Universidade do Minho
Escola de Engenharia

João Tiago Medeiros Paulo

Efficient Storage of Data in Cloud Computing

Novembro 2009

João Tiago Medeiros Paulo **Efficient Storage of Data in Cloud Computing**

UMinho|2009

Universidade do Minho
Escola de Engenharia

João Tiago Medeiros Paulo

Efficient Storage of Data in Cloud Computing

Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Orlando Pereira

Novembro 2009

Acknowledgements

This thesis would have not been written without the help of several people, whom I would like to give my appreciation.

First of all, to my advisor, Prof. José Orlando Pereira, for his guidance that was essential to accomplish my master thesis. Also, To Prof. Rui Oliveira for our valuable conversations that came to be extremely important for the definition of the path of my work.

To Francisco Maia and Francisco Cruz for all the work and non work discussions that became extremely important for my thesis. To Nuno Carvalho for all his support and ideas. To Ricardo Vilaça, Miguel Matos, Bruno Costa and Ana Nunes for their help.

To Paula for helping me with the revision of my thesis document and for all the support, even in the most stressing times.

Finally, to my parents and my brother for supporting me throughout all my work and my life.

Finally, it must be mentioned that this work was partially supported by project "Pastramy: Persistent and highly Available Software TRansactional MemorY" (PTDC/EIA/72405/2006).

Resumo

A necessidade de ter dados críticos seguros e acessíveis através de vários locais tem-se tornado uma preocupação global, seja tratando-se de dados pessoais, organizacionais ou de aplicações. Assim, verificamos a emergência de serviços de armazenamento *online*. É também importante ter em conta o recente paradigma de "Cloud Computing", o qual acarreta novas ideias para a criação de serviços que permitem aos utilizadores armazenar os seus dados e executar as suas aplicações na "Cloud". Através de uma gestão inteligente e eficiente dos dados armazenados por estes serviços, é possível melhorar a qualidade do serviço oferecido e optimizar a utilização da infra-estrutura onde os serviços correm. Esta gestão torna-se ainda mais crítica e complexa quando a infra-estrutura é composta por milhares de servidores, correndo várias máquinas virtuais e partilhando o mesmo sistema de armazenamento de dados. A eliminação de dados redundantes pode ser utilizada para simplificar e optimizar esta gestão.

Esta tese apresenta uma solução para detectar e eliminar dados duplicados entre máquinas virtuais que correm no mesmo servidor e escrevem os seus discos virtuais para o mesmo sistema de armazenamento de dados. É também apresentado e avaliado um protótipo que implementa esta solução. Finalmente, descreve-se um estudo sobre a eficiência de dois métodos usados para eliminar dados duplicados num conjunto de dados pessoais.

Abstract

Keeping critical data safe and accessible from several locations has become a global preoccupation, either being this data personal, organizational or from applications. As a consequence of this issue, we verify the emergence of on-line storage services. In addition, there is the new paradigm of Cloud Computing, which brings new ideas to build services that allow users to store their data and run their applications in the "Cloud". By doing a smart and efficient management of these services' storage, it is possible to improve the quality of service offered, as well as to optimize the usage of the infrastructure where the services run. This management is even more critical and complex when the infrastructure is composed by thousand of nodes running several virtual machines and sharing the same storage. The elimination of redundant data at these services' storage can be used to simplify and enhance this management.

This dissertation presents a solution to detect and eliminate duplicated data between virtual machines that run on the same physical host and write their virtual disks' data to a shared storage. A prototype that implements this solution is introduced and evaluated. Finally, a study that compares the efficiency of two different approaches used to eliminate redundant data in a personal data set is described.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Dependable backup services are increasingly important to enterprises but also to common users that want to keep their personal files safe. A traditional approach, for common users, is to have a copy of all their files in an external hard drive. One example of such system is Time Machine [50]. For enterprises the solution requires having a larger storage and a more complex solution to backup their critical data. For some enterprises, data is so important that several backup copies must be kept in different physical locations in order to avoid losing it in case of natural catastrophes.

Another important aspect, for both enterprises and common users, is the need of accessing their data remotely from different places. For this purpose, the web is a good solution, having in mind how easy is to insert and retrieve information of any kind from it.

This explains the emergence and success of on-line backup services like Dropbox [33], Box.net [30], RapidShare [48] and Google Docs [40], that allow clients to have their data safe in the web. These services are more than just simple data archives. Some of them support other features, like collaborative work, versioning, online editing and synchronization of user's data between several devices. With these new functionalities, a storage service that is intended to store and retrieve data efficiently is necessary. In classical archival

systems this was not needed to be contemplated. As expected, all these services have limits to the amount of data each user can backup and, therefore, clients must pay a fee to expand these limits.

Besides online backup services, the Cloud Computing model [1] allows users to shift their data and applications to the web and run them without the obligation of having their own physical infrastructure. With this new paradigm, new services were created with different goals [5].

One type of service provided gives the client the possibility of running applications in the cloud's infrastructure. Services like Amazon EC2 [25] and Google App Engine [39] provide this type of service. In Amazon EC2, the client must provide a customized virtual machine image with her application and deploy it into the cloud infrastructure. For Google service, the application must be ported to Python [47] or Java [7] and respect Google App Engine API.

Another kind of service has the goal of providing remote storage. This is the case of Amazon S3 [26], Amazon EBS [24] and Google App Engine Datastore [38]. These services can be combined with other cloud services or can be used as a remote storage directly, with the exception of Amazon EBS that was designed to be used with Amazon EC2. One example of this combination of services is present in Dropbox that uses Amazon S3 [28] to store its clients' data and Amazon EC2 to run their server application.

There are also other cloud services with different functionalities, like Amazon SimpleDB [27], that enable the client to store and query data with the advantage of retrieving only the information needed and doing it more quickly than with services like S3. This service is intended to store small data sets, but can be used with Amazon S3 in order to have Simple DB functionality with larger data sets.

All the cloud services described above allow the client to stop concerning with problems, such as:

Dependability. Clients' applications and data stored in these services must be accessible 24×24 hours a day and seven days per week. Besides that, clients' data is stored redundantly in several data centres in different geographical locations.

Elasticity. Clients have the illusion of having unlimited resources. For example, in Amazon EC2, when clients' applications need to scale, this can be done in a few minutes by running an additional virtual machine. Clients also have the possibility of starting with few resources, buying more resources only when they are necessary.

Another important aspect is the use of virtualization [9, 20] technology by cloud services [1, 54]. Virtual machines (VMs) allow these services to have increased flexibility in terms of deployment and redeployment of applications in the cloud. Deploying a new virtual machine or redeploying it in another physical server is faster and simpler than deploying a new physical server. Virtualization also allows having more control over cloud resources, like disk, network and computation power. Therefore, these resources can be distributed accordingly to the applications' needs. The use of virtual machines is a key aspect to achieve the elasticity property. Virtual machine's isolation property assures us that a failure in one VM does not affect the other VMs running in the same physical server.

Cloud infrastructure is therefore composed by several data centres. In each data centre there are several physical nodes running a certain number of virtual machines. The cloud storage has to be large enough to accommodate these virtual machines images and the clients' data that is stored remotely.

Both cloud services and on-line backup services will have a large amount of data that needs to be stored persistently and, as a consequence, a huge amount of duplicated data is expected to be found.

In the case of on-line backup services, many users will have dupli-

cated files, like music, videos, text files, etc. As it concerns enterprises that use backup services, identical data will probably be found between the employees' files. Yet, another source of duplicated data that may exist in some enterprises storage is the e-mail. There are studies, which show us that e-mail data sets have more than 32% of their blocks duplicated [8].

Regarding cloud computing provider's storage, duplicated data is expected to be found between the several virtual machines images and between data stored remotely. In fact, if we recall that Dropbox uses Amazon S3 as backend, Amazon's storage will have all that duplicated data mentioned above for on-line backup applications. Additionally, there is the possibility to run applications that use databases in the cloud, which may need to create several replicas in order to process a large number of concurrent read requests. This will further increase the number of duplicated data.

If some of this duplicated data is eliminated, the storage's space in use can be reduced and a better service can be achieved by supporting more clients without having to add extra storage resources or by providing a better service to each client. This storage space reduction also allows these services to make a more efficient and simple management of their data.

A certain level of redundancy is always needed to have a service that is resilient to failures and has efficient access to data from several locations. For this purpose, data must be replicated amongst several nodes and these nodes should be located in several geographical locations.

Usually, systems that provide solutions to reduce data duplication are known for indexing the storage's data in order to share data with the same content. The elimination of redundant data is known as *deduplication*. Deduplication can also be used to improve bandwidth usage for remote storage services. If the storage's data is indexed, then it is possible to choose what data really needs to be transmitted

to the storage server and what data is already there. The bandwidth cost and the data upload speed is described as one of the main obstacles for the growth of cloud computing [1].

1.1 Problem Statement

Effective deduplication in a cloud computing scenario raises however a number of challenges.

First, there are architecture challenges. In this scenario, at least one distinct VM is running for each client's application. This means that data is spread by several VMs virtual disks. Additionally, groups of VMs are running in distinct physical machines. Finally, there is the necessity of keeping data deduplication process transparent to the VMs and the applications running on them.

Second, there are algorithm challenges. An efficient method to detect modified data and to share identical data is needed. This method must use metadata to compare the modified data with the storage's data in order to share it. Metadata's size is needed to have in account to achieve an efficient deduplication algorithm. To detect modified data without having to scan all the storage, a method that intercepts I/O requests to the disk is also necessary. This approach reduces the CPU usage but can introduce significant overhead in the I/O requests. This overhead value must be as low as possible.

1.2 Objectives

The main goal of this dissertation is to show how deduplication can be achieved in a virtualized system[1], towards finding and eliminating redundant data in the context of cloud computing services.

[1]System where there is one physical server running several virtual machines that map their virtual disks into a shared storage.

A second objective is to evaluate the impact of deduplication in personal data towards demonstrating the usefulness of the proposed solution.

1.3 Contributions

As the first contribution, we present an approach to detect and eliminate redundant data in a server where several virtual machines are running. This server's virtual machines store their images in a shared storage.

As the second contribution, we present both our prototype, working with Xen [2], that implements the approach referred above and the results of our prototype's evaluation in terms of space saved and overhead generated.

Finally, we present a study and its results for the redundant data found in a personal files' data set. In this study, we compare two distinct methods: the whole file approach, which find files with the same content, and the fixed block size approach, which finds blocks, with a fixed size, that have the same content.

1.4 Dissertation Outline

This dissertation is organized as follows. In Chapter 2, the state of the art for data deduplication and Cloud services is described. An overview about remote storage services and virtualization systems, related with our work, is also presented. Chapter 3 introduces a study about the efficiency of two distinct deduplication methods in a data set that contains personal data. In Chapter 4, our architecture to find duplicated data among virtual disks of VMs that are running in the same physical host is presented. Chapter 5 describes the implementation of our architecture and introduces new mecha-

nisms to improve our prototype's performance. In Chapter 6, the prototype is evaluated by two different benchmarks and the results are discussed. Finally, in Chapter 7 we conclude our work by describing the objectives achieved and, in Chapter 8, we present some ideas that would be interesting for future research in this area.

Chapter 2

Related Work

This chapter presents the state of the art for: data redundancy detection and elimination methods, storage and backup services, Cloud Computing services and virtualization scenarios that are related with our work.

2.1 Finding and Eliminating Duplicated Data

There are several methods to find redundant data. The first of these methods is the whole file content hashing [17] that calculates a hash sum of the entire file's content. If two files have the same hash value then they have identical contents.

Another option is the fixed size block method [8, 11, 17], where duplicates are found at the block level. The process is identical to the one from the whole file approach, but instead of calculating hash sums for the entire file's content they are calculated for file's blocks with a fixed size.

A third method uses chunking and Rabin fingerprints [4, 8, 11, 17]. Chunks are also defined by content, but their bounds are not restricted to a fixed size, like in the fixed size block approach. A

sliding window moves through the file's content and calculates fingerprints for a chunk[1]. When a predefined pattern is found, the chunk boundaries are marked and its signature is calculated.

The main difference among the fixed size block method and the chunking method is visible when a file is modified. The chunk method only needs to recalculate the signature for the chunks where modifications were made. The fixed block size approach needs to recalculate the signature for the blocks where modifications were made and for all the subsequent blocks of the same file.

These methods focus only on detecting data that has exactly the same content. The super-fingerprint method [11, 12] can be used to find similar data. A super-fingerprint is a group of fingerprints belonging to different parts of the same file. If several super-fingerprints of a file are calculated, the resemblance among files is given by the number of super-fingerprints that match.

