Ideally, this technique would require an up-to-date list of fingerprints of all the blocks that are currently on the storage server. However, maintaining such a complete up-to-date list on the machine running the hypervisor is not practical because of storage limitations, increased access latency if the list is very large, and the block might get changed on the server thus making the list on the hypervisor side

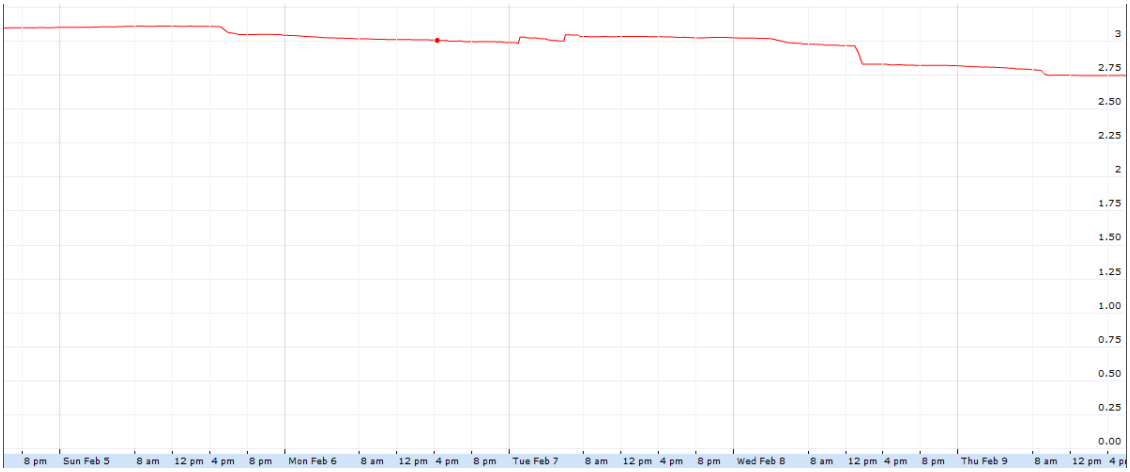


Figure 4.6: VDI Write Dedup Ratio. In this graph, the x-axis shows the date, day and time, whereas y-axis shows the dedup ratio.

out-of-date. Thus, what is maintained on the hypervisor side is a partial, potentially out-of-date, list of the fingerprints of blocks that are currently on the storage server. This list is managed in a similar way to a cache. This list is called a *Block Fingerprint Table* (BFT).

The block fingerprint table is maintained as a red-black tree where every tree node uses content fingerprint as a key. Figure 4.7 shows the tree node structure used by block fingerprint table. The implementation of replacement policy for a BFT is similar to the implementation discussed for cache directory implementation of location-addressed read cache. The correct functionality of write requests is still maintained if the BFT entries are simply invalidated. However, the eviction of fingerprints based on replacement policy, will be smarter, and shall improve the overall hit ratio in the BFT. For all the evicted / invalidated fingerprints, write request is required to send block data to the server.

The server also maintains a key-value pair of content fingerprint and corresponding server side disk block location (virtual disk file identifier and block number). The server uses this mapping table when it receives a write request to a disk block, with content fingerprint information. This table is maintained as a red-black tree, where every tree node structure uses the content fingerprint as a key, as shown in figure 4.8. Every tree node structure also maintains the corresponding disk block location, where the actual data can be found. This table is managed as a cache and uses LRU as a replacement policy.

```
// Each node consumes 33 bytes for Random, and 41 bytes for FIFO and LRU
struct node {
      string fingerprint_; // 20bytes
      char color_; // 1 byte
      struct node *parent_, *left_, *right_; // 4 bytes * 3

     // prev_ and next_ pointers are only used for FIFO and LRU
     struct node *prev_, *next_; // 4 bytes * 2
};

// statically allocated array of tree nodes
struct node array [NUM_OF_ENTRIES];
```

Figure 4.7: Data Structures used in Fingerprint Cache for Inline Write Deduplication

```
// Server side mapping table
// Each node consumed 43 bytes
struct node {
      string fingerprint_; // 20bytes
      string location_; // 10 bytes
      char color_; // 1 byte
      struct node *parent_, *left_, *right_; // 4 bytes * 3

     // prev_ and next_ pointers are only used for FIFO and LRU
     struct node *prev_, *next_; // 4 bytes * 2

};
```

Figure 4.8: Data Structures used in Server Side mapping table

For every write I/O issued by a virtual desktop, the content fingerprint is computed on hypervisor side and looked up in the BFT. If the fingerprint is present in the BFT, only the disk block location and fingerprint is transferred to the server. The server handles such write requests by looking up the fingerprint in the server-side mapping table, and retrieving the corresponding disk block location where the same data was previously written. This follows a server-side copy of the data from source disk block location given by the mapping table entry to the destination disk block given by the write request. However, if the server finds that it no longer has the data for a given fingerprint, it can fail the write operation, and force the client to re-issue the write with the full data block. The server would then write data to the newly allocated block, and add a fingerprint to disk block location mapping to the server mapping table.

If the fingerprint is not present in the BFT, the content fingerprint is added to the BFT, and the write data is transferred to the server.

If a block on the storage server is overwritten, the corresponding fingerprint mapping entry is the server side mapping table is invalidated. This ensures that a fingerprint is no longer mapped to an overwritten disk block. A mapping of new fingerprint and overwritten disk block location is then added to the server mapping table.

The system implementation explained above helps the partial list of fingerprint maintained in a BFT deal with key issues of limited size and out-of-date entries in the following way. This optimization is available only for a limited portion of the write traffic, which is able to eliminate sending the write data to the server. Also, for any out-of-date fingerprint entry in the BFT, the server will reject the write request if it cannot find the fingerprint in its mapping table. This would further require re-issuing the write requests with write data to the server.

In table 4.3, column 3 shows the total amount of main memory consumed by the BFT tree structure, for different number of entries in the BFT and replacement policies. The total memory consumed by BFT tree is a product of the number of entries in the BFT and size of tree node structure.

n * BlockSize	Replacement Policy	Memory consumed
4GB	Random	264
8GB	Random	528
16GB	Random	1056
4GB	FIFO	328
8GB	FIFO	656
16GB	FIFO	1312
4GB	LRU	328
8GB	LRU	656
16GB	LRU	1312

Table 4.3: Memory consumed by BFT in MB - Fingerprint Cache for Inline Write Deduplication. Here, n represents the number of block fingerprints tracked in the BFT.

The number of entries in the BFT is equal to the number of block frames that are determined to be present on the server. Hence, a BFT with n entries can eliminate transfer of write data to upto n * BlockSize bytes of data on the storage server. The leftmost column in the table 4.3, shows this size which is equal to the product of number of fingerprints tracked in the BFT and the block size.

CHAPTER 5

Evaluation

In this chapter, we discuss the trace driven simulation study of different hypervisor side optimization techniques for VDI specific I/O workload. The two most critical components of this projects were, collecting I/O Traces for the workload generated by VDI environment and simulating caches by replaying traces. Our trace data represents VDI workload generated by common applications like MS Office Suite, Web Browser, Video Player, File archiving software, etc. Apart from this, I/O traces also represent workload from bursty IO activities in VDI deployments like provisioning virtual desktops, boot storms, and antivirus scanning. Section 5.1 and section 5.2 explains VDI workload benchmarks and I/O tracing setup respectively. Section 5.3 explains any differences between the design and implementation of the optimization techniques discussed in chapter 4 and cache simulator implementation used in our experiments.

5.1 VDI Benchmarks

This section presents a detailed explanation of the I/O workload in VDI environments and the experiment setup used for evaluating hypervisor side read cache and write deduplication.

The gamut of VDI deployments cover organizations like administrative offices, call centers, universities, hospitals, banks, etc. One could imagine common desktop applications like Microsoft office suite, Web Browsers, Video Players, etc. running on popular Windows OS, to be prevalent in virtual desktops. Administrators also install custom applications that fit their organizational need, for example, universities might install a self-training software for their students, hospitals might install patient record management software, and so on. We

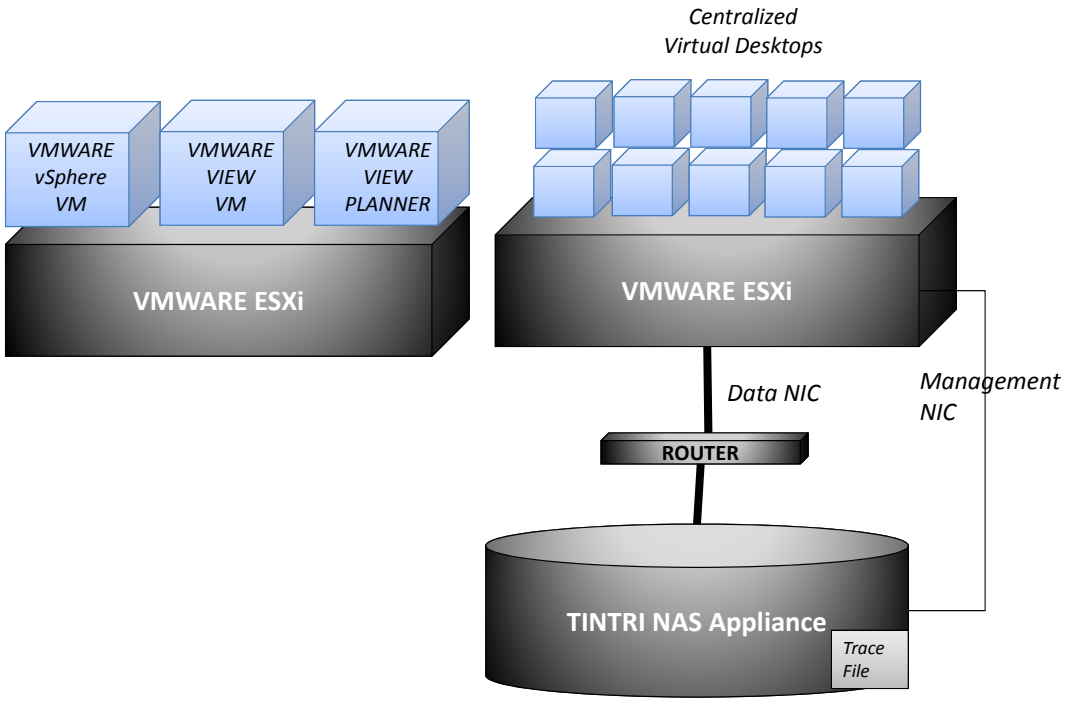


Figure 5.1: Experimental Testbed

call this traffic generated by common desktop applications as a *normal user activity workload*. Moreover, as discussed in chapter 3, the VDI workload is marked by a highly bursty behavior, for example, virtual desktop provisioning, bootstorms, and antivirus scans.

A typical VDI setup shown in figure 5.1 consists of following components: hypervisor, virtual desktop infrastructure management, virtual infrastructure management, and file servers.

- *Hypervisor* is a software which multiplexes the underlying physical resource among different virtual machines.
- *Virtual Desktop Infrastructure Management* automates the creation, management, and deletion of full clone and linked cloned virtual desktops.
- *Virtual Infrastructure Management* provides centralized management for multiple hypervisors and virtual machines running on the hypervisor.
- *NAS Storage* is a network file server used for storing virtual disks for virtual desktop.

Our goal was to collect traces from an I/O workload which would represent the above VDI

deployment. Unfortunately, we could not get traces from a real-world VDI deployment and hence we had to generate the workload synthetically. We used VMware View Planner [Vie11], to automate generation of synthetic VDI workload on Windows 7 virtual desktops running following applications: MS Word, MS Excel, MS Powerpoint, MS Outlook, Internet Explorer, Mozilla Firefox, Adobe Acrobat Reader, Archiving software, and Video playback software. We did not use any custom applications because the above application set represents most of the popular virtual desktop deployment setups. We used a NAS appliance from Tintri as storage for virtual desktop disks. This appliance was mounted on VMware ESXi hypervisor host, hosting virtual desktops. The physical server running ESXi had 8 CPU cores and 72GB of RAM. All the I/O traces where collected on NFS server running in the Tintri appliance as discussed in section 5.2.

In this work, we study a wide range of I/O activity common in VDI workload, as described below,

- Virtual Desktop Provisioning: This activity involves creation of a virtual desktop, cloned from a parent template. Provisioning is a very common scenario in VDI deployments and high write I/O traffic activity. We trace provisioning activity of 30 full clone and linked clone virtual desktops separately.

- Normal user activity: This activity comprises of running a set of application commonly used in VDI environment. Each benchmark run comprises of following user operation being triggered:

  - Adobe Acrobat Reader (Open, Browser, Minimize, Maximize, Close)
  - MS Excel Sort (Open, Compute, Entry, Maximize, Minimize, Save, Close)
  - Firefox, IE Apache Doc IE Web Album (Open, Browse, Close)
  - MS Outlook (Open, Maximize, Read, Restore, AttachmentSave, Close)
  - MS Powerpoint (Open, Append Slides, Minimize, Maximize, ModifySlides, RunSlideShow, Close)
  - Video (Open, Play, Close)

- 7-zip (Compress)
- MS Word (Open, Modify, Save, Minimize, Maximize, Close)

The sequence in which these user operations are exectued is randomized on every virtual desktop. We run seven iterations of this benchmark which takes approximately eight hours to finish, representing one full day of work. The time spent by human user to take an action after an application has responded to a previous request is called think time. We configure this think time to be 15 seconds. Moreover, we also simulate the following activities on 50 full clone and linked clone virtual desktops separately.

- Bootstorm/Loginstorm: As part of the initial start-up of the benchmark run, we trace power-on and login operation of virtual desktops used to run View Planner benchmark.
- Antivirus Scans: These scans generate a lot of read I/O traffic on the file systems being scanned. To simulate the scanning activity and collecting traces for the same, we use AVG Antivirus software to perform scheduled antivirus scans every five hours.

5.2 IO Tracing

We have implemented the trace collection module on the Tintri NFS server. The NFS client implementation in ESXi hypervisor issues NFS read() and write() calls to the NFS server. Each of this read and write call is recorded into a file in a format as defined by Dataseries tracing library. Dataseries [AAI09] is an on-disk format, runtime library and set of tools for storing and analyzing structured serial data, for example, I/O traces. Dataseries outperforms common trace formats and databases by an order of magnitude and requires less disk space.

The ESXi hypervisor used in our experimental setup has 8 CPU cores for running upto 50 virtual desktop. The NFS client implemenation in the ESXi hypervisor handles the interleaving of all the I/O requests to virtual disks stored on NFS server. We use only one ESXi hypervisor and hence the I/O workload to NFS server is generated by a single NFS

FileId	Op	Offset	Size	Latency	Timestamp	BlockLevel	Fingerprint	CF
1205	0	1538	1024	463...	13305...	F		
1205	0	1538	512	463...	13305...	T	f4eb80...	0.38
1205	0	2050	512	463...	13305...	T	ede47d...	0.68
...	...	...	...	...	...	...	...	...

Table 5.1: Sample trace file

client.

A sample trace file in shown in table 5.1. Each trace entry has a set of fields, as described below,

- FileId: This field represents a unique virtual disk file identifier used to identify the virtual disk.
- Op: This field represents the kind of operation, i.e. either read or write, and is used by the simulator to be handle the I/O trace entry for either read or write optimization.
- Offset: This field represents the offset location in the virtual disk used to calculate the block number and form a location string, as discussed in chapter 4.
- Size: This field represents the size of I/O operation, and is used by the simulator to calculate the amount of data transferred in the I/O operation.
- Latency: This field represents the time spent to perform a read or a write operation in microseconds. This field is not used by the simulator.
- Timestamp: This field represents the timestamp at which the trace entry recorded. The timestamp information can help us know the time at which I/O activity took place.
- BlockLevel: This field represents if the trace entry was collected at NFS I/O level or at block granularity level of 512-bytes. Every I/O request is first traced at NFS level and then the data read/written is split into chunks of 512 bytes blocks and traced at block

level. The size of 512 bytes was chosen because it is minimum I/O request size made by hypervisor and every other request size is also a multiple of 512. Hence, we manage our cache at 512-byte granularity using all the block-aligned traces. We consider this small size to be a limitation of our study because of larger block sizes used in current file system cache. Large block size being a more practical option due to better locality and low metadata overhead. However, because we use traffic ratio as our evaluation metric, this small block size will not effect our results dramatically. In addition, our block size does not suffer from any fragmentation issues.

- Fingerprint: This field reprsents a SHA-1 hash of block content being traced. Fingerprint is used by content-addressed read cache, and fingerprint cache for inline write deduplication.

- CF: This field represents the compression factor for the data block read or written. This field is not used by the simulator.

Apart from this trace file data, we maintain a separate mapping of virtual disk file identifier to virtual disk file names. This is useful to distinguish accesses by different virtual desktops. We do not include file name as part of each trace entry, thus achieving significantly storage space savings. This mapping is useful in determining the read-only shared disks in linked clones, so that they can be stored on local media on hypervisor side.

5.3 Cache Simulator Implementation

We have implemented a cache simulator for evaluating the optimization techniques described in chapter 4. The main function of the cache simulator is to compute the overall reduction in the I/O traffic with the introduction of hypervisor side read cache or write deduplication mechanism. The cache simulator uses only a cache directory implementation to calculate the total number of I/O request that hit in the cache directory, thereby determining the overall reduction in the I/O traffic. The simulator does not maintain an on disk cache data file for reading and writing cache data. It replays the I/O traces, discussed in section 5.2,

and performs an optimization on each trace entry. The simulator calculates the total traffic that is transferred over the network, after applying the optimization techniques. An overall traffic ratio can be computed as shown in section 5.4. Our cache simulator implementation is functionally equivalent to the actual system implementation.

The simulation results might not be as accurate as the actual system implementation results. This is because, if the I/O request responded faster, the think time of a user might vary. Moreover, operating systems might get more work done in less amount of time. Although, these are common drawbacks of simulation studies, simulations behaves very close to real world systems.

The traces collected on the NFS server represents one particular interleaving of I/O requests from different virtual desktops. If the timing among the different virtual desktops is skewed slightly differently, traces will represent a different interleaving of I/O requests. Studying the effect of a different interleaving of I/O requests on caching behavior is an interesting aspect. In this setup, we can determine the validity of simulation results for different interleavings by perturbing the trace. As every trace entry maintains timing information of the I/O activity, it is easy to simulate different interleavings. We plan to collect more simulation results for different interleaving using the same workload (same traces) in future.

The cache directory implemented in the simulator uses the Set data structure from C++ Standard Template Library (STL). This set data structure is an associative container for storing unique elements. Internally this set data structure is implemented as a red-black tree, and hence, resembles to the design of cache directory discussed in section 4.2. The simulator uses the following three APIs exposed by the set data structure,

- insert(): This function is used for inserting an element into the set. Internally, it inserts an element into the binary search tree in such a way that the order is maintained. Hence, the time complexity of this insertion is O(logN).

- find(): This function is used for performing a lookup operation on the set, and get a pointer to the set entry. Internally, it performs a lookup on the binary search tree, and returns a pointer to the node found. Hence, the time complexity of this lookup is

O(logN).

- erase(): This function is used to delete an element from the set. For this function to be called, a pointer to the node should be obtained, which is then used to delete the node. Assuming that the pointer to the node is available, deleting a node can take place in constant time. Otherwise, the total time complexity of the overall erase operation, i.e. obtaining a pointer to the node present in the tree, and then calling erase() on the pointer to the node, is O(logN).