Delta encoding [11] is used to reduce redundancy between similar files. This method generates a delta file containing differences between the files, which allow keeping only one file and the delta file needed to rebuild the other file. This method does not find similar files.

Compression [8, 11] is a well known form of reducing duplicated information. This technique can be used to compress a single file, and only reduce redundant data from that file, or to compress several files, and reduce redundancy across those files. Most of the approaches that use this technique only find redundancy among close files when compressing several files.

REBL [11] uses compression, chunking and delta-compression of the chunks. First, chunks are calculated and hashed in order to find duplicated data and to eliminate it. Then, similar chunks are found with super-fingerprints and are delta-compressed. Finally, all the

[1] This sliding window method can be used because fingerprints are distributive over addition.

chunks that were neither delta-compressed nor eliminated in the chunking process are compressed.

All the methods described above present ways to find or remove redundant data, with the exception of REBL and compression that present both methods. Despite that, these methods present ways of saving space, but they are not concerned with the actual process of sharing data, from different users, and the necessity of maintaining this data consistent when clients need to access it.

The Rsync [23] approach is used to reduce bandwidth usage in the specific case where we want to update two files (in distinct computers) to have the same content. In this solution, the receiver splits the file into blocks and calculates two hash functions for each block. The sender receives the blocks' hashes and compares them with its file blocks' hashes. This way, Rsync only sends data from blocks that are missing at the receiver and information to reconstruct the rest of the file with blocks that the receiver already has.

All the methods described above were tested in a scenario where the detection of duplicated data was accomplished with a scan approach[2]. With this approach, the opportunities for sharing data are detected by scanning all the storage. The scan approach fits well for the purpose of the studies described in this section. However, for our specific scenario where data will be modified constantly, the scan approach will introduce significant overhead in computational power. This happens because this approach needs to scan the storage several times to check for modified data and share it. Having this specific scenario in mind, a dynamic approach that detects modified data, suits better on our work. As an example, this can be achieved by intercepting I/O write requests to the storage.

This change of paradigm to detect duplicated data also changes the applicability of the methods described above. Compression method is not appropriated for a scenario where several files are accessed

[2]It may also be referred as static approach.

and updated frequently. Chunking method with rabin fingerprints also introduces unnecessary complexity for this new scenario. As stated above, the main advantage of this method, when compared to fixed size block approach, is visible when a file is modified in a specific part. With the fixed size block approach, we may have to calculate signatures for blocks that were not modified. However, with chunking, we only calculate the signature for the block that was updated. In our scenario, this is not relevant because I/O requests are intercepted dynamically. If we assume that these requests have a fixed size for the data that is written, then we can use the fixed size block approach for that specific size and only calculate signatures for the data that was updated. Besides that, if we compare the values shown in [17] for the fixed size block and chunking approaches, in terms of space saved by eliminating redundancy, we see that both methods have identical results. In fact, for interactive contexts like ours, where hash sums are only calculated for the data that really was updated, the fixed size block method offers higher processing rates.

2.2 Remote Backup/Storage Services

LBFS [15] is a network file system designed to reduce bandwidth when transmitting files to the server. Files are divided into content defined chunks using Rabin fingerprints. For each chunk, a SHA-1 [53] digest is calculated and stored locally. Before transmitting a file, SHA-1 signatures of its chunks are sent and compared with the ones available at the receiver. This way, clients only send the chunks that are missing at the server. LBFS approach uses duplicate detection to reduce bandwidth.

Pastiche [6] provides a solution to reduce storage redundancy in a backup peer-to-peer system that resembles LBFS. Like LBFS, Pastiche uses content based indexing. When a peer contacts another to

backup its data, chunks signatures are sent first. This way, Pastiche only stores chunks that do not exist already at the receiver. The main difference from LBFS is that Pastiche's data is stored on peers as chunks. By storing data this way, the sharing process is simplified because an additional table relating data stored with chunks is not needed to keep. These chunks also contain information about the peers sharing them for garbage collection. This solution also introduces the idea of choosing peers to hold backup by their data proximity. This way, when one node wants to backup its data; it calculates signatures of some of its chunks and compares them with the signatures calculated from other peers. This approach allows finding buddies that will probably be more suitable to keep backups of that node in terms of storage space and bandwidth saved.

The Venti [18] archival storage system is aimed at keeping data that has a write-once policy. This way, data is never modified neither deleted from the storage. Venti presents a simple interface to read and write blocks of several sizes. Backup applications can use Venti as backend with this API. In Venti, blocks are identified by their hash, which allows eliminating duplicated data at the storage. The index that keeps all blocks signatures is implemented with a disk resident hash, which represents a performance penalty because every request must use it.

Deep Store [57] is an archival storage system aiming scalability, reliability and efficiency. The storage service works at the file abstraction level and has a simple interface that allows the clients to store files, retrieve files and delete them. Deep Store is intended to run in several nodes and each node contains its own processor, memory and disk. Each of these nodes runs several processes in order to provide the archival storage service. This solution also presents a study about three different ways to eliminate redundancy across their storage service. The methods described are the whole file content, delta-compression and the chunking method for file's parts.

All the solutions described above can give us the idea of a storage service's infrastructure and architecture. However, they do not address the scenario where virtualization is used on the physical nodes.

Besides the systems described above, we also have commercial products that provide storage systems using data deduplication techniques. This is the case of EMC Avamar [34], Exagrid [35], IBM ProtectTire [41] and DataDomain [31]. All these services provide a server that bundles all the software and hardware, including the storage hardware, necessary to be used as a remote storage service that can be easily deployed in the client's infrastructure.

IBM ProtectTire and DataDomain also present a gateway solution that can be used with different types of external storage. This is useful for clients that use a third party external storage system.

With the exception of Avamar and Exagrid, all the solutions use an in-line approach to detect duplicated data. Through this approach, duplicated data is found and shared before storing the data. This approach spares more disk space, but has the disadvantage of being slower because digests must be calculated and compared before storing the data on disk. EMC Avamar detects duplicated data at the client side, which reduces bandwidth usage. In contrast, Exagrid uses a post-processing approach. With this technique, the data is stored immediately and the sharing process only occurs after the operation is over. This technique is faster when compared to the in-line deduplication, but it consumes more storage space.

All these solutions are intended as backup solutions. This way, they are not concerned with the need to store and retrieve data efficiently. Besides that, their implementation details are not known because they are commercial products. These approaches are not intended to store virtual machines images. The exception is EMC Avamar that has specific software used to store VMware virtual machines images, but, once again, this solution is intended for the backup scenario, which is not expected to have a huge load of read and

write requests to the VMs images.

2.3 Cloud Services

Cloud Computing storage services like Amazon S3 [26], Amazon EBS [24], Amazon SimpleDB [27] and Google App Engine Datastore [38] provide remote storage services for their clients. Amazon S3 offers a storage service with a simple API that allows clients to store, retrieve and delete objects from different namespaces, called buckets, that are also managed by the client. In fact, this service's API resembles the Deep Store interface described in section 2.2. Amazon SimpleDB and Google App Engine Datastore allow clients to store data and to be able to perform queries on it.

Google App Engine [39] and Amazon EC2 [25] allow clients to run their applications in these services' infrastructure. Google App Engine forces the users to write their applications in Python or Java and to respect this service APIs. These applications run in Google infrastructure and, if needed to scale, this is done automatically if the user pays the additional load. Amazon EC2 service has a different approach. In this service, the user customizes a virtual machine image with her application and deploys that virtual machine into Amazon infrastructure. If the application needs to scale, the client can run an additional virtual machine instance with her application image in minutes. This is not done automatically, so the user must explicitly start the instances she wants to use. If the client wants to keep her virtual machine images state stored persistently, then she can use the S3 service or the EBS service. Amazon EBS provides a block level storage volume that can be attached to an Amazon EC2 instance. These volumes persist even if the instance is terminated or fails. The main benefit of using EBS, instead of S3, is that this service presents a better solution in terms of efficiency and simplicity for applications that need to use a raw block storage, file system or

databases. Using S3 for an application that uses a database can also be achieved, but this solution has some restraints attached to it [3].

In the context of our work, the combination of Amazon EC2 and EBS is very interesting because it represents the precise scenario where our approach performs deduplication. With this combination, we have virtual machines writing to their virtual disks, which will probably be mapped to a common storage. Duplicated data can be found inside the virtual disks and across them.

The cloud services described above do not present any public information about their infrastructure and architecture. This way, we cannot know precisely their details. However, the information above shows us the importance of virtualization in these services and also gives us some hints about the type of information that needs to be stored.

Eucalyptus [16] is a project that presents a framework to implement cloud computing services on top of private clusters. Eucalyptus current version provides two types of services: one resembling Amazon EC2; and another resembling Amazon S3. In fact, the APIs provided by Eucalyptus are identical to the Amazon APIs. Regarding Eucalyptus design, each physical node in the clusters has a node controller that has the responsibility of managing the VM instances and the resources of its physical node. More specifically, this node is able to start and stop instances as well as to provide information about physical node resources and virtual machines instances running on it. Currently, there are only supported virtual machines that run atop the Xen hypervisor.

A cluster controller is used to manage several node controllers. This controller can be used to schedule incoming requests to a specific node controller and gather resources information about a set of node controllers.

There is also a storage controller (Walrus) that is used by the client to store data into the cloud. This is done via an interface identical

to the Amazon S3's interface. Walrus is also used to store virtual machines images. When node controllers need to start virtual machines, they can request the virtual machines images from Walrus.

The work described in this section does not address any deduplication approach. However, this work is important to understand design details of cloud services and to understand the type of system where we want to eliminate duplicated data.

2.4 Virtualization Scenarios

Parallax [13, 56] presents a solution to reduce redundancy among persistent snapshots[3] of virtual machines images and their current image. This solution is intended for a cluster where there are several nodes and each node can run more than one VM. All these nodes have access to a shared storage (block device) where they keep their VMs' disk images (VDIs) and their snapshots. For each physical node, there is an instance of parallax running that controls all I/O requests from VMs that are also running on that same node. When a snapshot is taken, its blocks are shared with the current image; when a request to write to a block that is shared is intercepted by parallax, a copy-on-write operation is performed to keep the block content consistent.

Blktap [29] is used to implement the copy-on-write mechanism and to provide virtual disks for each VM by intercepting block's write and read requests from VMs. A radix tree is used to map virtual block addresses[4] to physical block addresses[5]. Every VDI has its own radix tree and these trees are also stored as metadata in the shared storage.

[3] Read only copies.

[4] Addresses used by the VMs to request blocks.

[5] Addresses that point to the location at the physical device where blocks are stored.

By having copy-on-write techniques, each Parallax instance needs to be able to reclaim free blocks to execute this operation. To solve this problem, a lock mechanism is used to ask for free blocks. In order to avoid using the distributed lock mechanism every time a Parallax instance needs space to write VDI's data, an extent mechanism is introduced. Blockstore[6] is divided into fixed size extents. These extents are typed. Data extents hold VDIs' blocks and metadata extents hold information, such as radix trees, and are locked by Parallax instances in order to write to the Blockstore. There is a special extent that holds information about shared storage's size, extent's size, extent's type and their lock holder.

New VDIs can be created from snapshots. These VDIs will also share duplicated blocks with the snapshots used to create them. However, data from different VDIs without a common ancestral is not shared.

Satori [14] and VMware ESX Server [55] present two approaches that find and eliminate duplicates in memory instead of disk. Both approaches are intended for the scenario where there are several virtual machines running on the same physical host, and the objective is to share VMs' memory pages. In VMware ESX, memory is shared by doing a scan to all the VMs' memory pages and by calculating their hashes and storing them in a Hash Table. This scan is done periodically and all the pages that are shared are marked as copy-on-write. VMware ESX also introduces the Ballooning mechanism that is used when the server needs to reclaim free pages from their VMs.

Satori presents a solution that does not use a scan approach to find duplicated memory pages. Their approach modifies the VM's virtual disk implementation in order to intercept read I/O requests made by the VM to its disk. When a block read request is intercepted, its content is hashed and compared with the other pages digests to see if the page is duplicated. If the page is duplicated then it is shared

[6]Name given to the shared storage.

and marked as copy-on-write. The Repayment FIFO mechanism is introduced to provide free pages when a page share is broken. This FIFO is a list of volatile pages that the VM's operating system is willing to relinquish at any time.

One of the advantages presented in Satori approach is the possibility of detecting short-lived sharing opportunities. In other approaches, like VMware ESX Server, that use the scan method, some of these opportunities are not processed.

Summary

In this chapter, two approaches that use virtualization and intercept I/O requests from VMs are presented. Neither of them has the purpose of eliminating duplicated data at a storage that is shared among several VMs. Nevertheless, these solutions are important for a better understanding of how dynamic detection of modified data can be achieved, for our specific scenario. By intercepting I/O write requests from VMs, modified data can be detected. This solution is more efficient than the one that uses a static approach, which must scan either the VMs' virtual disks or the storage, to find data that has changed and share it. Nevertheless, dynamic approaches can introduce overhead in the I/O requests to the storage. This happens when write requests are intercepted and its completion is delayed because the sharing process is being executed.

Chapter 3

Redundancy Study

This chapter presents a study about the redundancy found in a personal data set. The viability of two different methods, which find redundant data, is discussed in terms of space saved and in terms of space used by metadata. This metadata is necessary to share identical data in a scenario resembling the one we address.

3.1 Redundancy Detection

This section presents our study about duplicated data found in GSD's[1] data set. This data set contains 1,676,046 personal files associated with research projects from GSD and has a size of 108.11 GB. We choose it because its content is expected to resemble the one found in services like Dropbox, where personal data from several users is stored. On the one hand, we know that this data is slightly more related because personal files belong to researchers that have projects in common. On the other hand, the files that everyone possesses have few copies when compared to Dropbox's data set.

To detect duplicated data in our study's data set two methods were compared:

[1] Distributed Systems Group is part of the Department of Informatics of the Universidade do Minho.

Whole file - In this approach, duplicated data is found at the file level. This is done by calculating each file's SHA-1 [53] digest and storing it in a table. Each collision in the table shows that files are duplicated.

Fixed size block - This approach is similar to the whole file method, but finds redundancy at the block level. This way blocks digests are used instead of files digests. In this method the block's size is fixed.

GSD's data set was used to test the applicability of these two methods in our work. Freedup [36] was used to detect duplicated files. Freedup is an application that detects files with the same content inside a directory and its subdirectories. Files only need to have the same content to match and file names do not need to be identical.

An application was written, in C [10], to detect duplicated blocks. This application receives as arguments the path for the directory where duplicated data will be searched and the value to be used for the block's size. The application reads regular files' contents located inside the directory and its subdirectories, and computes a SHA-1 digest for each file's block. These hashes are stored in a hash table to detect collisions. This algorithm uses the fixed size block method but, when it reaches the end of the file and the last block's size is inferior to the default size, the hash for that block is calculated anyway.

Both these applications have as output a log file with information about duplicates found. Logs are analyzed and results extracted using a python script. These results are useful to choose the best technique for our approach, described in Section 4, and also to predict the duplicated data that will be found in a data set that contains several users' personal files.

Table 3.1 shows the number of items[2] scanned,the number of items

[2]Files for the whole file approach and blocks for the fixed size block approach.

Table 3.1: GSD's data set redundancy results: comparison between the whole file and the fixed size block approaches.

	File	Block 4KB	Block 8KB	Block 12KB
Files/Blocks Scanned	1,676,046	29,530,586	15,461,284	10,794,932
Files/Blocks Without Duplicates	536,418	22,610,369	11,581,715	7,912,071
Unique Files/Blocks	765,594	24,449,410	12,584,335	8,632,784
Files/Blocks to Eliminate	910,452	5,081,176	2,876,949	2,162,148
Space Saved	13.37 GB	16.75 GB	16.31 GB	16.01 GB
Duplicates per Regular File/Block	0.68	0.23	0.25	0.27
Duplicates per Duplicated File/Block	3.97	2.76	2.87	3.00

that do not have any duplicate, the number of unique items, the number of identical copies found that can be removed, the space saved by eliminating these copies, the average number of duplicates per item and the average number of copies that a duplicated item has.

Space saved improves by using the fixed size block method instead of the whole file. We ran the fixed size block algorithm for three different sizes, 4 KB, 8 KB and 12 KB, and the space saved is identical within these three options.

The best approach, in terms of space saved, is the 4KB fixed size block method. 15.5% of the total space is saved by using this approach. The whole file approach saves 12.4%. which represents the worst method. We can also witness that, in average, each file has less than one duplicate, but if the file is replicated, it possesses, in average, 4 identical copies. For blocks, the average results are inferior, but the relation is similar.

Figure 3.1 shows the number of files we can find in the GSD's data set with a certain number of duplicates. This figure shows values up to twenty five duplicates of the same file. Figure 3.2 shows the same information for files with more than twenty five duplicates. These figures are interesting to understand that the number of files with few duplicates is substantially higher and that this number

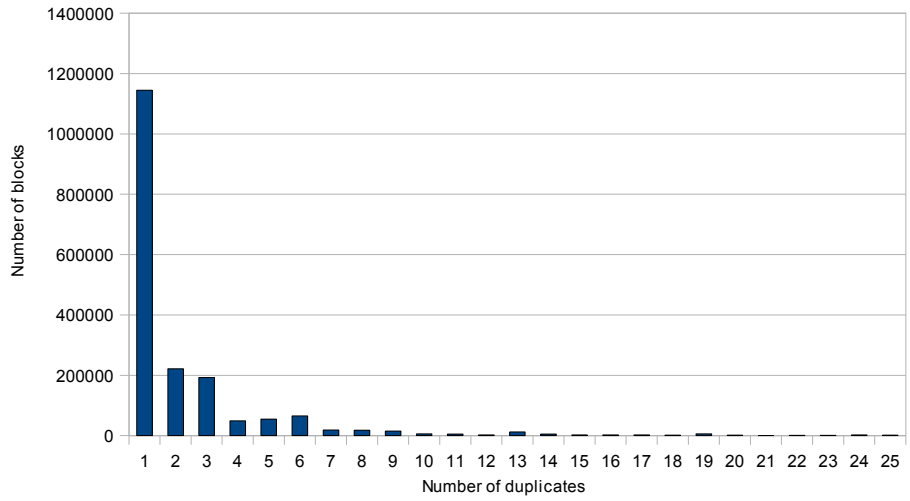

Figure 3.1: GSD's data set redundancy results: files with less than twenty five duplicates.

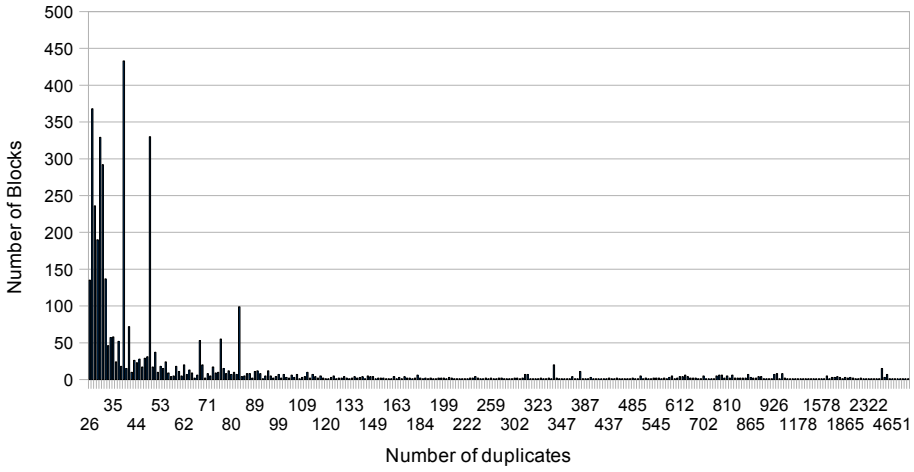

Figure 3.2: GSD's data set redundancy results: files with more than twenty five duplicates.

drastically decreases when the number of duplicates per file grows. We also think that is interesting to notice that there are files with more than two thousand duplicates. We only present here charts for the file results because the blocks charts are very similar and the conclusions we can extract from them are the same.

The results described above only show the efficiency of the methods in terms of space saved. However, in our work, we also need to have in account the space that will be occupied by metadata that will be used to hold information needed for sharing identical data.

3.2 Metadata

As said before, our main goal is to find duplicated data and share it in an interactive system where several VMs' virtual disks are mapped into a shared storage. Besides having an efficient method to find identical data, metadata holding information that will be used to share blocks with the same content and to keep VMs' disks I/O operations consistent is needed. In this section, we use our study's results to estimate metadata's size.

As a consequence of presenting virtual disks to VMs, a table (Translation table) that translates virtual addresses to physical addresses is needed. Virtual addresses are the the addresses that VMs request to read and write to their virtual disk. Physical addresses are the addresses that point to the location of the file/block in the physical storage. This table is needed because VMs will see virtual disks that will make the sharing process transparent to them. However, these disks will have shared content that will be mapped into the same physical address at the physical storage.

Translation table's size can be calculated using the following formula:

$$N_{fb} \times Phyadd$$

N_{fb} is the maximum number of items[3] that VMs' virtual disks can store and Phyadd is the size of the physical address. We are talking about virtual disks where redundancy elimination must be transparent. This way the number of items found at this level will be higher than the one found at the physical storage, where the virtual disks are mapped and where duplicated data is eliminated. In this formula, we do not contemplate virtual address's size. We assume that implicit values for virtual addresses can be used to reduce this overhead. As an example, an array where the index represents the virtual address and the value pointed by the index represents the physical address can be used. Other data structures like trees can also be used to reduce the overhead.

GSD's data set was used to estimate the size for this table. The size of physical address used was 64 bits and for the N_{fb} parameter we used the values described in the first row of Table 3.1, which represents the total items found at the data set before eliminating duplicated data. Table 3.2 shows the values obtained. This table shows the expected size of the Translation table for a mapping between virtual and physical addresses of files, 4 KB blocks, 8 KB blocks and 12 KB blocks.

As expected, the size of the Translation table for the file scenario is drastically lower when compared to the blocks' results. 8 KB and 12 KB blocks have identical values and the 4 KB block is the worst solution in terms of space consumed.

Besides the Translation table for each VM, another table that keeps mappings between blocks' hashes and their physical address at the storage is necessary. This table is essential to share blocks with the same content. This table possesses an additional column, for every physical address, that represents the number of virtual addresses pointing to it. This last column is useful when a shared block is modified by one VM and is necessary to know if the old physical

[3]Files or blocks.

Table 3.2: Translation table's size.

	Translation table's Size
4 KB	225.3 MB
8 KB	117.96 MB
12 KB	82.36 MB
File	12.79 MB

address can be freed or if that address is still being pointed by another virtual address. If the block is not being used by any virtual address, then it can be freed, otherwise the block's content cannot be modified.

For this table, that we call Hash-to-Address, we used the following formula to calculate the worst scenario possible in terms of space occupied. This is the scenario where the size of the table is proportional to the number of unique items at the global storage:

$$N_u \times (Hashs + Phyadd + Refcount)$$

Table 3.3: Worst scenario for Hash-to-Address table's size.

	Hash-to-address	Translation table	Total	Space Saved
4 KB	746.14 MB	225.3 MB	971.44 MB	15.8 GB
8 KB	384.04 MB	117.96 MB	502 MB	15.83 GB
12 KB	263.45 MB	82.36 MB	345.81 MB	15.67 GB
File	23.36 MB	12.79 MB	36.15 MB	13.33 GB

N_u is the number of different items at the physical storage, Hashs is the hash's size, Phyadd is the physical address's size and Refcount is the size occupied by the field representing the number of virtual addresses sharing a specific physical address.

Table 3.3 shows the result of this formula applied on the same data set described above. We have used 160 bits for hash's size, 64 bits

for physical address's size and 32 bits for representing the number of virtual addresses sharing the same physical address. We have used the values described in the third row of Table 3.1 for the Nu parameter. We choose 32 bits because we think that this value is sufficient to represent an upper bound for the number of addresses sharing the same block in a large dataset. By analyzing study's results for the 4KB blocks, there is one block with 225,165 duplicates that represents the higher value for this approach and for all the others. If we assume a scenario where addresses point to 4KB blocks and Refcount has 32 bits, we can have 16 TB[4] of virtual content pointing to the same physical address. This value is more than enough for the case we described above and for larger data sets. A worst scenario can occur if all the free blocks at the storage are being shared between the virtual machines and pointing to the same physical address. In such case, we think that the 16 TB value described above is an acceptable value. Other aspects, like keeping some redundancy in the storage, which we do not take into account, also reduce the number of virtual addresses pointing to the same physical address.

This table also shows values for the space occupied by the Translation table and the Hash-to-Address table and what impact these values have on the space saved that is described in Table 3.1. We see that the whole file approach continues to be the worst solution despite the small size occupied by its metadata. Among all block based approaches the results are very similar.