The implementation of replacement policy is functionally identical for the actual and simulator implementation. However, the time complexity of Random, FIFO and LRU eviction algorithms differ in the simulator implementation. For FIFO and LRU replacement policies, the eviction of a victim node from the tree involves using the key in the head node of the doubly linked list to perform a find operation on the set, getting a pointer to the set entry, and finally calling an erase operation on the pointer. Hence, the time complexity of FIFO and LRU eviction algorithm is O(logN). For Random replacement policy, the eviction of a victim node from the tree involves a sequential iterations over the tree nodes, till it reaches the nth node which is the victim node, and where n is a randomly generated number. A sequential iteration is must on every tree node because the set data structure does not support a random access iterator. Hence, the time complexity for Random eviction algorithm is O(N) for the simulator.

The implementation of all the optimization techniques in the cache simulator matches the actual system implementation, except the content-addressed read cache, as discussed below.

The cache simulator implementation of level-1 table in a content-addressed read cache differs from actual system implementation. As shown in table 4.2, the amount of memory consumed by level-1 and level-2 tree structures can be as high as 5GB. Most of the development machines we used to run these simulations were low on the physical memory. Hence, the level-1 tree structure which consumes 20 bytes per tree node was replaced by a block bitmap per virtual disk. The data structures used by content-addressed cache simulator,

```
// Maps a virtual disk to its corresponding block bitmap.
map< uint64, uint64* > level1Cache_;

// For each virtual disk we maintain a block bitmap datastructure
// For a 25GB virtual disk, approx. size of this block bitmap is 6MB
blockBitMap = new uint64[ NUM_OF_BLOCKS / ( sizeof( uint64 ) * 8 ) ];
level1Cache_[vdiskFileId_] = blockBitMap;

//Each node consumes 33 bytes for Random, and 41 bytes for FIFO and LRU
struct node {
      string fingerprint_; // 20bytes
      char color_; // 1 byte
      struct node *parent_, *left_, *right_; // 4 bytes * 3

     // prev_ and next_ pointers are only used for FIFO and LRU
     struct node *prev_, *next_; // 4 bytes * 2

     // stores a list of locations that map to this fingeprint node
     vector<string> locationArray;
};
```

Figure 5.2: Data Structures used in Content Addressed Read Cache Simulator

```
// Determine if the bit corresponding to a virtual disk block is set/unset in the block bitmap
uint64 arrayIndex = blockNum_ / ( sizeof( uint64) * 8 );
uint8 indexOffset =  blockNum_ % ( sizeof( uint64 ) * 8 );

// Every virtual disk has a blockBitMap array
bool level1Hit = ( blockBitMap[ arrayIndex ] & ( 0x1 << indexOffset ) ) == 0
                        ? false : true;

// Set the bit corresponding to a virtual disk block
blockBitMap[ arrayIndex ] |= ( 0x01 << indexOffset );
```

Figure 5.3: Bitwise operations on per virtual disk block bitmap

i.e. block bitmap per virtual disk for level-1 table and tree node structure of level-2 table are shown in figure 5.2. For every access to a block location, the corresponding bit in the virtual disk block bitmap is set. Any future reads to same block location first determines if the corresponding bit is set/unset in that virtual disk block bitmap. If the bit is set, it is a level-1 cache hit, otherwise, it is level-1 cache miss. As we input a trace file to our simulator, each trace entry has a fingerprint value, which is then looked up in level-2 tree structure. For a cache hit, the block location must be set in the bitmap, and fingerprint should hit in the level-2 table. For writes, if the bit corresponding to write block location is set, the fingerprint from the trace entry is looked up in the level-2 table. If the fingerprint is not found, it is added to the level-2 table. Otherwise, we implement a write-around policy, where the bit and level-2 table remains unchanged. For every fingerprint that is evicted from level-2 table, all the block location bits mapped to the evicted fingerprint entry are unset. We use the locationArray shown in figure 5.2, to determine the bits which should be unset on a fingerprint eviction.

This design is not feasible for the actual system implementation described in section 4.2.2 because, there is no way to know a fingerprint which corresponds to a disk block location without maintaining a location to fingerprint mapping. We chose a block bitmap design for level-1 table in our simulator implementation because of low memory overhead and fast bitwise operations used to determine a cache hit or cache miss in level-1 table. The bitwise operations used in our code to determine if bit corresponding to a virtual disk block is set/unset is given in figure 5.3.

The use of block bitmap is an optimal design which helped us efficiently use the small amount of physical memory available on our development machines, used to run simulations. In case of a hypervisor side content-addressed cache running on high-end virtualized servers, we expect the actual system implementation to run without any resource starving. This is just an optimization to reduce the memory overhead, and in no way violates the content-addressed read cache functionality.

St. Ts.	End Ts.	#Reads	#Writes	Read Traffic Ratio	Write Traffic Ratio
02:14	02:15	106337	6696	0.456	0.342
02:15	02:16	54283	7114	0.523	0.431
02:16	02:17	121190	10568	0.675	0.318
...	...	...	...	...	...

Table 5.2: Epoch-by-Epoch Analysis file

5.4 Analysis

We started our analysis by asking following questions,

- What are the characteristics of the IO storage workload generated by the setup explained in section 5.1? Can we identify the periods of bursty I/O traffic?
- How does one cache perform as compared to other?
  - Is this observation true for both full clones and linked clones?
  - Is this observation true for provisioning and normal user activity workload?
- How does one cache replacement policy perform as compared to other?
- What is a good evaluation metric for this trace-driven simulation study?
- What is the footprint size?

  We calculate the footprint of the workload traced, i.e. the number of unique blocks touched over the period of trace duration. The footprint size can help us determine an optimal benefit for any optimization simulated.

We fed the I/O traces described in section 5.2 to different cache combinations implemented in our cache simulator. For each combination, the simulator generated a detailed analysis of the I/O workload as shown in table 5.2. For a No Cache scenario, where we replay the trace entries without any optimization, this table helped us understand the overall traffic over time and calculate the average and peak network bandwidth consumed. In the

scenario of simulating optimizations, we record entries in the analysis file after every 1GB of I/O traffic from the trace entries is being replayed through the simulator. This helped us calculate the overall, average and peak traffic ratio, with optimizations simulated.

What is a good evaluation metric for this trace-driven simulation study?

As this work is a trace driven simulation study, using network bandwidth or IOPs would have been an incorrect measure to evaluate different optimization techniques. This is because access latency would be different for an access from a local disk cache as against access over network. Hence, we use traffic ratio as an evaluation metric which is defined as follows,

$$\frac{Bytes\ Accessed\ Over\ Network}{Bytes\ Requested\ By\ Client} \tag{5.1}$$

As location-addressed cache and content-addressed cache eliminates only the read traffic, and fingerprint cache for inline write deduplication eliminates only the write traffic, we state read traffic ratio and write traffic ratio seperately. In all cases, the goal of the various mechanisms is to minimize the traffic ratio.

What is the footprint size?

Footprint size of an entire workload signifies the number of unique blocks touched. With a fully associative cache, a cache of footprint size can accomodate all the unique blocks touched, thus avoiding any cache misses other than compulsory misses. By simulating a cache of footprint size, we can determine the optimal benefits of any mechanisms under a particular workload.

The footprint size for a workload using a location-addressed read cache is calculated by maintaining a block bitmap per virtual disk. For every access to a block location, we set the corresponding bit in the virtual disk block bitmap. At the end of the simulator run, we calculate the number of bits set in all the block bitmaps. The footprint size for the workload is a product of total number of bits set and block size.

The footprint size for a workload using a content-addressed read cache is the number of unique fingerprints touched. This can be implemented by maintaining a set of all the fingerprints replayed over the trace duration. For every access to a block, the fingerprint is added to the set, if it is not already present. At the end of the simulator run, we calculate

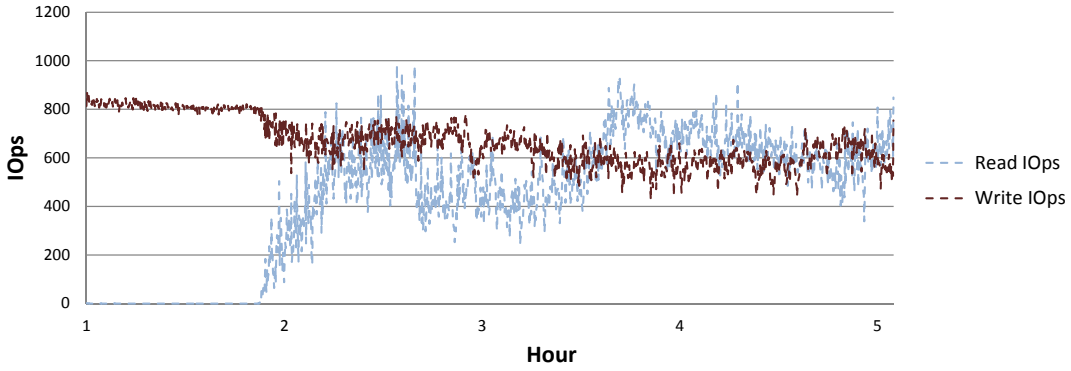


Figure 5.4: Provisiong 30 full clone virtual desktop traffic in IOPs

the number of fingerprint in the set. The footprint size for the workload is a product of number of fingerprints and block size.

5.4.1 Full Clone

5.4.1.1 Provisioning Workload

We traced provisioning activity of 30 full clone virtual desktops running Windows 7 operating system. Each virtual desktop was provisioned with a virtual disk of 30GB. Figure 5.4 on page 45 and figure 5.5 on page 46 shows IO traffic measured as I/O rate (in IOPs) and throughput (in MBps) over the entire trace duration.

Provisioning is a period of high write activity because of copying of OS and application data to newly cloned virtual desktop. This is evident in table 5.3, where the amount of data written is 85% of the total data transferred over the network. In addition, VMware View boots the virtual desktops multiple times in order to perform specific configurations in virtual desktops. Therefore, overall provisioning workload represents copying the virtual desktop data, first-time booting of virtual desktops, and VMware View specific configurations. Table 5.3 also shows the peak and average I/O rate (in IOps) and throughput (in MBps) for read and write traffic separately.