Figure 3.3 presents the percentage of space used for Translation and Hash-to-Address tables by using the worst case approach. Percentages are presented for the whole file approach and the fixed size block approach with sizes of 4 KB, 8 KB and 12 KB. For example, for a data set with 108.11 GB and using the 4 KB block approach, the tables' size will be $108.11 \times 0.0088 = 0.95GB$.

This scenario has its disadvantages when we think in the metadata's

[4] $(2^{32} \times 4) \div 1024^3 = 16TB$

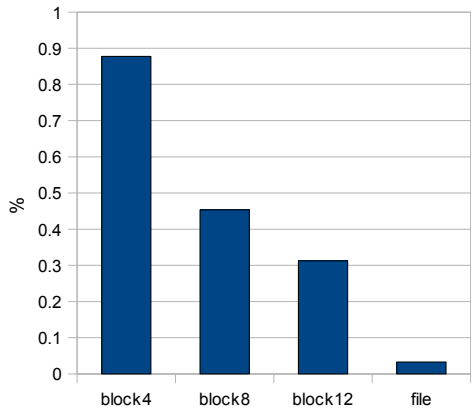

Figure 3.3: Percentage Space Used.

size. However, if we try to reduce Hash-to-Address table's size, there is always a trade off. For instance, we can think in the opposite scenario where the Hash-to-Address table is not used. In this new scenario, a digest must be calculated for each block at the storage, to find a match for the block that is being shared. As a matter of fact, this solution presents serious drawbacks because it generates a huge amount of CPU and I/O overhead. Therefore, there is always a trade off between metadata's size and computational overhead generated.

As stated before, some redundancy is necessary at the storage. In this dissertation, we do not contemplate this issue. However, the decision to maintain some redundancy can be easily achieved in two ways:

- One solution is to update an entry in the VM's Translation table to point to a shared physical address in some occasions only. This way redundancy is achieved automatically. This approach allows maintaining some redundancy without having to keep additional metadata information.

- Two extra columns can be added to the Hash-to-Address table for each duplicated block that is desirable to have in the system. One of the columns has the physical address at the global

storage and the other has information regarding virtual addresses that are sharing that physical address. This approach consumes space but, with it, we can control the number of duplicated data, the number of virtual addresses that are sharing each of the physical blocks and handle read requests that fail because the block they are pointing to is corrupted. With this last method, we have more control but we lose in terms of space occupied by metadata.

In this section, we do not describe all the metadata that is necessary for our approach to work. Our interest in this section is to present metadata's size that we can predict with this study and that must be used by all approaches that detect duplicated data in a scenario like our own. Other types of metadata will be discussed in next chapters.

Chapter 4

Deduplication Architecture

This chapter introduces an architecture to detect and share duplicated data from VMs running in the same physical host and sharing the same storage. Figure 4.1 describes this scenario.

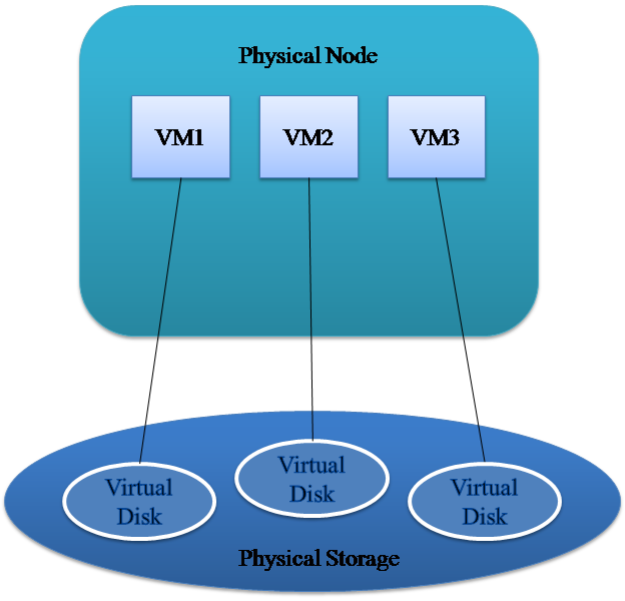

Figure 4.1: System description.

4.1 Overview

The architecture is composed by three modules. The I/O Interception module is used to intercept I/O requests from VMs and register the blocks that were written. For each VM there is an independent I/O Interception module. The Share module is responsible for processing the modified blocks and sharing them with other blocks with identical content. The Garbage Collector module is necessary to provide free blocks to the I/O interception module for copy-on-write (COW) operations and to collect free blocks that were freed by the Share module or by the COW requests. COW is necessary to keep I/O requests coherent. Figure 4.2 shows a description of the architecture.

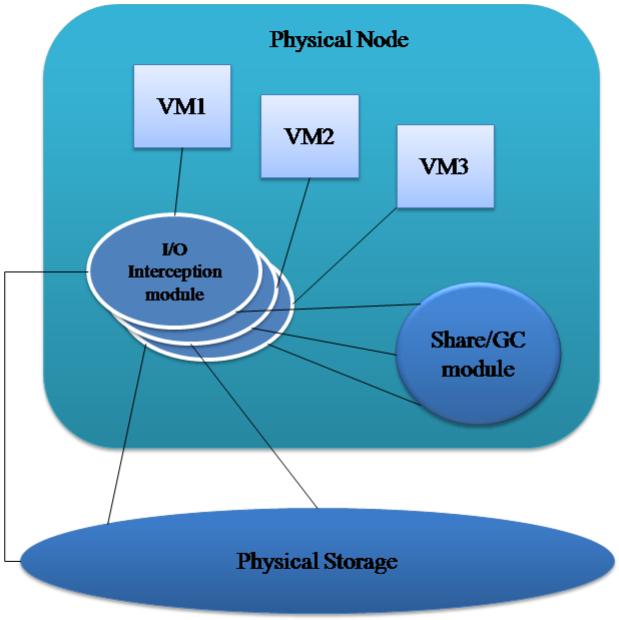

Figure 4.2: Architecture's diagram.

4.2 Intercepting I/O Requests

The I/O Interception module is responsible for intercepting I/O requests, from VMs to their virtual disks, at the block level. A Translation table is necessary for each VM. This table maps virtual addresses into physical addresses and is crucial to share physical blocks between several virtual addresses, being these virtual addresses from the same VM or from several VMs. Read and write I/O requests from the VMs require checking the Translation table, which is needed to find the location of the physical block necessary for processing the I/O operation. Listing 4.1 shows the pseudo code for the I/O Interception module.

```
1  read_request(virtual_address){
2
3    lock_mutex_page_table()
4    physical_address = check_for_address_in_page_table(virtual address)
5    unlock_mutex_page_table()
6
7    read_from_storage(physical_address)
8  }
9
10 write_request(virtual_address,content){
11
12   lock_mutex_page_table()
13   physical_address = check_for_physical_address_in_page_table(virtual address)
14   boolean = has_cow_mark(physical address)
15
16   if(boolen == FALSE){
17       unlock_mutex_page_table()
18       write_to_storage(physical_address,content)
19   }
20   else{
21       free_block_address = ask_for_free_block()
22       update_page_table_virtual_address_mapping(virtual_address,free_block_address)
23       unlock_mutex_page_table()
24
25       write_to_storage(free_block_address,content)
26
27       lock_mutex_free_cow_queue()
28       insert_address_free_cow_queue(physical_address)
29       unlock_mutex_free_cow_queue()
30   }
31
32   lock_mutex_dirty_addresses_table()
```

```
33    insert_address_dirty_addresses_table(virtual_address)
34    unlock_mutex_dirty_addresses_table()
35  }
```

Listing 4.1: Intercepting read and write requests.

In read requests, this module only needs to check the Translation table to redirect the read operation to the right physical address (line 4). As it concerns write requests, additional operations are needed to be performed. When a write requests is intercepted, the signature of the block's content is not calculated immediately to perform the sharing of that block. The write operation is processed normally (line 18 and 25) and the VM's virtual address pointing to that block is registered to be processed later (line 33). We refer to this table of virtual addresses as the Dirty Addresses table. All the registered virtual addresses will be processed in background by another module responsible for sharing identical blocks. With this approach, the I/O overhead in write requests is reduced, but more disk space is needed because duplicated data is not eliminated before being written to the storage. There is always a tradeoff between space consumed and I/O write operations overhead. In our scenario, we see the I/O overhead as a more critical issue because this is not an archival system and VMs need to have an acceptable throughput when writing to their virtual disks. In addition to these two operations, a copy-on-write mechanism is used. This mechanism is necessary to ensure that write operations on a physical block, that is being shared, do not corrupt other virtual addresses that are also pointing to that block. To achieve this goal, the Translation tables' physical addresses that are being shared by more than one virtual address are marked as copy-on-write. When a copy-on-write block's content needs to be updated, it is not modified directly that block. Instead, a new copy where the modified data will be written is created. When a write request for a physical address marked as copy-on-write is intercepted, an unused block (line 21) to write the new data is asked. This way, the virtual address will point to this new block and the old physical

block's content will remain unmodified. An additional mechanism is necessary to check if the old physical address is still being pointed by any virtual address or can be freed. This work is done by another module, that will collect all the physical addresses to be processed via the insert_address_free_cow_queue() method.

For each VM there is an independent module intercepting I/O requests. Data structures described above are not shared between VMs. This means that each VM has its own Translation Table and Dirty Addresses Table.

4.3 Share Module

Data that has been written can potentially be shared and must be examined. Virtual addresses that have content susceptible to be shared are kept in the Dirty Addresses table and processed later. In this section, it is described our module that processes this table and shares identical data. Listing 4.2 shows the pseudo-code for this module.

```
1  process_dirty_addresses_table(){
2
3     /*move operations copy elements from source to destination and remove
4    them from the source*/
5
6    lock_mutex_dirty_addresses_table()
7    move_dirty_addresses_table_addresses_to(hot_set)
8    unlock_mutex_dirty_addresses_table()
9
10   identical_addresses = get_identical_addresses_in(hot_set,cold_set)
11   remove_addresses_from_cold_set(identical_addresses)
12
13   addresses_to_be_processed = cold_set
14   remove_all_addresses(cold_set)
15   move_hot_set_addresses_to(cold_set)
16
17   return addresses_to_be_processed
18 }
19
20
21 share_data(){
```

```
22
23    addresses_to_be_processed = process_dirty_addresses_table()
24    for_each_virtual_address_in(addresses_to_be_processed){
25
26      lock_mutex_page_table()
27      physical_address = check_for_physical_address_in_page_table(virtual_address)
28      mark_as_cow(physical_address)
29      unlock_mutex_page_table()
30
31      content = read_from_storage(physical_address)
32      signature = calculate_signature(content)
33
34      lock_mutex_hash-to-address()
35      if(hash_is_in_hash-to-address_table(signature)){
36
37        shared_physical_address =
38                get_physical_address_value_from_hash-to-address_table(signature)
39
40        lock_mutex_page_table()
41
42        if(physical_address ==
43                              check_for_physical_address_in_page_table(virtual_address)){
44
45
46          update_page_table_virtual_address_mapping(virtual_address,
47                                                              shared_physical_address)
48          new_free_block(physical_address)
49          increment_entry_references_field_value_from_hash-to_address_table(signature)
50        }
51        unlock_mutex_page_table()
52      }
53      else{
54
55       references_field_value = 1
56       shared_physical_address = physical_address
57       insert_into_hash-to-address(signature,shared_physical_address,shares_field_value)
58      }
59      unlock_mutex_hash-to-address()
60    }
61  }
```

Listing 4.2: Share module pseudo-code.

Dirty Addresses table contains virtual addresses that were modified and consequently marked for the sharing process. This table is scanned periodically by the Share module. The addresses at this table are not shared immediately. A mechanism that finds what ad-

dresses are less susceptible to be written in a near future is used. This is done because some blocks are constantly modified and there is not any advantage in sharing them. If blocks that are constantly being modified are shared, the use of COW mechanism increases and thus the write requests' overhead also increases.

This approach to find what blocks are appropriated to be shared requires the use of two additional sets of virtual addresses. When the Share module scans the Dirty Addresses table, it copies its addresses to a first set of virtual addresses, which we call the Hot set (line 7). After the copy operation is completed, all the addresses that were marked to be processed are removed from the Dirty Addresses table. All addresses at the Hot set are compared with the addresses found at a second set, which we call the Cold set (line 10). Addresses that are common to the Hot set and to the Cold set are removed from the Cold set (line 11). When this comparison is finished, all the elements that remain at the Cold set are ready to be shared (line 13). After sharing all the addresses in Cold set and removing them from it, the Hot set's content is copied to the Cold set (line 14 and 15). In each scan, this process is repeated and, by doing this, there is the warranty that will only be shared addresses that were not marked as dirty in two consecutive scans.