We simulate *location-addressed cache*, *content-addressed cache*, and *fingerprint cache for inline write deduplication*, for full clone provisioning workload discussed above. We dis-

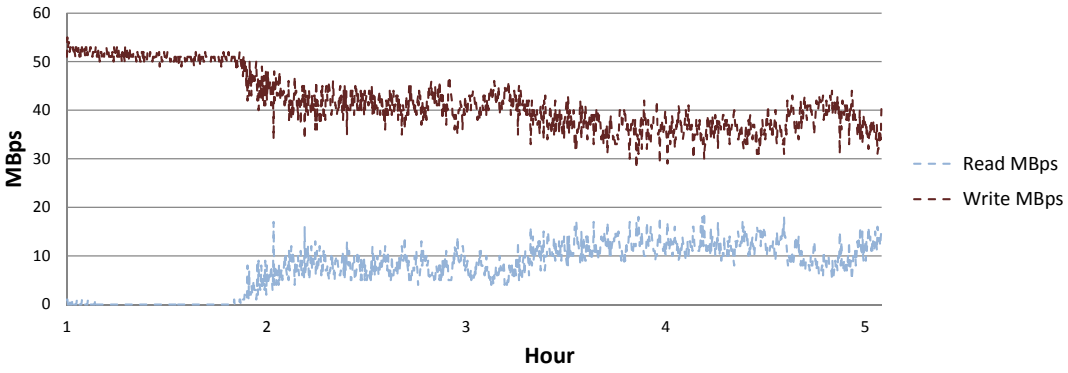


Figure 5.5: Provisiong 30 full clone virtual desktop traffic in MBps

	Read	Write
# of IOs	5731K	8811K
Data transferred in GB	92GB	532GB
Peak Rate (IOps)	983	871
Peak Throughput (MBps)	19	55
Average Rate (IOps)	434.265167	667.4001363
Average Throughput (MBps)	7.580095433	41.84253579

Table 5.3: Provisiong 30 full clone virtual desktop traffic

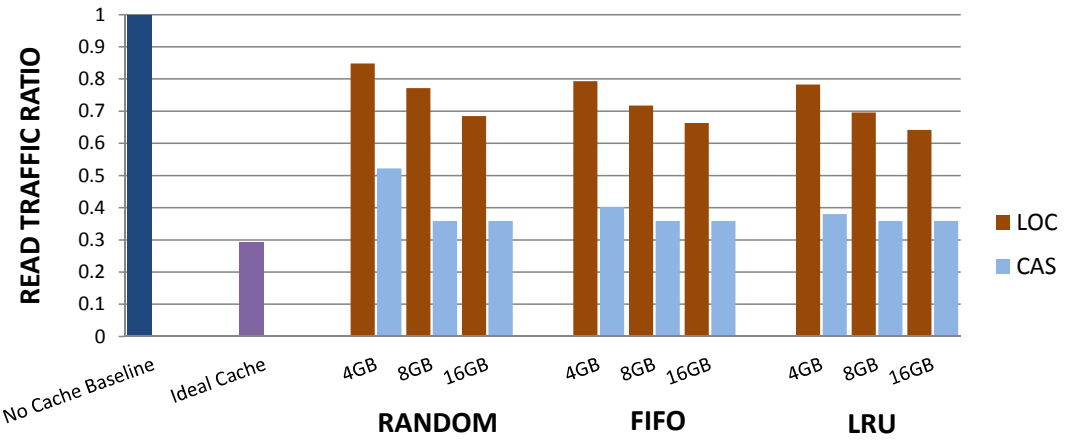


Figure 5.6: Provisioning 30 full clone virtual desktop - Comparison between Location-addressed Cache (LOC) and Content-addressed Cache (CAS) for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). No cache is a baseline scenario which replays the trace without any optimization. Ideal cache is an infinitely sized cache and represents an upper bound on traffic reduction.

cuss the benefits of each of the optimization technique for different cache sizes and cache replacement policies.

Figure 5.6 on page 47 shows a comparison of read traffic ratio between location-addressed cache and content-addressed cache for varying cache size and cache replacement policy. Figure 5.7 on page 48 shows the write traffic ratio as it is calculated for varying cache size and cache replacement policy. All the results are compared with a baseline which represent a no cache scenario. A brief summary of the results in figure 5.6 and figure 5.7 is described below,

- Location-addressed Cache: As shown in figure 5.6, the minimum percentage reduction in the *read traffic* was 15% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 36% as observed with a 16GB cache using a LRU replacement policy.

  The footprint size for full clone provisioning workload was 27GB. The cache of footprint size is an ideal cache, and results in 70% reduction in the *read traffic*.

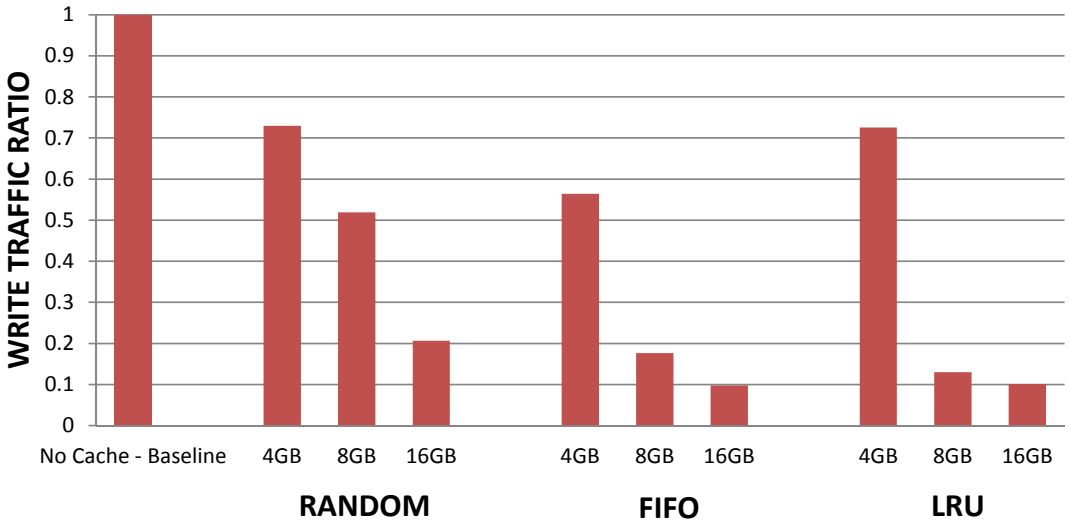


Figure 5.7: Provisioning 30 full clone virtual desktop - Fingerprint Cache for write deduplication for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). Note that, a 4GB cache size represents 4GB/BLOCK_SIZE number of entries in the fingerprint cache.

- Content-addressed Cache: As shown in figure 5.6, the minimum percentage reduction in the *read traffic* was 48% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 64% as observed with a 8GB cache for any of the three replacement policy.

- Fingerprint Cache for Inline Write Deduplication: As shown in figure 5.7, the minimum percentage reduction in the *write traffic* was 27% as observed with a 4GB cache using random replacement policy, and maximum percentage reduction in the *write traffic* was 90% as observed with a 16GB cache using any of FIFO or LRU replacement policy.

5.4.1.2 Normal User Activity Workload

As described in section 5.1 we traced I/O activity of a workload which represents a typical VDI deployment using a synthetic workload generator, View Planner. Trace was collected over period of 9 hours. The benchmark run begins with power on operation of 50 virtual desktops. VMware View Planner introduces a sleep of 1min after powering on every 4 virtual

	Read	Write
# of IOs	28770K	7689K
Data transferred in GB	477GB	358GB
Peak Rate (IOps)	6341	1486
Peak Throughput (MBps)	47	41
Average Rate (IOps)	1032.084302	275.5306848
Average Throughput (MBps)	18.04618863	13.69379845

Table 5.4: Normal user activity 50 full clone virtual desktop traffic

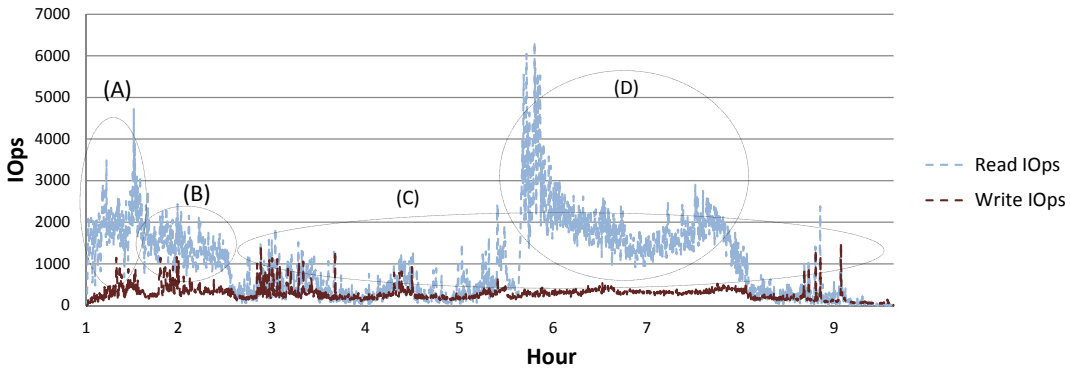


Figure 5.8: Normal user activity 50 full clone virtual desktop traffic in IOPs

desktops, thus avoiding very high I/O bursts. This boot and login operation is followed by seven iterations of common desktop applications run. First iteration is period of high I/O activity, because the buffer cache within the guest operating system is cold. This iteration is followed by remaining six iterations where the order of execution of user operations is randomized.

Table 5.4 describes the traffic details, including bytes transferred, peak and average I/O rate (in IOps), peak and average throughput (in MBps). Table 5.5 describes the division of the I/O workload for traffic shown in figure 5.8 on page 49. This division of I/O traffic is used to identify the periods of bursty behaviors in the workload. Figure 5.9 on page 50 shows a traffic dominated by higher read bandwidth, mainly because these applications generate higher read traffic than write traffic. As shown in the figure, this workload is read-heavy in IOPs (73% higher) and throughput (27% higher).

Activity	Description
(A)	Booting and login operation of 50 full clone virtual desktop
(B)	First iteration run of View Planner benchmark run
(C)	Subsequent six iterations of View Planner benchmark run
(D)	Scheduled Antivirus scan

Table 5.5: Division of I/O traffic during normal user activity

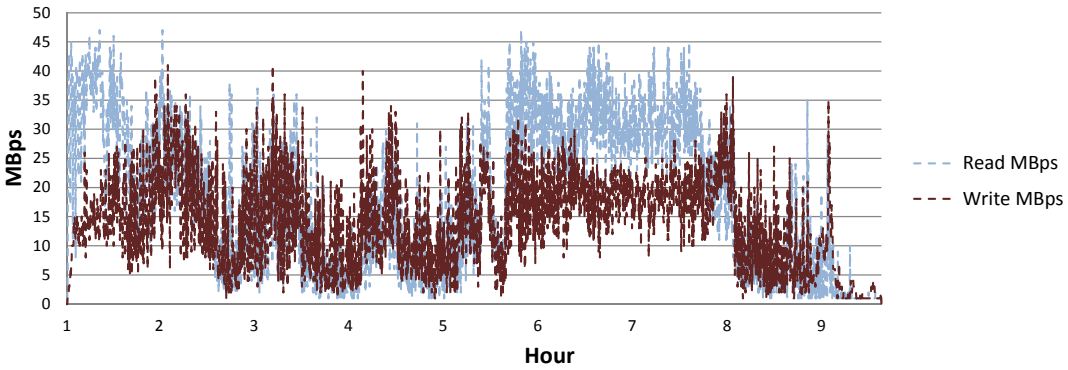


Figure 5.9: Normal user activity 50 full clone virtual desktop traffic in MBps

Similar to provisioning workload, we simulate *location-addressed cache*, *content-addressed cache*, and *fingerprint cache for inline write deduplication*, for full clone normal user activity workload discussed above. We discuss the benefits of each of the optimization technique for different cache sizes and cache replacement policies.

Figure 5.10 on page 51 shows a comparison of read traffic ratio between location-addressed cache and content-addressed cache for varying cache size and cache replacement policy. Figure 5.11 on page 51 shows the write traffic ratio as it is calculated for varying cache size and cache replacement policy. All the results are compared with a baseline which represent a no cache scenario. A brief summary of the results in figure 5.10 and figure 5.11 is described below,

- Location-addressed Cache: As shown in figure 5.10, the minimum percentage reduction in the *read traffic* was 3% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 19% as observed with a 16GB cache using a LRU replacement policy.

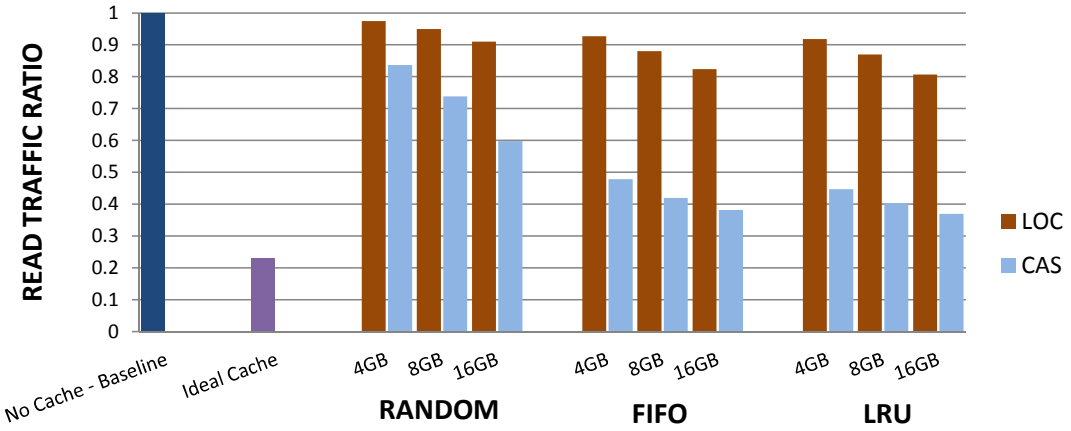


Figure 5.10: Normal user activity 50 full clone virtual desktop - Comparison between Location-addressed Cache (LOC) and Content-addressed Cache (CAS) for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). No cache is a baseline scenario which replays the trace without any optimization. Ideal cache is an infinitely sized cache and represents an upper bound on traffic reduction.

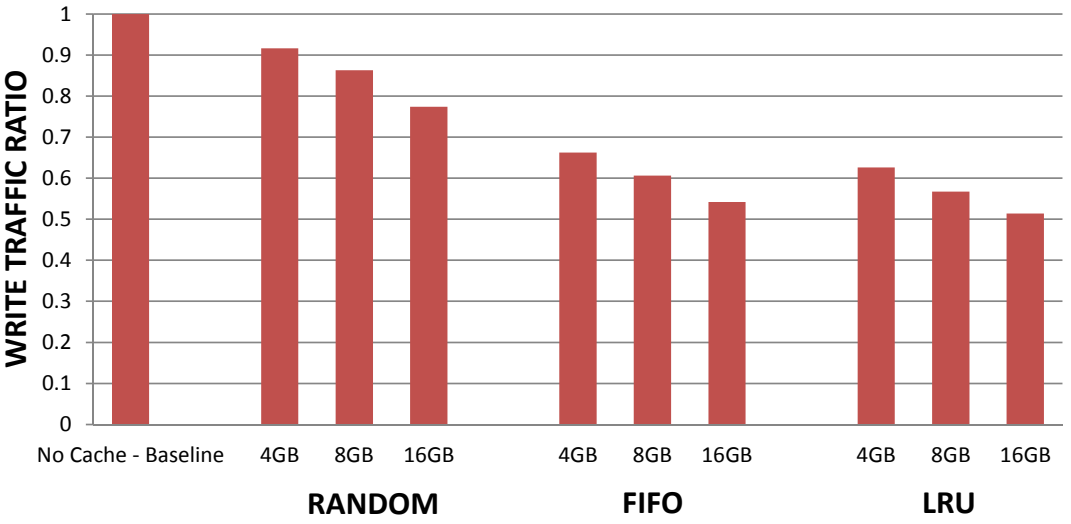


Figure 5.11: Normal user activity 50 full clone virtual desktop - Fingerprint Cache for write deduplication for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). Note that, a 4GB cache size represents 4GB/BLOCK_SIZE number of entries in the fingerprint cache.

The footprint size for full clone normal user activity workload was 110GB. The cache of footprint size is an ideal cache, and results in 77% reduction in the *read traffic*.

- Content-addressed Cache: As shown in figure 5.10, the minimum percentage reduction in the *read traffic* was 16% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 63% as observed with a 16GB cache using a LRU replacement policy.

- Fingerprint Cache for Inline Write Deduplication: As shown in figure 5.11, the minimum percentage reduction in the *write traffic* was 8% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *write traffic* was 49% as observed with a 16GB cache using a LRU replacement policy.

5.4.2 Linked Clone

5.4.2.1 Provisioning Workload

Unlike provisioning activity of full clones, linked clone virtual desktop provisioning involves only one copy of the parent virtual disk image which is used as a shared read-only virtual disk between multiple virtual desktops. A read-write differential disk is created for each virtual desktop. Hence, the traffic is not as write-heavy as seen in section 5.4.1.1. Significant amount of I/O involves read traffic from VMware View specific configurations to establish an extra layer of indirection of base and differential disks.

Figure 5.12 on page 53 and figure 5.13 on page 53 shows IO traffic measured as I/O rate (in IOPs) and throughput (in MBps) over the entire trace duration. Table 5.6 also shows the peak and average I/O rate (in IOps) and throughput (in MBps) for read and write traffic separately. Table 5.6 shows that I/O traffic is read-heavy in IOPs (70.5% higher) and throughput (52% higher).

In addition to simulating *location-addressed cache*, *content-addressed cache*, and *fingerprint cache for inline write deduplication*, we also use three additional optimization techniques specific to linked clones. These optimizations leverage the read-only shared virtual

	Read	Write
# of IOs	19365K	5703K
Data transferred in GB	405GB	195GB
Peak Rate (IOps)	4857	2513
Peak Throughput (MBps)	97	56
Average Rate (IOps)	1478.893471	435.7828179
Average Throughput (MBps)	31.66323024	15.2652921