After choosing what virtual addresses are ready to be shared, each of them is processed independently (line 24). First, the VM's Translation table's entry, where that virtual address belongs, is used to mark the physical address, pointed by it as copy-on-write (line 28). As a consequence of reading the block's content (line 31) with a certain delay from the time that the write requests were intercepted, there is the possibility of reading a physical block that was modified prior to the decision of sharing it. This detail does not bring any issues for our algorithm because all the modifications to that physical block were done via the same virtual address. The mechanism that only processes blocks that were not written in two consecutive scans

allow to reduce the number of times this situation occurs. Block's content is used to calculate its signature (line 32) that is used to search for identical blocks in the Hash-to-Address table (line 35), mentioned earlier in Section 3.2. In this table are kept mappings between blocks' content's signatures and their physical address at the storage. If the signature is found at the table, there is a physical block with the same content at the storage. To share the blocks, the virtual address, contained inside the VM's Translation table, needs to be updated to point to the new physical address found at the Hash-to-Address table's entry (line 46). After doing this, the two blocks are shared and the physical block that was pointed by the virtual address can be freed (line 48). Finally, the value that describes the number of virtual addresses sharing that physical block, which we call the number of references field (line 48), must be incremented. Each Hash-to-Address table's entry stores its own value for this parameter.

If the signature is not found at the Hash-to-Address table, then it is added as a new entry (line 54-57). This entry is composed by the signature, the physical address where the block is stored and the number of references field, which is initialized with the value one.

The Share module is the same for all VMs, but processes concurrently the algorithm described above for each VM. This way, a concurrency control mechanism is necessary for the Hash-to-Address table that will be accessed concurrently to eliminate redundancy between all the VMs' virtual disks. A similar mechanism is also necessary for the VM's Translation table and Dirty Addresses table because they are accessed concurrently by the Share module and by the module that intercepts I/O requests.

The new_free_block() function is used to report a free block to the Garbage Collector module (line 48); this function will be explained in Section 4.4. The interval of time that this module keeps the VM's Translation table locked must be as low as possible because this will

generate overhead for the I/O read and write requests. In function share_data() (line 21), the VM's Translation table is locked to read the physical address and mark it as copy on write but, this lock is released while the addresses' content signature is being calculated (line 26-29). When the signature is found at the Hash-to-Address table, the virtual address must be updated to point to another physical address (line 35). Before doing this procedure, the physical block's content must be checked to ensure that it has not changed while the VM's Translation Table's lock was not being held (line 42). If this content has changed this means that a copy-on-write operation was performed meanwhile and the physical address cannot be shared because is no longer in use by that VM.

4.4 Garbage Collector Module

The Garbage Collector module has a queue of free blocks (Free Blocks queue) and is responsible for distributing unused blocks across the I/O Interception modules when a copy-on-write occurs and is necessary a new block to write data modifications. GC module also is responsible for collecting unused blocks that are produced by sharing and copy-on-write operations. Listing 4.3 shows the pseudo code for this module's behavior.

```
1
2  new_free_block(physical_address){
3
4    lock_mutex_free_blocks_queue()
5    insert_into_free_blocks_queue(physical_address)
6    unlock_mutex_free_blocks_queue()
7  }
8
9  ask_for_free_block(){
10
11   lock_mutex_free_blocks_queue()
12   free_block_address = pop_address_free_blocks_queue()
13   unlock_mutex_free_blocks_queue()
14
15   return free_block_address
16 }
```

```
17
18 process_free_cow_queue(){
19
20   lock_mutex_free_cow_queue()
21   physical_address = pop_address_free_cow_queue()
22   unlock_mutex_free_cow_queue()
23
24   /*
25   if physical address is empty then the free_cow_queue has no more addresses
26   */
27   while(physical_address_is_not_empty(physical_address)){
28
29     content = read_from_storage(physical_address)
30     signature = calculate_signature(content)
31
32     lock_mutex_hash-to-address()
33     decrement_entry_shares_field_value_from_hash-to-address_table(signature)
34     shares_value = get_shares_field_value_from_hash-to-address_table(signature)
35
36     if(shares_value == 0){
37
38         new_free_block(physical_address)
39         remove_entry_from_hash-to-address_table(signature)
40     }
41     unlock_mutex_hash-to-address()
42
43     lock_mutex_free_cow_queue()
44     physical_address = pop_address_free_cow_queue()
45     unlock_mutex_free_cow_queue()
46   }
47 }
```

Listing 4.3: Garbage Collector module pseudo-code.

When two physical blocks are shared by the Share module, the block that is no longer used is immediately freed and made available to be used as a free block (line 2). This unused block is collected by the GC module and inserted in its Free Blocks queue (line 5).

The Garbage Collector also collects unused blocks that are produced at the copy-on-write operation. When a copy-on-write operation is executed for a specific physical address, there is one virtual address that is no longer pointing to that physical address. All these unused physical addresses are kept in a queue (Free COW queue) that will be consumed by the GC module (line 18). This module is respon-

sible for checking if these physical addresses are no longer pointed by any other virtual address. To achieve this goal, the GC module must calculate the signature of each physical block and search for its entry at the Hash-to-Address Table (line 29-32). After finding the correct entry, the value of the number of references field that represents the number of virtual addresses sharing that physical block is decremented (line 33). If the field value is zero, then that physical block can be added to the Free Blocks queue because is no longer used by any virtual address (line 36-40). If the value is greater than zero, that physical block cannot be collected. This approach can be taken because a physical block, which was marked as COW, cannot be modified until the GC module frees it.

Another important policy decision is to choose when the Garbage Collector must process the Free COW queue's elements. These requests are not processed immediately when they are introduced at the queue because it would generate more overhead in the I/O write operation. Garbage Collector runs when the queue reaches a determined size that can be modified accordingly to the space we are willing to spare for the queue. This method alone is not satisfactory because a scenario where the queue takes too long or never reaches that size threshold is possible to happen. This way, a second mechanism was introduced to activate the Garbage Collector if a determined time has passed since the last run.

The Garbage Collector's Free Blocks queue must be loaded with some free blocks in it. This is important because the blocks freed by the sharing process may not be sufficient for attending the unused block requests to perform COW operations. Such scenario is only expected in early stages of the sharing algorithm or for very specific cases, where shared blocks are constantly modified and sharing opportunities are not being properly addressed. For this last case, one solution is to increase the timing between Share module's scans, which will allow the Hot and Cold set approach, described in the

prior section, to perform better.

At this module, the garbage collection algorithm is processed concurrently for each VM. This way, concurrency control mechanisms when inserting and retrieving elements from the Free Blocks queue, which is shared between all the VMs, are needed. The Garbage Collector algorithm also accesses the Hash-to-Address table in parallel with the Share module. Each VM has its own Free COW queue, which is accessed concurrently by the Garbage Collector and the I/O interception module. The function new_free_block() is used by the GC and Share modules. Function ask_for_free_block() is used by the I/O Interception module when is necessary to ask for a free block for a COW operation.

Chapter 5

Prototype

This section describes the implementation details of our architecture. A mechanism that is used to load VMs images into the shared storage and to share data between VMs images is also presented. Finally, two optimizations are described. Our prototype works with Xen [2] version 3.3 as hypersivor and all the code is written in C.

5.1 Intercepting I/O Requests

There are several options to intercept I/O requests from a VM to its virtual disk. Dm-userspace [32] provides a way to control the device-mapper with a user space application. This solution allows intercepting VM's disk requests and redirecting them to a different location by using a user space process. The requests intercepted have information about the block's address that is being accessed and the type of operation (read/write). Requests are intercepted at block level and Dm-userspace is available to use with Xen or a Linux distribution. This solution's main problem is that it is no longer maintained. Since 2006 there are no new updates for Dm-userspace.

Blktap [29] presents virtual disks to virtual machines. This solution intercepts block I/O requests from VMs and reports them to a user space process. New user space drivers that present custom virtual

disks to VMs can be implemented easily. Read and write requests are reported separately and are intercepted at the block level. For each VM's disk, there is a different user space process intercepting the I/O requests. Blktap is only supported by Xen. With either Blktap or Dm-userspace, solutions that need copy-on-write operations can be implemented easily because read and write requests are reported independently, which allows knowing exactly the blocks that have been modified.

Fuse [22] provides a simple API in order to implement a file system in a user space program. It works at file level and it is supported for Windows, Linux and Mac OS X.

Another option is to intercept I/O requests at the virtual machine's operating system. This option is more complex than the others described above and may require having a different version for some operating systems. Since our approach was conceived to work at the block level instead of file level, the options we have to intercept blocks requests are blktap, dm-userspace and modifying directly guest operating systems. Dm-userspace is no longer maintained and modifying guest's OS directly is a complex solution. This way, our solution uses blktap to intercept block I/O requests.

In our study described at Chapter 3, the results for 4KB blocks are very similar to 8KB blocks in terms of space saved, having in account metadata's size and the redundant data found. In our implementation 4 KB blocks are used because our storage file system is ext3 [43], which allows to define block's sizes of 1KB, 2KB and 4KB. By choosing the default size of 4KB, the complexity of our implementation is reduced and the performance is maximized. Nevertheless, our code can be easily changed to work with 8KB blocks if necessary.

Blktap allows I/O requests to be intercepted via a user space process. We have modified the Tap Aio driver to implement our Blktap driver. Tap Aio implements a simplistic user space driver that intercepts I/O

requests and redirects them to the correct address at the VM image. I/O requests are processed asynchronously in batch.

Regarding our prototype, for each VM, a different process is created to intercept I/O requests. All these processes need to communicate with the Share module process. This communication is necessary to share identical data between VMs' virtual disks. The Share module must have access to each VM's Translation table and Dirty Addresses table. This is accomplished by sharing memory between the Share module process and the processes intercepting I/O requests. The `mmap` [44] function is used to share memory between two processes. An independent memory region is created for each process that is intercepting I/O requests. This region is shared with the daemon process. Each of these memory regions contains the VM's Translation table and the Dirty Addresses table.

The VM's Translation table is implemented as an array, where the indices act as implicit virtual addresses and the value pointed by these indices represent the physical address. The Dirty Addresses table is implemented as an array of bits. Each bit index represents an implicit virtual address. The value pointed by the index, zero or one, represents if that address needs to be processed or not. This data structure introduces more overhead to the metadata's size; however, this overhead is not significant. If we have into account the data set described in Chapter 3, where there are 29,530,586 4KB blocks, this table's size is: $29,530,586 \times 1bit = 3.5MB$.

To mark a physical address as copy-on-write, the high order bit of that physical address is used. If that bit is one, then the address is marked as copy-on-write.

5.2 Share Module

The Share module runs in an independent process. This process creates an independent thread to share blocks for each VM. This

thread has access to the shared memory where the VM's Translation table and the Dirty Addresses table are kept. This allows the Share module to scan the Dirty Addresses table for blocks to be shared, to update virtual addresses mappings at the VM's Translation table and to mark physical addresses as copy-on-write. In this shared memory, there are also mutexes that are used to control the concurrent accesses to both tables. To implement the Hot blocks' set and Cold blocks' set, an array of bits is used for each. This array is identical to the one used for Dirty Addresses table, described in the previous section.

Glib's Hashtable [37] is used to implement the Hash-to-Address table and SHA1 implementation of library openssl [49] is used to calculate the signatures of the physical blocks' content. As stated in Venti [18], the probability of a collision for a storage system with one exabyte of data is less than 10^{-20}, so we think that the use of SHA-1 hashes as unique identifiers for the blocks is acceptable.

5.3 Garbage Collector Module

The Garbage Collector module runs in the same process as the Share module. Each VM has two independent threads. One of them manages unused blocks and the other is responsible for providing unused blocks to the VM when a copy-on-write operation occurs. Both these threads access concurrently the Free Blocks queue, which is implemented as a FIFO queue.

To support the communication between the Garbage Collector's threads and the processes intercepting I/O requests, an additional data structure was introduced at the shared memory region. This data structure is a buffer of free blocks that is used by the I/O Interception process when it needs free blocks to complete a copy-on-write operation. The Garbage Collector's thread is responsible for filling this buffer when needed. This buffer was chosen to minimize the

waiting time to obtain an unused block. The size of the buffer must be adjusted to a value that minimizes the chance of this buffer being empty. Mutexes and condition variables are used in order to serialize the accesses to the buffer and to know when the buffer is empty or full. For each VM, there is one independent memory region and, consequently, one buffer of free blocks.

The other data structure that needs to be shared between the GC's thread and the VM's I/O Interception process is the Free Cow queue. This queue's size is variable, so the queue is not kept in the shared memory regions. Instead, the `mkfifo` [42] function is used to create a FIFO queue between the two processes, where physical addresses are inserted and retrieved. The size of the elements that are inserted in this queue is 64 bits, which assures that write and read operations from this queue are atomic. Consequently, there is no need to worry about concurrency control. Nevertheless, this FIFO queue's size is restricted and, because of that, the queue is only used in the communication process. The addresses are kept in another queue at the GC process. This queue is implemented with our C library and it is processed by the GC's thread that frees unused blocks that resulted from COW operations. In this process, the Hash-to-Address table is also used to check if physical addresses can be freed or are still in use by other virtual addresses.

5.4 Loading Virtual Machines Images

VM images must be loaded into the physical storage (block device) in a specific way required by our prototype. An application was written by us to provide this functionality. This application requires the VM's id to be specified as parameter. This id is needed to copy blocks from different VM's without overlapping them at the physical storage. This application is also responsible for generating the shared memory files that will be used between the I/O interception processes

and the Share/Garbage Collector process. Additionally, all the data structures that are at the shared memory are initialized with this application. VM's id is also important to have an independent share memory region for each VM. The blocks that will be initially loaded into the Free Blocks queue, which is managed by the GC module, are also chosen with this mechanism.

An additional feature that allows sharing the content of virtual machines images was also implemented. The algorithm described in Chapter 4 only explores redundancy when the physical blocks are modified and, in some cases, there are some virtual machines blocks that may never be written. This new feature marks all the VM's virtual addresses to be shared at the Dirty Addresses Table, which is loaded by this application. This will force the Share module to exam, and when appropriate share, all these addresses. This will only occur when the VM is started.

5.5 Prototype Optimizations

To decrease the overhead introduced by our prototype in the VMs' I/O requests, two prototype optimizations were implemented.

Both I/O Interception module and Share module must access simultaneously the VM's Translation table. This requires a lock mechanism. To optimize this lock mechanism, an approach that does not requires locking the entire Translation table to modify only one entry was implemented. For our specific scenario, an approach that uses one mutex per entry is not possible because the number of entries in the Translation table is too big[1]. Our optimization uses a predefined number of mutexes that are divided by the Translation table's entries. This way, the probability of two concurrent accesses to different entries locking in the same mutex is reduced.

[1]For a VM image with 10 GB there are 2.621.440 entries in the VM's Translation table.

Our second optimization improves the process of refilling the buffer of free blocks with unused blocks. This refilling operation is performed by the Garbage Collector. Condition variables are used to control when the buffer's size drops and a new block is needed. In the default approach the buffer is filled when its size drops by one element. To improve this mechanism, the granularity of the refill operations was changed. Instead of filling the buffer immediately when an address is consumed, the buffer is filled when a specific number of addresses are consumed. This way, the number of times that the GC module needs to fill the buffer is reduced and, consequently, the number of times the buffer needs to be locked because the producer and consumer are accessing it concurrently is also reduced.

Chapter 6

Experimental Evaluation

This chapter discusses the performance of our prototype. Three different benchmarks are presented as well as the results of running them in our prototype and in Tap Aio.

6.1 Tests Description

The main purpose of these benchmarks is to test the viability of our prototype and architecture to eliminate duplicated data in a virtualized scenario. This is achieved by measuring the VM's I/O requests throughput and latency, the CPU and RAM usage at the Dom 0 and the data being shared.

The three benchmarks used in our evaluation are:

Bonnie++ Benchmark

Bonnie++ is a stress benchmark that tests I/O requests throughput by using different sets of tests.

In our case, Bonnie++ runs in each VM and executes the following tests:

Sequential Write – Three different tests write data sequentially.

One uses `putc` stdio macro and another uses the `write` function to write blocks. The third test rewrites the data by reading it with the `read` function, modifying it and writing it again with the `write` function.

Sequential Read – Two different tests read data sequentially. The first uses the `getc` stdio macro and the other uses the `read` function to read blocks.

Random Seeks – In this test several processes run in parallel and use the `lseek` function to go to different locations in the file. These locations are given by a uniform distribution. When the location is chosen the file's block is read with the `read` function. In 10% of the cases the block is modified and written back with the `write` function.

Bonnie++ benchmark also executes file creation tests that we do not use for our evaluation. Bonnie++ is not intended to test algorithms like ours. This benchmark writes several times the same content. This means that our algorithm will achieve share ratings that are not realistic and our share module will be overloaded. In addition, bonnie does not have any test that writes data blocks in a random way and follows a distribution that is not uniform. This means that all the blocks are written with the same probability. To test the efficiency of the Hot and Cold sets of blocks, described in Section 4.3, some blocks must be written more frequently than others.

Write Benchmark

A write benchmark was implemented with two objectives. First, a realistic amount of duplicated data to be written is needed to be simulated. This is important to test if the prototype is sharing an expected amount of data and the share module is running with a realistic charge. Second, a scenario where some blocks are written more frequently than others is also necessary to be simulated. By

having a non uniform distribution is possible to test our algorithm's mechanism that shares blocks that are not expected to be rewritten several times.

The write benchmark is written in C and is aimed at running in each VM. In each VM, several processes that write data into a file, which is unique for each process, can be spawned. The number of processes and the file's size are configurable. All the write requests are done with `pwrite` [46] function and use a block's size of 4KB. The location for the write operation is given by NURand function from TPC-C benchmark [51]. This function generates a distribution where there are few blocks that are written several times and most blocks are written few times. In Figure 6.1 this distribution is exemplified by showing the number of blocks for each write frequency. These values were extracted from a run of our benchmark and belong to one of the processes that wrote 269,882 blocks into a 2 GB file.

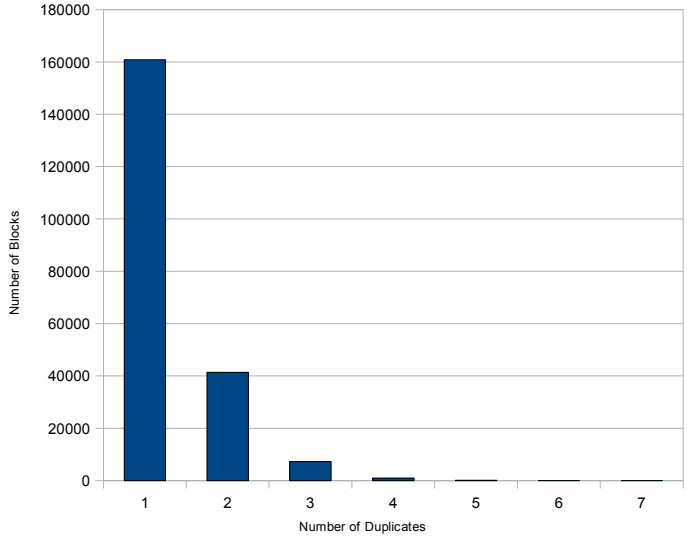

Figure 6.1: NURand distribution.

For generating the content of the 4KB blocks written by our benchmark, a distribution identical to the one found in the GSD's data, described in Chapter 3, is used. The results obtained from the GSD's study for the fixed size block approach with a block's size of 4Kb

are used to simulate this identical distribution. This allows having a more realistic scenario for the blocks' content that should be written. Consequently, a better approximation to the expected values of redundancy found by our prototype can be achieved. Of course this is only an approximation because we do not control the other blocks that are being written by the VMs while our algorithm is running. Additionally, some blocks are rewritten several times, which also difficult the prediction of the expected sharing rate because shared blocks can be modified and the share must be broken.

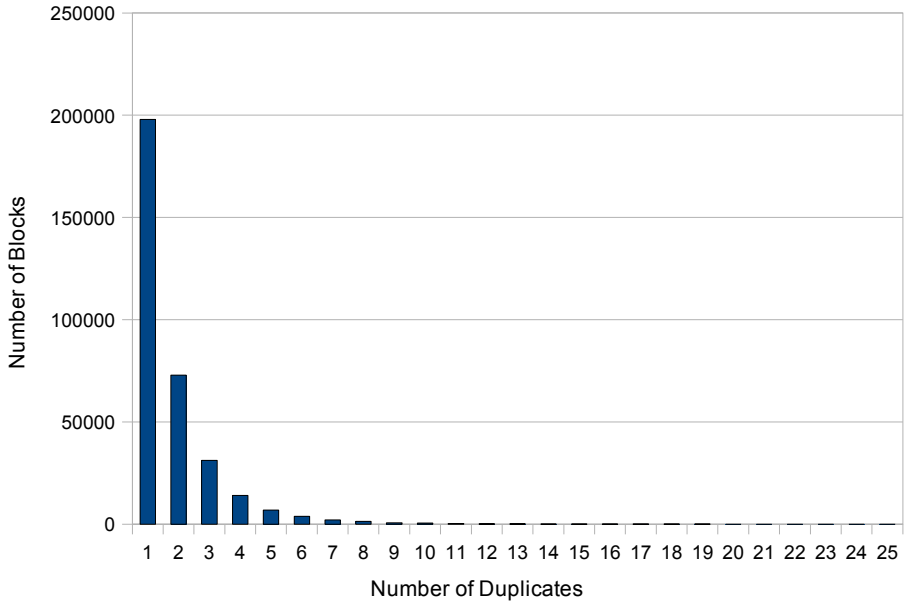

Figure 6.2: Redundancy distribution: Blocks with less than twenty five duplicates.

Figures 6.3 and 6.2 show the number of blocks, with a certain number of duplicates, that can be found in a typical run of our benchmark. In this run three different VMs that executed our benchmark application were used. In each VM, 4 processes that wrote blocks in files with a size of 2 GB were spawned. Each application ran for 30 minutes and the amount of data written in the three VMs was approximately 24 GB. In this figure, the number of blocks without duplicates, which is 5,222,288, is not described. If we compare these

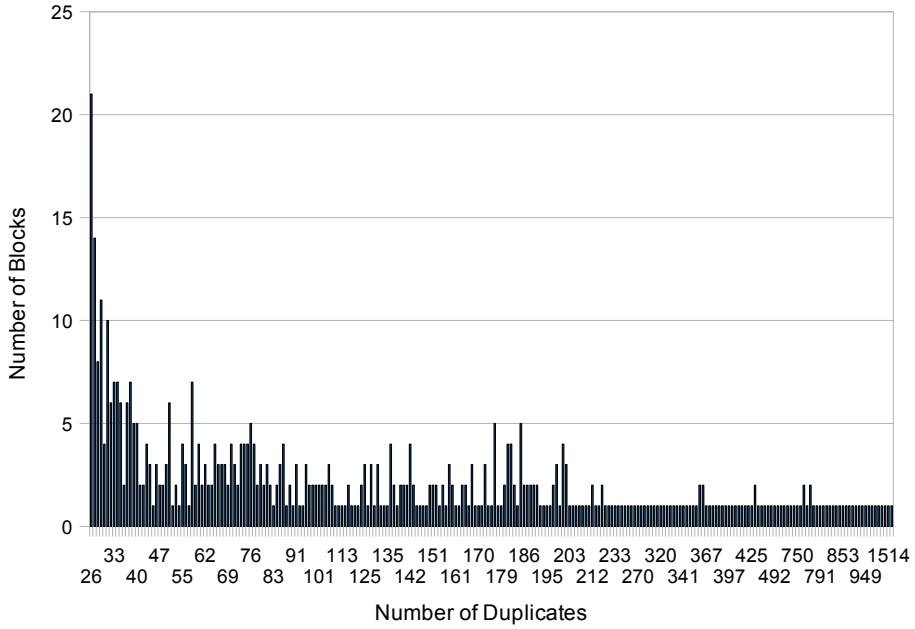

Figure 6.3: Redundancy distribution: Blocks with more than twenty five duplicates.

values with the ones described in Section 3.1, we conclude that this distribution is identical to the one we got from GSD's data set. Table 6.1 compares the values obtained in this distribution with the ones found in GSD's data set for the total number of blocks, the number of blocks that do not have any duplicate, the number of unique blocks, the number of identical copies found, the average number of duplicates per block and the average number of copies that a duplicated block has. By analyzing these values, we can conclude that they are proportional to the ones extracted from the GSD's study.

Finally, the amount of time that the benchmark will run can be defined as parameter.

Read Benchmark

A read benchmark was implemented to evaluate the overhead introduced in VMs' I/O read requests. More specifically, this bench-

Table 6.1: Benchmark redundancy results: Comparison with GSD's results.