Table 5.6: Provisiong 30 linked clone virtual desktop traffic

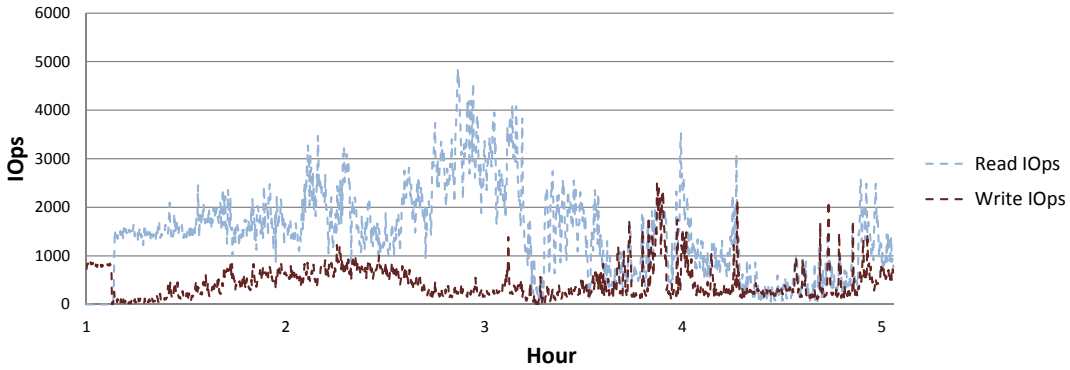


Figure 5.12: Provisiong 30 linked clone virtual desktop traffic in IOPs

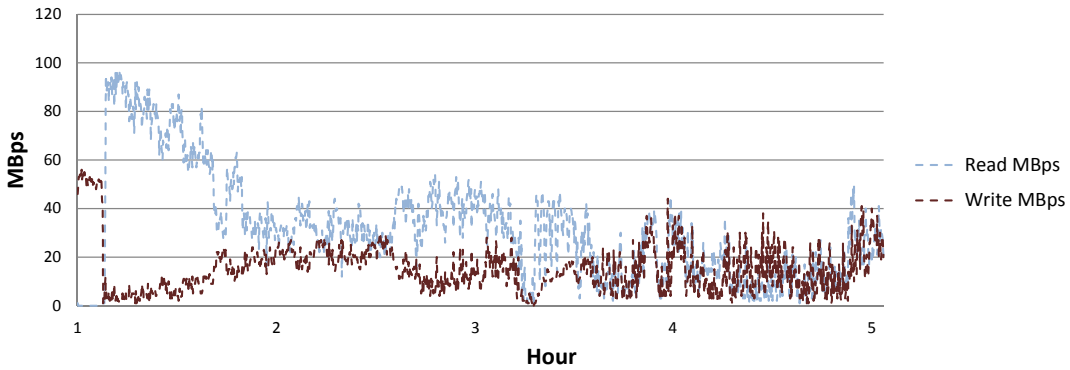


Figure 5.13: Provisiong 30 linked clone virtual desktop traffic in MBps

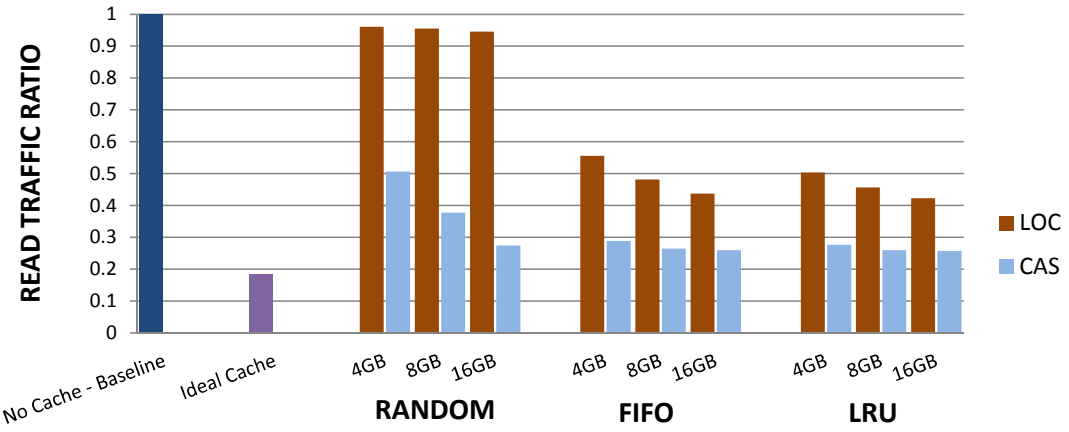


Figure 5.14: Provisioning 30 linked clone virtual desktop - Comparison between Location-addressed Cache (LOC) and Content-addressed Cache (CAS) for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). No cache is a baseline scenario which replays the trace without any optimization. Ideal cache is an infinitely sized cache and represents an upper bound on traffic reduction.

disk by storing it on local media of virtualized servers. We discuss the benefits of each of the optimization technique for different cache sizes and cache replacement policies.

Figure 5.14 on page 54 shows a comparison of read traffic ratio between location-addressed cache and content-addressed cache for varying cache size and cache replacement policy. Figure 5.15 on shows the write traffic ratio as it is calculated for varying cache size and cache replacement policy. All the results are compared with a baseline which represent a no cache scenario. A brief summary of the results in figure 5.14 and figure 5.15 is described below,

- Location-addressed Cache: As shown in figure 5.14, the minimum percentage reduction in the *read traffic* was 4% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 58% as observed with a 16GB cache using LRU replacement policy.

  The footprint size for linked clone provisioning workload was 74GB. This footprint size is larger than full clone provisioning workload because of the higher read activity on each differential disk to perform VMware View specific configuration. The cache of

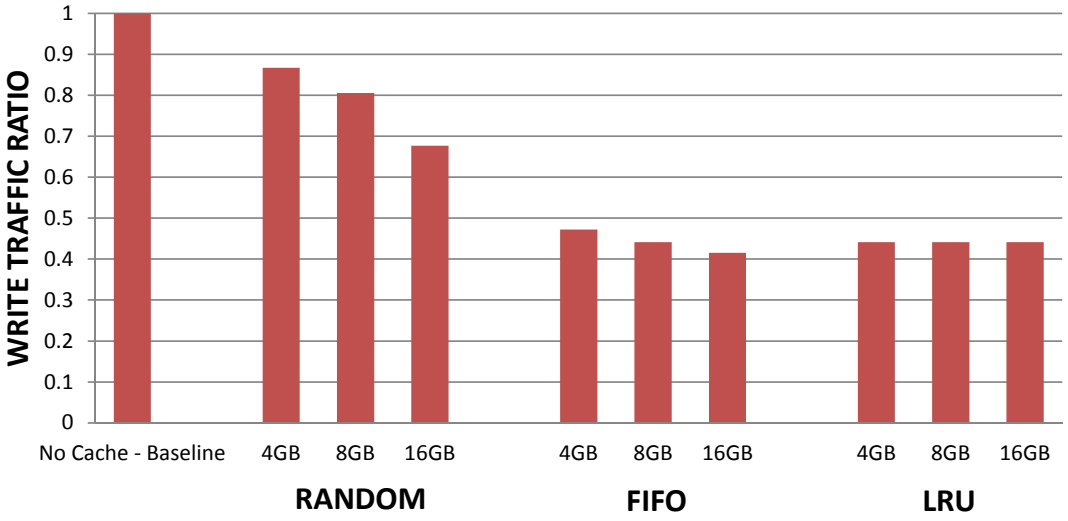


Figure 5.15: Provisioning 30 linked clone virtual desktop - Fingerprint Cache for write deduplication for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). Note that, a 4GB cache size represents 4GB/BLOCK_SIZE number of entries in the fingerprint cache.

footprint size is an ideal cache, and results in 81% reduction in the *read traffic*.

- Content-addressed Cache: As shown in figure 5.14, the minimum percentage reduction in the *read traffic* was 49% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 74% as observed with a 16GB cache using LRU replacement policy.

- Local store for Shared Read-only Disk: Simply storing the entire shared read-only virtual disk on local media resulted in a percentage reduction of 42% in *read traffic*.

- Local store for Shared Read-only Disk and location-addressed Cache: In this experiment we stored the shared read-only virtual disk on local media in addition to using a 16GB location-addressed cache using LRU replacement policy. All the read request to the shared disk were served from the local copy of disk, and reads to the differential disk was served from the location-addressed cache. Approximately 60% of *read traffic* reduction was observed with this double layer of caching. We replaced 16GB cache with a small 1GB cache and observed upto 50% reduction in the *read traffic* which was

approximately same as a vanilla 4GB location-addressed cache with LRU replacement policy.

- Local store for Shared Read-only Disk and Content-addressed Cache: In this experiment we stored the shared read-only virtual disk on local media in addition to using a 16GB content-addressed cache using LRU replacement policy. All the read request to the shared disk were served from the local copy of disk, and reads to the differential disk was served from the content-addressed cache. Approximately 75% of *read traffic* reduction was observed with this double layer of caching. We replaced 16GB cache with a small 1GB cache and observed upto 71% reduction in the *read traffic* which was approximately same as a vanilla 4GB content-addressed cache with LRU replacement policy.

- Fingerprint Cache for Inline Write Deduplication: As shown in figure 5.15, the minimum percentage reduction in the *write traffic* was 13% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *write traffic* was 58% as observed with a 16GB cache using FIFO replacement policy. Unlike other observations, FIFO replacement policy achieved maximum write traffic reduction. However, percentage reduction in write traffic with LRU was 56%, and hence comparable to FIFO.

5.4.2.2 Normal User Activity Workload

We ran benchmark similar to the one described in section 5.4.1.2 for 50 linked clone virtual desktops deployment. Table 5.7 describe the traffic details, including bytes transferred, peak and average I/O rate (in IOps), peak and average throughput (in MBps). Table 5.8 describes the division of I/O traffic as shown in figure 5.16 on page 57. Figure 5.17 on page 57 shows a traffic dominated by higher read bandwidth, mainly because these applications generate higher read traffic than write traffic. This workload is read-heavy in IOPs (73% higher) and throughput (40% higher).

We simulate all the optimization techniques simulated for provisioning workload of 30

	Read	Write
# of IOs	42225K	11608K
Data transferred in GB	732GB	441GB
Peak Rate (IOps)	7089	2063
Peak Throughput (MBps)	52	44
Average Rate (IOps)	1381.554345	379.4786451
Average Throughput (MBps)	25.04624448	15.29455081

Table 5.7: Normal user activity 50 linked clone virtual desktop traffic

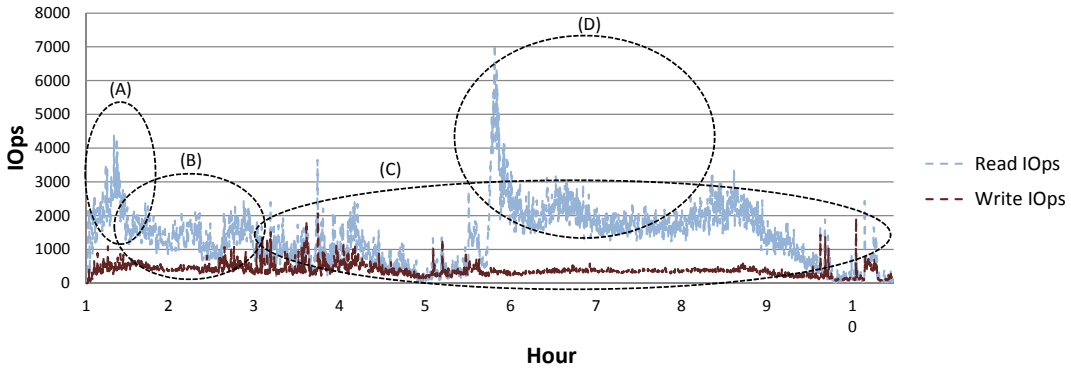


Figure 5.16: Normal user activity 50 linked clone virtual desktop traffic in IOPs