	GSD's data set	Write Benchmark	Factor
Total Blocks	29,530,586	6,299,922	4.68
Blocks Without Duplicates	22,610,369	5,222,288	4.32
Unique Files/Blocks	24,449,410	5,555,557	4.40
Blocks to Eliminate	5,081,176	744,365	6.82
Duplicates per Regular Block	0.23	0.17	1.36
Duplicates per Duplicated Block	2.76	2.23	1.23

mark's purpose is to measure read requests throughput and latency for a scenario where the VMs' Translation tables are initialized in a realistic way and the Share module is not actively sharing addresses. RAM and CPU usage are also measured.

This benchmark resembles the write benchmark in some functionalities and it is also implemented in C. One instance of the benchmark runs in each VM and several processes that will perform I/O requests in an independent file can be spawned for each benchmark instance. The number of processes and file's size is defined as parameter. First, these processes write data to the file with the `pwrite` function. The position and content, of these requests, are calculated by using the same distributions used in the write benchmark. These write operations are important to achieve a realistic initialization of the VMs' Translation tables. The amount of time that the processes write data is defined as parameter.

After the write operations are completed, the application waits a certain amount of time, also defined as parameter, to start the read operations. This is necessary to ensure that when read requests start, the Share module is not sharing addresses from the write requests done before. However, when read requests are being performed, the share module will share some addresses that were not written by the benchmark application. Finally, after the waiting time has elapsed,

the read operations are performed with the `pread` [45] function. The location for the read operations is given by the NURand distribution and the size of the blocks to read is 4KB. The time interval for reading data is also defined as a parameter.

Test Environment

A server equipped with a AMD Opteron (tm) Processor 242, 4 GB of RAM and a 500Gb partition provided by a HP StorageWorks 4400 Enterprise Virtual Array (EVA4400) with RAID 0 was used for all the tests. In this server, we ran three VMs with Ubuntu 8.04 2.6.24-24-xen Kernel, 256 MB of RAM and 10 GB for their virtual disks. The operating system used for Dom0 OS was Ubuntu 8.04 with 2.6.24-24-xen Kernel's version. In our server it was installed Xen's version 3.3.

Each of the benchmarks ran in two distinct scenarios:

Tap Aio – In this scenario all the VMs use the Tap Aio driver that comes by default with Xen 3.3. Tap Aio driver was chosen as baseline because our Interception I/O module is a modification of this driver. The VM's images are copied into our global storage but are independent from each other, meaning that their data is not being shared.

Our Prototype – In this scenario we run our prototype at the Dom0. We use our mechanism, described in Section 5.4, for loading the VMs images into the storage.

Both scenarios were contemplated because we wanted to compare the results extracted from our prototype with the ones extracted from a baseline approach. This comparison allowed us to measure our prototype's overhead.

The mechanism presented at Section 5.4, which is intended to share VMs images while they are being loaded into the storage was not

used because we wanted to evaluate the performance of our algorithm for detecting duplicated data when VMs were running. This mechanism makes it harder to understand what data is being shared in runtime. However, this mechanism to share VMs images is only expected to introduce some overhead while VMs are being started and in the first scans of our Share and GC modules. After these first scans the system behavior should be identical to the one we evaluated.

The following values were considered for our prototype's static parameters:

- The interval between two consecutive runs of the share module's algorithm is 3 minutes. With this values the share algorithm runs approximately 6 times for each VM[1], performing a total of 18 times.

- The size of the Free COW queue that triggers the GC to process is 20,000. For the cases where this value is not reached in 5 minutes the GC processes the queue automatically. With these values the GC runs approximately 5 times for each VM. With this configuration, the maximum size that this queue will have in RAM is 156 KB. All the blocks stored at this queue are potential blocks to be freed by the garbage collector. This way, blocks that are hold in this queue represent potential wasted space. With the value 20,000 for the queue's size, the maximum space wasted, in the worst case that occurs when the queue is full, is 78.13 MB.

- The buffer of free blocks has space for 1200 addresses. Regarding RAM space, this choice does not occupy more that 10 KB per buffer. This buffer needs to be filled with addresses that point to storage's unused blocks. This means that for each

[1]The value for the number of times each component runs in our tests was calculated by running the benchmarks a few times with the appropriate settings.

buffer 4.68MB of the storage's free blocks are needed. This value for the buffer's size was chosen with the intent of never having the buffer empty. If the buffer is empty and a free block is needed there will be a significant delay in the VM's I/O write request.

- The number of mutexes per Page Table used is 100 and the fill size for the buffer of free blocks is 400 blocks.

These values were chosen with the intent of having a minimum impact on VMs' I/O requests throughput and latency. However, we tried to keep these values acceptable in terms of the RAM and Storage's size needed.

6.2 Bonnie++ Results

We ran one instance of Bonnie benchmark in each VM. Each of these instances wrote/read 12 GB.

Bonnie++ ran for the Tap Aio scenario and for our fully optimized prototype scenario (Mutex & Buffer).

Table 6.2 shows the throughput results for both scenarios and the percentage of overhead generated by our prototype. The throughput values at this table represent the sum of the results obtained for the three VMs. By looking at these results we can see that the four tests that were executed in the middle generate more overhead. The most important factor that explains this decrease in performance is the amount of load that this benchmark introduces in our share algorithm.

Table 6.3 shows the amount of data that was shared and relates it with the amount of data written. With Bonnie benchmark our algorithm shares 36% of the data. To explain this result is important to understand that each sequential write test writes 4 GB of data into files with 1 GB. All these tests, including the random test, write

Table 6.2: Bonnie++ benchmark I/O throughput results.

	Tap Aio	Mutex & Buffer	Overhead
Putc Test	73,285 KB/sec	67,517 KB/sec	8%
Write Block Test	162,483 KB/sec	124,520 KB/sec	24%
Rewrite Test	39,064 KB/sec	33,014 KB/sec	16%
Getc Test	35,549 KB/sec	27,388 KB/sec	23%
Read Block Test	106,119 KB/sec	84,400 KB/sec	21%
Random Seeks Test	304 KB/sec	290 KB/sec	5%

data into the same files. In addition, the content written to the files is the same in each I/O write request. More concretely, when the putc test finishes there are 4GB of identical data that can be shared. Then, all the other write tests will rewrite the same files with the same content, which means that sharing will be broken and redone for all the files' data, in each test. When the benchmark is over, 4 GB of identical data are expected to be found. In our case, we have three VMs, which means that is expected to be found 12 GB of redundant data at the storage. Looking at the table's results, we see that were found 12.9 GB. This value is slightly higher than the expected one, because it is necessary to contemplate the data that was shared between VMs and was not related with the benchmark. To conclude, this value allows us to verify that our prototype sharing algorithm is correct and to explain why this benchmark is not suited for our prototype. By writing the same content and rewriting it in each test, our Share module is overloaded, which decreases significantly the performance.

Table 6.3: Bonnie++ benchmark redundancy results.

	Mutex & Buffer
Space Saved	12.99 GB
Space Written	36 GB
Percent of Space Saved	36%

Looking at the results in Table 6.2, the higher values for the overhead are in the write block, rewrite, read with putc and read block tests. This happens because these tests are running while our prototype is actively sharing data. For the putc test where our algorithm has not started yet and for the random test where our algorithm has already shared all the data from the write tests the overhead values decrease. Despite these results, in average, our prototype introduces 15% of overhead in write and read requests for the Bonnie++ test, which is not a negative result. Table 6.4 shows the average values for CPU and RAM usage. The difference between RAM values can be justified with the data structures needed for the sharing process. Regarding CPU usage, the difference between the two scenarios can also be attributed to the load that is introduced to our share algorithm by Bonnie++.

Table 6.4: Bonnie++ benchmark CPU and RAM results.

	Tap Aio	Mutex & Buffer
Average CPU Usage	18.35%	41.30%
Average RAM Usage	26.91 MB	287.3 MB

6.3 Write Benchmark Results

Our write benchmark application ran in three VMs. In each VM, 4 processes were spawned and each wrote data into a file with 2 GB. The benchmark ran for 30 minutes.

For this benchmark only, we tested three versions of our prototype. One of them did not use any optimization (Base), another one used only the mutex optimization (Mutex) and the last one used the buffer and mutex optimizations (Mutex & Buffer).

Table 6.5 shows the values for throughput, write requests latency, RAM usage and CPU usage obtained for the four scenarios that were tested. The throughput and latency values at this table represent the

sum of the results obtained for the three VMs. The CPU and RAM average values are extracted from the Dom0. Each of the optimizations improves the throughput value without increasing significantly the RAM and CPU usage. In fact, CPU average value when using both optimizations is smaller. This may be happening because our prototype is batching requests for the buffer of free blocks. By taking this approach, the number of operations performed decreases and consequently the computational power required is also reduced. Our fully optimized approach also has an average CPU usage that is acceptable when compared to the Tap Aio results. The increase in RAM's usage is a consequence of having blocks being shared. In fact, the differences between the values for each prototype's version are justified by larger throughput values that also increase the number of blocks that will be shared. In this case our fully optimized approach uses significantly more RAM than Tap Aio. This happens because the data structures that will be used by our algorithm are kept in memory. This value is acceptable for these tests and can be decreased significantly if some data structures like the Hash-to-Address table and Free Blocks queue are stored partially on disk. Finally, the throughput and write latency of our fully optimized approach are similar to the ones extracted from Tap Aio. The overhead introduced by our prototype is less than 4% for this last comparison.

Table 6.5: Write benchmark results.

	Tap Aio	Base	Mutex	Mutex & Buffer
Write Throughput (blocks/sec)	2952	2218	2598	2841
Write Latency	3.9 ms	5.1 ms	4.5 ms	4.1 ms
Average RAM usage	2.43 MB	220.3 MB	248.8 MB	252.45 MB
Average CPU Usage	18.9%	28.67%	31.30%	26.52%

Table 6.6 shows the amount of data that was shared by each version of our prototype. The values are identical for the three scenarios and the variations are justified by the throughput values that each version accomplished. Each of these version shared approximately

25% of the data that was written. This percentage is slightly higher than the one found in the GSD's data set that was around 16%. Once again, the value for the amount of data that should be shared is difficult to calculate precisely. Nevertheless, this value seems to be coherent with the expected one.

Table 6.6: Write benchmark redundancy results.

	Prototype	Mutex	Mutex & Buffer
Total Blocks Shared	1,128,817	1,229,958	1,327,355
Space Saved	4.31 GB	4.69 GB	5.06 GB
Space Written	15.23 GB	17.84 GB	19.51 GB
Percent of Space Saved	28%	26%	26%

6.4 Read Benchmark Results

Our read benchmark application ran in three VMs. In each VM, 4 processes were spawned and each wrote and read data from a 2 GB file. Write requests were performed for 10 minutes; then, there was an interval of 10 minutes before starting the read operations, which were performed for 20 minutes. The benchmark ran for the Tap Aio scenario and for our fully optimized scenario (Mutex & Buffer). In both scenarios, approximately 1.5 GB of data was written in each VM.

Table 6.7 shows the values for VM's read requests throughput and latency and for the CPU and RAM usage. RAM values refer to the entire run of the benchmark and CPU values refer only to the read operations. Our prototype does not introduce a significant amount of overhead, 1%, in VMs' I/O read requests, for the evaluated scenario. Regarding CPU usage, the average value increases 1%. The RAM usage is explained, once again, by the metadata that is used by our prototype to share identical data.

Table 6.8 shows the amount of data that was shared by our prototype

Table 6.7: Read benchmark results.

	Tap Aio	Mutex & Buffer
Read Throughput (blocks/sec)	882	875
Read Latency	13.5 ms	14 ms
Average RAM usage	2 MB	204 MB
Average CPU Usage	7%	8%

and the amount of data written for the three VMs. By Looking at these values, 55% of the data was shared. This value is higher than the values obtained in the other two benchmarks. This happens because this benchmark writes less content than the other two, which makes more visible the data shared by the VMs that was not written by the benchmark.

Table 6.8: Read benchmark redundancy results.

	Mutex & Buffer
Space Saved	2.5 GB
Space Written	4.5 GB
Percent of Space Saved	55%

6.5 Summary

The results described above allow us to extract several conclusions. Regarding Bonnie++ benchmark's results, we can conclude that this benchmark is not suited to test algorithms like ours. This happens because Bonnie++ writes exactly the same data for each write operation, which overloads the sharing algorithm. Besides that, the location where data is written is chosen with a uniform distribution, which does not allows taking advantage of our mechanism to detect blocks that are modified frequently. Nevertheless, this benchmark allows us to see how our prototype behaves in an adverse scenario and to conclude that our prototype finds almost all duplicated data

and shares it.