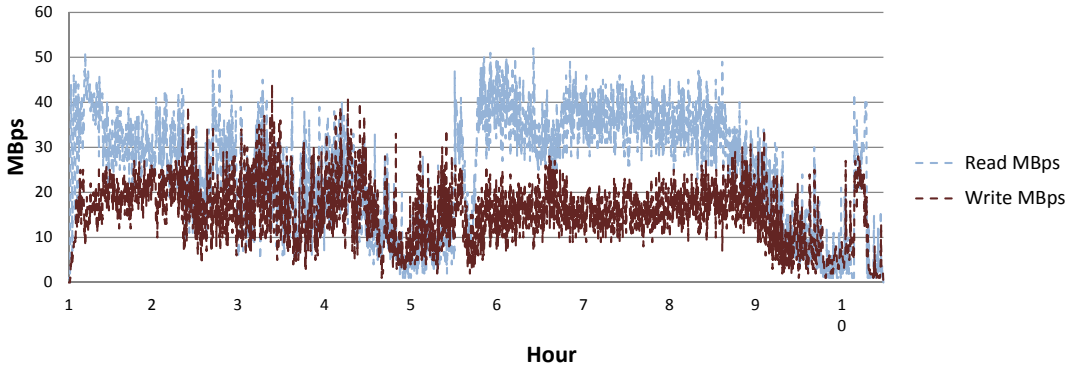


Figure 5.17: Normal user activity 50 linked clone virtual desktop traffic in MBps

Activity	Description
(A)	Booting and login operation of 50 linked clone virtual desktop
(B)	First iteration run of View Planner benchmark run
(C)	Subsequent six iterations of View Planner benchmark run
(D)	Scheduled Antivirus scan

Table 5.8: Division of I/O traffic during normal user activity

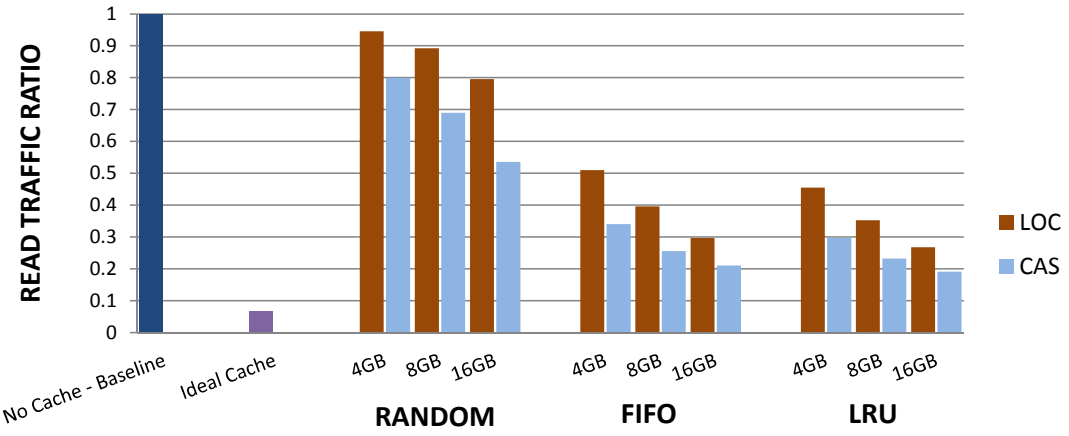


Figure 5.18: Normal user activity 50 linked clone virtual desktop - Comparison between Location-addressed Cache (LOC) and Content-addressed Cache (CAS) for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). No cache is a baseline scenario which replays the trace without any optimization. Ideal cache is an infinitely sized cache and represents an upper bound on traffic reduction.

linked clone virtual desktop in section 5.4.2.1. We discuss the benefits of each of the optimization technique for different cache sizes and cache replacement policies.

Figure 5.18 shows a comparison of read traffic ratio between location-addressed cache and content-addressed cache for varying cache size and cache replacement policy. Figure 5.19 on page 59 shows the write traffic ratio as it is calculated for varying cache size and cache replacement policy. All the results are compared with a baseline which represent a no cache scenario. A brief summary of the results in figure 5.18 and figure 5.19 is described below,

- Location-addressed Cache: As shown in figure 5.18, the minimum percentage reduction in the *read traffic* was 5% as observed with a 4GB cache using a random replacement

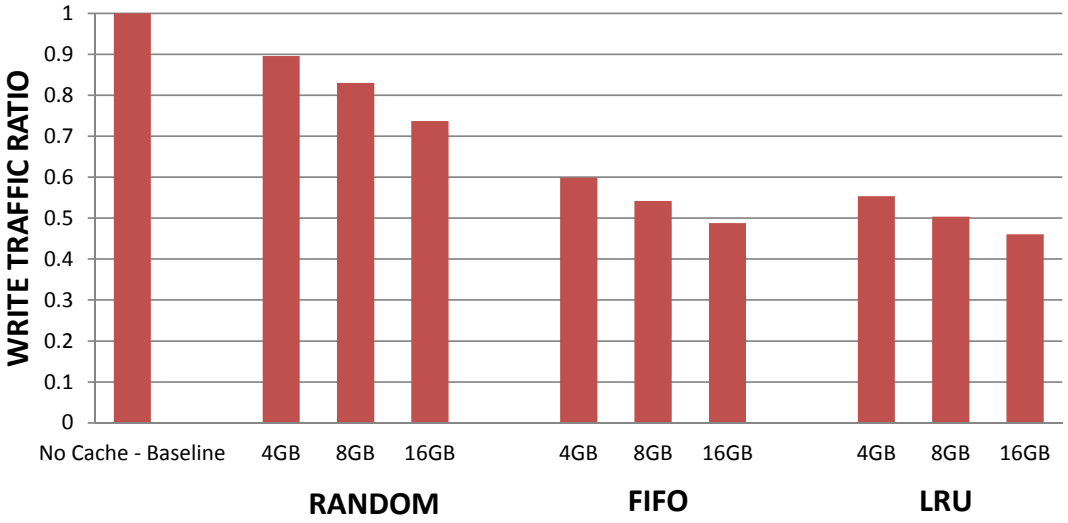


Figure 5.19: Normal user activity 50 linked clone virtual desktop - Fingerprint Cache for write deduplication for different cache sizes (4GB, 8GB, 16GB) and cache replacement policies (FIFO, LRU, RANDOM). Note that, a 4GB cache size represents 4GB/BLOCK_SIZE number of entries in the fingerprint cache.

policy, and maximum percentage reduction in the *read traffic* was 73% as observed with a 16GB cache using LRU replacement policy.

The footprint size for linked clone normal user activity workload was 50GB. This footprint size is smaller than full clone normal user activity workload because a significant amount of read traffic is for the block locations in the shared virtual disk. The cache of footprint size is an ideal cache, and results in 93% reduction in the *read traffic*.

- Content-addressed Cache: As shown in figure 5.18, the minimum percentage reduction in the *read traffic* was 20% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *read traffic* was 81% as observed with a 16GB cache using LRU replacement policy. This result shows that unlike full clone, percentage read traffic reduction with content addressable cache is comparable to location-addressed cache.

- Local store for Shared Read-only Disk: Simply storing the entire shared read-only virtual disk on local media resulted in a percentage reduction of 50% in *read traffic*.

- Local store for Shared Read-only Disk and Location-addressed Cache: In this experiment we stored the shared read-only virtual disk on local media in addition to 16GB location-addressed cache using LRU replacement policy. All the read request to the shared disk was served from the local copy of disk, and reads to the differential disk was served from the location-addressed cache. Approximately 77% of *read traffic* reduction was observed with this double layer of caching. We replaced 16GB cache with a small 1GB cache and observed upto 58% reduction in the *read traffic* which was approximately same as a vanilla 4GB location-addressed cache with LRU replacement policy.

- Local store for Shared Read-only Disk and Content-addressed Cache: In this experiment we stored the read-only shared disk on local media in addition to 16GB content-addressed cache using LRU replacement policy. All the read request to the shared disk was served from the local copy of disk, and reads to the differential disk was served from the content-addressed cache. With this double layer of caching there was 83% reduction in *read traffic*. We replaced 16GB cache with a small 1GB cache and observed upto 68% reduction in the *read traffic* which was approximately same as a vanilla 4GB location-addressed cache with LRU replacement policy.

- Fingerprint Cache for Inline Write Deduplication: As shown in figure 5.19, the minimum percentage reduction in the *write traffic* was 10% as observed with a 4GB cache using a random replacement policy, and maximum percentage reduction in the *write traffic* was 54% as observed with a 16GB cache using LRU replacement policy.

5.5 Observations

In the previous section we discussed simulation results from 27 cache combinations for full clones (3 cache algorithms * 3 cache sizes * 3 replacement policies) and 30 cache combinations for linked clones (3 cache algorithms * 3 cache sizes * 3 replacement policies + 3 cache algorithms) on each of the provisioning and normal user activity workload. This section

summarizes those results and provides few advice in designing hypervisor side read caching and write deduplication optimizations for VDI environment.

Table 5.9 shows peak and average I/O traffic ratio for following cache combinations:

1. No Cache (NC)
2. Location-addressed cache and Fingerprint Cache for Inline Write Deduplication (LOC-WD)
3. Content addressed cache and Fingerprint Cache for Inline Write Deduplication (CAS-WD)
4. Local store for Shared Read-only disk and Fingerprint Cache for Inline Write Deduplication (REP-WD)
5. Local store for Shared Read-only disk, Location-addressed cache and Fingerprint Cache for Inline Write Deduplication (RLOC-WD)
6. Local store for Shared Read-only disk, Content addressed cache and Fingerprint Cache for Inline Write Deduplication (RCAS-WD)

Cache size of location-addressed cache, content-addressed cache and fingeprint cache is 16GB, and uses LRU replacement policy. Left most column specifies one of the following workloads: full clones provisioning (FC-P), full clones normal user activity (FC-N), linked clones provisioning (LC-P), and linked clones nomal user activity (LC-N).

Finally, we summarize our observation of total traffic ratio for the above optimizations and different desktop/workload configurations in figure 5.20. The total traffic ratio bar in this summary graph also shows the percentage of read and write traffic separately. Based upon the simulation results, we make the following observations:

1. Provisioning traffic in VDI environment is write-heavy for full clones and read-heavy for linked clones as shown in table 5.3 and 5.6 respectively.

	NC		LOC-WD		CAS-WD		REP-WD		RLOC-WD		RCAS-WD	
	Peak	Avg	Peak	Avg	Peak	Avg	Peak	Avg	Peak	Avg	Peak	Avg
FC - P	1.00	1.00	0.68	0.18	0.68	0.14	-	-	-	-	-	-
FC - N	1.00	1.00	0.95	0.68	0.86	0.43	-	-	-	-	-	-
LC - P	1.00	1.00	1.00	0.42	0.84	0.30	1.00	0.51	1.00	0.40	0.84	0.30
LC - N	1.00	1.00	0.82	0.34	0.75	0.29	0.90	0.48	0.76	0.32	0.73	0.28

Table 5.9: Peak and Average Traffic Ratio with Cache

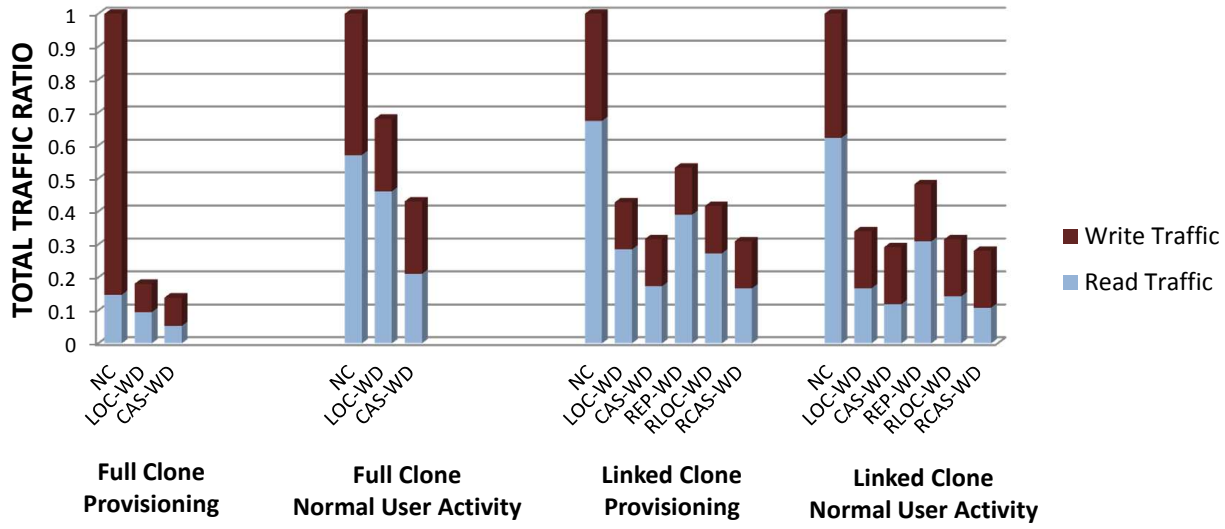


Figure 5.20: Total Traffic Ratio for different cache simulations and desktop/workload configurations

2. Location-addressed cache is a poor cache choice for full clones. Content addressed cache achieved a high read traffic reduction of 64% on bursty provisioning workload and 63% on normal user activity workload for full clones.

3. Unlike full clones, the advantage of content-addressed caches over location-addressed caches is significantly lower with linked clones.

4. The technique of storing the read-only virtual disk on local media on hypervisor side is a simple and cost effective optimization technique. However, its advantage was significantly lower than location-addressed or content-addressed read cache.

5. In case of linked clones, using a local store of read-only disk combined with either of content-addressed cache or location-addressed cache results in maximum reduction in the read traffic.

6. Fingerprint cache with inline write deduplication eliminates upto 50% of write traffic and this observation is valid for both full clones and linked clones on normal VDI activity workload. This optimization technique achieved 90% write traffic reduction while provisioning 30 full clone virtual desktops. Hence, inline write deduplication can reduce write traffic significantly, irrespective of type of virtual desktop used and the kind of VDI workload.

CHAPTER 6

Future Work

This work has laid a foundation for us to build a complete system which uses local storage on virtualized servers to cache data. We plan to re-use the cache directory implementation from our simulation module in addition to implementing a module which will handle I/O to flat files on local disks. Hopefully, with this work we have a clear understanding of which cache combination works well with kind of workload and type of virtual desktop. Hence, we leverage this knowledge in designing hypervisor side caches. We plan to implement this cache in Xen and compare the performance numbers from trace driven simulations and actual implementation.

When VDI deployments use multiple hypervisor hosts configured in a clustered environment, these caching techniques can be easily extended to cooperate. This kind of shared cooperative caching can bring immense benefits when the cluster running virtual desktops are configured on high bandwidth local area networks, and shared storage are used on wide area environments [DWA94, Tol03]. Current virtual infrastructure deployments have multiple virtualized servers configured in a cluster, for better load balancing and high availability. Hence, our next step is to simulate cooperating caches for furthermore I/O traffic reduction between hypervisor and storage servers. Comparing cooperative caches as they are implemented as a shared cache between multiple hosts or as a private cache per host will help choose the right methodology.

CHAPTER 7

Conclusion

This work has analyzed a variety of caching techniques for a unique VDI workload. This trace-driven simulation study can prove as a very useful pointer to file system developers who want to design hypervisor side software caches for VDI environment. In this work, synthetic VDI workload generator has been used to simulate VDI environments and I/O traces are collected on NFS server. This comprehensive study of hypervisor side caching involved design and implementation of I/O trace collection module on modified Tintri NFS server and different cache simulation modules. Each cache simulation module take I/O trace as the input and perform analysis. All the cache simulations in no way violate the consistency guarantees ensured by NFS protocol.

The results from the evaluation chapter demonstrates the power of hypervisor side caching in virtual desktop environments, especially reducing the overall I/O traffic over the network between hypervisor and shared storage. One of the observations prove that, cache technique selection is sometimes dependent on type of virtual desktop used. When different cache techniques used result in comparable performance numbers, the memory overhead in managing the cache is a good metric for cache selection.

References

[AAI09] E. Anderson, M. Arlitt, C. Morrey III, and A. Veitch. "DataSeries: an efficient, flexible data format for structured serial data." *ACM SIGOPS Operating Systems Review (OSR) HPL Special issue*, pp. 70–75, January 2009.

[Bad98] G. Badros. "An Enhanced Disk-Caching NFS Implementation for Linux." *1998 Linux Exposition*, April 1998. `badros.com/greg/doc/enhanced-linux-nfs-client`.

[BDF03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the art of virtualization." In *SOSP*, pp. 164–177, October 2003.

[DWA94] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance." In *OSDI*, pp. 267–280, Novermber 1994.

[FB96] K. Froese and R. Bunt. "The Effect of Client Caching on File Server Workloads." In *HICSS (1)*, pp. 150–159, January 1996.

[GNT07] A. Gulati, M. Naik, and R. Tewari. "Nache: Design and Implementation of a Caching Proxy for NFSv4." In *FAST*, pp. 199–214, February 2007.

[MAC08] D. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. Feeley, N. Hutchinson, and A. Warfield. "Parallax: virtual disks for virtual machines." In *EuroSys*, pp. 41–54, April 2008.

[Man09] P. Manning. "Best Practices for running VMware vSphere on Network Attached Storage." Technical report, VMware, January 2009. `vmware.com/files/pdf/VMware_NFS_BestPractices_WP_EN.pdf`.

[MCM01] A. Muthitacharoen, B. Chen, and D. Mazières. "A Low-Bandwidth Network File System." In *SOSP*, pp. 174–187, October 2001.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. "Caching in the Sprite Network File System." *ACM Trans. Comput. Syst.*, **6**(1):134–154, February 1988.

[PS09] C. Pettey and H. Stevens. "Gartner says Worldwide Hosted Virtual Desktop Market to Surpass 65 Billion Dollars in 2013." Technical report, Gartner, March 2009. `gartner.com/it/page.jsp?id=920814`.

[SBG12] K. Srinivasan, T. Bisson, G. Goodson, and Inc. K. Voruganti, NetApp. "iDedup: Latency-aware, Inline Data Deduplication for Primary Storage." In *FAST*, February 2012.

[SHN85] M Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. "The ITC Distributed File System: Principles and Design." In *SOSP*, pp. 35–50, December 1985.

[SMW11] M. Shamma, D. Meyer, J. Wires, M. Ivanova, N. Hutchinson, and A. Warfield. "Capo: Recapitulating Storage for Virtual Desktops." In *FAST*, pp. 31–45, February 2011.

[SS02] Q. Sean and D. Sean. "Venti: A New Approach to Archival Storage." In *FAST*, pp. 89–101, February 2002.

[Tol03] N. Tolia. *"CASPER: A Recipe Based File System."*. Master's thesis, Carnegie Mellon University, 2003.

[Und] "VMware Workstation 5.5, Understanding Clones." `vmware.com/support/ws55/doc/ws_clone_overview.html`.

[VAB04] B. Varanasi, R. Arpaci-Dusseau, and P. Barford. *"NFS Proxies: Design, Implementation and Applications."*. Master's thesis, University of Wisconsin, Madison, 2004.

[Vag10] S. Vaghani. "Virtual machine file system." *SIGOPS Oper. Syst. Rev.*, **44**:57–70, December 2010.

[Vie11] "VMware View Planner is a product of VMware, Inc.", January 2011. `communities.vmware.com/docs/DOC-15578`.

[Wal02] C. Waldspurger. "Memory Resource Management in VMware ESX Server." In *OSDI*, December 2002.