Regarding read benchmark's results, we can conclude that our prototype does not introduce significant overhead in VMs' I/O read requests in a scenario where the VMs' Translation table is initialized in a realistic way and the Share module is not actively sharing data.

From the write benchmark's results, we can extract the most meaningful conclusions because it represents a realistic scenario where there is an intensive charge of write requests to the storage. These results allow us to conclude that with 7.62% of overhead in CPU usage and 4% of overhead in I/O write requests our prototype can save 26% of space in the storage.

Regarding the RAM usage, for all benchmarks, we think that is possible, as future work, to improve its usage by choosing a better approach to manage the data structures needed by our approach.

Chapter 7

Conclusion

This dissertation introduces a solution to find and eliminate duplicated data in a virtualized system. First, the effectiveness of two techniques to find duplicated data was evaluated with the GSD's data set, which contains personal files from several researchers. One of the techniques detects duplicated files and the other detects duplicated blocks with a fixed size. For our specific scenario, we concluded that the block approach is better. This observation is still true when the space overhead introduced by the metadata necessary to eliminate duplicated data is taken in account. Three different block's sizes were used and the results, in terms of space saved, were similar. This study was essential to define a realistic benchmark to test our prototype.

A solution to detect and eliminate redundancy between virtual disks of VMs, which are running in the same physical machine and sharing a common storage, was presented. Our solution does not use a typical scan approach to detect duplicated data at the storage. Instead it uses a dynamic approach that intercepts I/O write requests from the VMs to their virtual disks and uses this information to share identical data. With this approach, we reduce the computational power that would be necessary by using a scan method. We also minimize the overhead introduced into the I/O write requests, by

delaying the process of calculating the blocks signatures and sharing them. Our architecture is composed by three modules and each has a different purpose. The I/O interception module intercepts VMs' I/O requests and redirects them to the correct physical address. This module keeps a list of all the blocks that were written, which will be consumed by our Share module. The Share module is responsible for processing each element of that list and sharing it. At this module, an additional mechanism was introduced to prevent the sharing of blocks that are modified frequently. Besides these modules, there is a Garbage Collector module responsible for distributing free blocks to the I/O Interception module and collecting unused blocks that result from the sharing process and from the copy-on-write operations. The copy-on-write operations are fundamental to prevent VMs from modifying blocks that are being shared.

We also presented a prototype that uses Xen and implements the architecture described above. This prototype contemplates a new mechanism that was not introduced in our architecture. This mechanism is used to load the VMs' images into the global storage and allows sharing data contained in VMs images that is never written. Two optimizations were introduced in our prototype with the intent of increasing the throughput of VMs' write and read requests. In fact, these optimizations were crucial to achieve I/O throughput rates that are identical to the ones obtained from a baseline approach were no deduplication is performed.

Finally, we benchmarked our prototype using a distribution to generate the content to be written that resembles the distribution of GSD's data set. The TPC-C NURand function was used to calculate the file's position where read and write operations are executed. From these benchmarks, we concluded that our prototype shares identical data between several VMs without introducing a significant amount of overhead in the CPU usage and in I/O requests throughput and latency. Regarding RAM usage, the value is accept-

able for our study purpose. In fact, RAM usage can be improved significantly by using an optimized approach to store the metadata used by our prototype.

To conclude, this document presents a solution to find and eliminate duplicated data in a virtualized scenario, which is not addressed, as far as we know, by any commercial or open-source product.

Chapter 8

Future Work

This work addresses the challenge of deduplication between VMs running in the same physical host. As future work, our algorithm can be adapted to a distributed environment where virtual disks from VMs, running in different physical machines, are shared. Our approach was designed to be easily adaptable to this new scenario. The two main remaining challenges are: the need of a distributed Hash-to-Address table, shared by all the physical machines; and a distributed system to manage the global storage's free blocks. This last module is responsible for collecting and retrieving storage's unused blocks from the GC module that is running in each physical node. This way, there is a distinct instance of the Share module and GC module running in each physical node, interacting with these two new mechanisms. Solutions for both these problems are already documented. A Distributed Hash Table [19, 21] may be used to implement the distributed Hash-to-Address table. An "extend" approach, identical to the one described in Section 2.4 from Parallax, can be used to manage the storage's unused blocks.

A dynamic scenario, where the VMs are created, started and stopped while our algorithm is running, can be also addressed as future work. In our prototype, VMs can be stopped and resumed, but the number of VMs is defined statically when the Share module starts. Regard-

ing the dynamic creation of new VMs, a mechanism that calculates if the storage has enough space for that VM's virtual disk and reports the new VM to the Share and Garbage Collector modules is necessary. One possibility is to use the garbage collector module to handle the virtual disk's space allocation. The process of stopping and resuming VMs can also be improved. In our work, we are not concerned with the loading or unloading of VM's data structures, present at the share and GC modules, when the VMs are resumed or stopped respectively. The resume and stop VM's operations must have in account the loading/unloading of data structures like VM's Translation table, Dirty Addresses table and Free COW queue, in order to optimize memory consumption. The occurrence of failures in our modules and in VMs is also important to consider in a future version.

Both Hash-to-Address table and Free Blocks queue are kept in memory in our implementation. However, they can grow significantly for large data sets. As future work, an approach where these data structures are kept on disk and a cache mechanism is used can be designed.

The solution proposed by VMware ESX Server [55] describes a hint mechanism that is used to reduce the number of addresses marked as copy-on-write. This mechanism allows marking only as copy-on-write the addresses that are really being shared. In our work, we also have physical addresses, which are not actually being shared, marked as copy-on-write. A similar mechanism can be created for our approach.

As discussed in chapter 4, our algorithm does not address the issue of maintaining some redundant data to provide a fault-tolerant storage service. In section 3.2, we introduced two ideas to achieve this goal but they are not currently being used.

Some of the process of detecting duplicates can be adapted and used to optimize data transmission. An approach that uses the Hash-to-

Address table to check what data is already at the storage and sees what parts of the request are not worth uploading can be implemented.

A study concerning the impact of the modification of the static parameters, used in our approach, is also important to be accomplished. All these variations will represent a trade off between the RAM needed, the Storage's space needed and the overhead introduced to the VM's I/O requests. This study is important to understand how the modification of one parameter interacts with the others, as well as to understand the best combinations for different workloads.

Finally, there is also the TRIM solid-state-drive command that allows the operating system to communicate with a solid-state-drive controller and pass information about what data blocks are no longer in use [52]. For now this solution is in its early stages but, in the future, it can be useful to simplify our Garbage Collector mechanism.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[2] 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 '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[3] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264. ACM, 2008.

[4] A. Z. Broder. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[5] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Comput-*

ing and Communications, pages 5–13. IEEE Computer Society, 2008.

[6] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Symposium on Operating Systems Design and Implementation*, pages 285–298. USENIX Association, 2002.

[7] B. Eckel. *Thinking in Java.* Prentice Hall PTR, 1998.

[8] T. E.Denehy and W. W. Hsu. Duplicate management for reference data. Technical report, IBM Research, 2003.

[9] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[11] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5. USENIX Association, 2004.

[12] U. Manber. Finding similar files in a large file system. In *Usenix Winter 1994 Technical Conference*, pages 1–10. USENIX Association, 1994.

[13] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 41–54. ACM, 2008.

[14] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *2009 USENIX Annual Technical Conference.* USENIX Association, 2009.

[15] A. Muthitacharoen, B. Chen, D. Mazieres, and D. M. Eres. A low-bandwidth network file system. In *In Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187. ACM, 2001.

[16] D. Nurmi, R. Wolsky, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A technical report on an elastic utility computing architecture linking your programs to useful systems. Technical report, University of California Computer Science Department, 2008.

[17] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6. USENIX Association, 2004.

[18] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101. USENIX Association, 2002.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350. ACM, Nov. 2001.

[20] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM, 2001.

[22] M. Szeredi. Filesystem in user space. http://fuse.sourceforge.net/. accessed 5th October, 2009.

[23] A. Tridgell and P. Mackerras. The rsync algorithm. Technical report, Australian National University, 1998.

[24] (Unattributed). Amazon ebs documentation. http://aws.amazon.com/ebs/. accessed 15th September, 2009.

[25] (Unattributed). Amazon ec2 documentation. http://aws.amazon.com/ec2/. accessed 15th September, 2009.

[26] (Unattributed). Amazon s3 documentation. http://aws.amazon.com/s3/. accessed 15th September, 2009.

[27] (Unattributed). Amazon simple db documentation. http://aws.amazon.com/simpledb. accessed 15th September, 2009.

[28] (Unattributed). Amazon web services used in dropbox. http://developer.amazonwebservices.com/connect/entry!default.jspa?categoryID=89&externalID=1955&fromSearchPage=true. accessed 10th September, 2009.

[29] (Unattributed). Blktap documentation. http://wiki.xensource.com/xenwiki/blktap. accessed 5th October, 2009.

[30] (Unattributed). Box.net documentation. http://www.box.net/info. accessed 10th September, 2009.

[31] (Unattributed). Data domain documentation. http://www.datadomain.com/products/. accessed 15th September, 2009.

[32] (Unattributed). Dm-userspace documentation. http://wiki.xensource.com/xenwiki/DmUserspace. accessed 5th October, 2009.

[33] (Unattributed). Dropbox documentation. http://www.getdropbox.com/. accessed 10th September, 2009.

[34] (Unattributed). Emc avamar documentation. `http://www.emc.com/products/detail/software/avamar.htm`. accessed 15th September, 2009.

[35] (Unattributed). Exagrid documentation. `http://www.exagrid.com/products/exagrid_product_line.asp`. accessed 15th September, 2009.

[36] (Unattributed). Freedup documentation. `http://www.freedup.org/`. accessed 5th October, 2009.

[37] (Unattributed). Glib hash tables module documentation. `http://library.gnome.org/devel/glib/unstable/glib-Hash-Tables.html`. accessed 5th October, 2009.

[38] (Unattributed). Google app engine datastore documentation. `http://code.google.com/appengine/docs/python/datastore/`. accessed 15th September, 2009.

[39] (Unattributed). Google app engine documentation. `http://code.google.com/appengine/docs/`. accessed 15th September, 2009.

[40] (Unattributed). Google docs documentation. `http://docs.google.com/support/bin/topic.py?hl=en&topic=15114`. accessed 10th September, 2009.

[41] (Unattributed). Ibm protect tier documentation. `http://www-07.ibm.com/vn/storageinnovations/index2.html#2`. accessed 15th September, 2009.

[42] (Unattributed). Mkfifo linux manual page. `http://linux.die.net/man/3/mkfifo`. accessed 5th October, 2009.

[43] (Unattributed). Mkfs.ext3 linux manual page. `http://linux.die.net/man/8/mkfs.ext3`. accessed 10th October, 2009.

[44] (Unattributed). Mmap linux manual page. http://linux.die.net/man/3/mmap. accessed 5th October, 2009.

[45] (Unattributed). Pread linux manual page. http://linux.die.net/man/2/pread. accessed 15th October, 2009.

[46] (Unattributed). Pwrite linux manual page. http://linux.die.net/man/2/pwrite. accessed 15th October, 2009.

[47] (Unattributed). Python 2.6 on-line documentation. http://docs.python.org/. accessed 5th October, 2009.

[48] (Unattributed). Rapidshare web page. http://www.rapidshare.com/. accessed 10th September, 2009.

[49] (Unattributed). Sha-1 linux manual page. http://linux.die.net/man/3/sha1. accessed 5th October, 2009.

[50] (Unattributed). Time machine web page. http://www.apple.com/macosx/what-is-macosx/time-machine.html. accessed 15th October, 2009.

[51] (Unattributed). Tpc-c standard specification, revision 5.5. http://www.tpc.org/tpcc/spec/tpcc_current.pdf. accessed 15th October, 2009.

[52] (Unattributed). Trim overview. http://www.anandtech.com/storage/showdoc.aspx?i=3531&p=10. accessed 10th October, 2009.

[53] (Unattributed). Fips 180-2, secure hash standard, federal information processing standard (fips), publication 180-2. Technical report, National institute of standards and technology, Department of Commerce, 2002.

[54] (Unattributed). Introduction to cloud computing architecture. White paper, Sun Microsystems, Inc, 2009.

[55] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceeedings of the 5th USENIX symposium on Operating Systems Design and Implementation*. USENIX Association, 2002.

[56] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: managing storage for a million machines. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 4–4. USENIX Association, 2005.

[57] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 804–8015. IEEE Computer Society, 2005.