IOSR Journal of Engineering (IOSRJEN)
e- ISSN: 2250-3021, **p**-ISSN: 2278-8719
Vol. 3, Issue 7(July. 2013), ||V5 || PP 01-10

Efficient Storage of Large Scale Data in Cloud Computing

B. Vamshi Krishna[1], D. Raman[2]
[1]M.Tech (CSE), Vardhaman College of Engineering, Hyderabad, India
[2]Associate professor in CSE Dept, Vardhaman College of Engineering, Hyderabad, India

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's storage, it is possible to improve the quality of service ordered, 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's storage can be used to simplify and enhance this management. This review study 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. Finally, a study that compares the efficiency of two different approaches used to eliminate redundant data in a personal data set is described.

Keyword: - *Cloud Computing; Data Redundancy; Virtual machines.*

I. 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. 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, Box.net, Rapid Share and Google Docs, that allow clients to have their data safe in the web. These services are more than just simple data archives. 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 and Google App Engine provide this type of service. Another kind of service has the goal of providing remote storage. There are also other cloud services with different functionalities, like Amazon SimpleDB 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 hours a day and seven days per week. Besides that, clients' data is stored redundantly in several data centers 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, 17] technology by cloud services. Virtual machines (VMs) allow these services to have increased flexibility in terms of deployment and re-deployment 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 al-lows 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 centers. 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.

Regarding cloud computing provider's storage, duplicated data is expected to be found between the several virtual machines images and between data stored remotely. 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.

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.

II. PROBLEM STATEMENT

Effective deduplication in a cloud computing scenario rises 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.

2.1 Objectives

The main goal of this dissertation is to show how deduplication can be achieved in a virtualized system, towards finding and eliminating redundant data in the context of cloud computing services. A second objective is to evaluate the impact of deduplication in personal data towards demonstrating the usefulness of the proposed solution.

2.2 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 Xian [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 les' data set. In this study, we compare two distinct methods: the whole le approach, which find les with the same content, and the fixed block size approach, which finds blocks, with a fixed size, that have the same content.

III. 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.

3.1 Finding and Eliminating Duplicated Data

There are several methods to find redundant data. The first of these methods is the whole file content hashing [15] 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 [6, 9, 15], 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, 6, 9, and 15]. 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. 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 modification were made. The fixed block size approach needs to re-calculate the signature for the blocks where modification 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 [9, 10] can be used to find similar data. A super-fingerprint is a group of fingerprints be-longing 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 [9] is used to reduce redundancy between similar files. This method generates a delta file containing differences between the les, which allow keeping only one le and the delta le needed to rebuild the other file. This method does not find similar files.

REBL [9] 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 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.

All the methods described above were tested in a scenario where the detection of duplicated data was accomplished with a scan approach. 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.

3.2 Remote Backup/Storage Services

LBFS [13] is a network le system designed to reduce bandwidth when transmitting les to the server. Files are divided into content defined chunks using Rabin fingerprints. For each chunk, a SHA-1 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 Vent [16] 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.

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.

3.3 Cloud Services

Cloud Computing storage services like Amazon S3, Amazon EBS, Amazon SimpleDB and Google App Engine Data store 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 Amazon SimpleDB and Google App Engine Data store allow clients to store data and to be able to perform queries on it.

Google App Engine and Amazon EC2 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 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 [14] 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 Xian 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. 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.

3.4 Virtualization Scenarios

Parallax [11] presents a solution to reduce redundancy among persistent snapshots 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.

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 [12] and VMware ESX Server present two approaches that 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. 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.

IV. 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.

4.1 Redundancy Detection

This section presents our study about duplicated data found in GSD's 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. 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. Freed up was used to detect duplicated files. Freed up 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 [8], 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.

Table 4.1: 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	1,676,047	29,530,586	15,464,284	10,794,932
Files/Blocks Without Duplicates	536,417	22,610,369	1,158,715	7912074
Unique Files/Blocks	765,594	24,449,140	12,584,338	8,632,784
Files/Blocks to Eliminate	910,452	5,081,876	2,87,949	2,164,844
Space Saved	13.37GB	16.75GB	16.35GB	16.01GB
Duplicates per Regular File/Block	.68	.23	.25	.27
Duplicates per Duplicated File/Block	0.39	2.76	2.87	3.00

Space saved improves by using the fixed size block method instead of the whole le. 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 le approach saves 12.4%. This represents the worst method. We can also witness that, in average, each file has less than one duplicate, but if the le is replicated, it possesses, in average, 4 identical copies. For blocks, the average results are inferior, but the relation is similar.

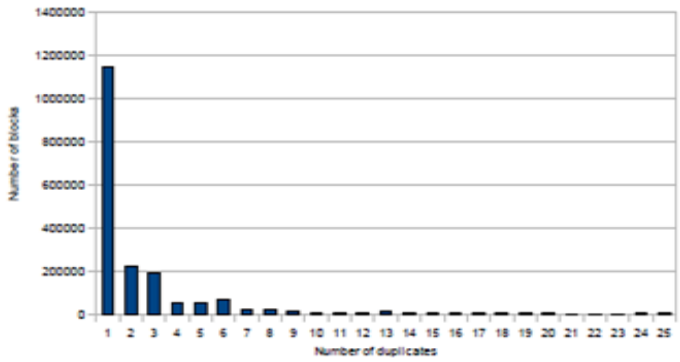

Figure 4.1 shows the number of les we can find in the GSD's data set with a certain number of duplicates.

This figure shows values up to twenty five duplicates of the same file.

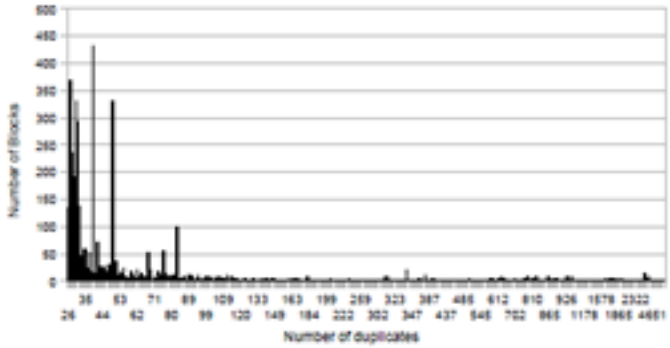

Figure 4.2 shows the same information for files with more than twenty five duplicates.

These figures are interesting to understand that the number of les with few duplicates is substantially higher and that this number drastically decreases when the number of duplicates per le grows. We also think that is interesting to notice that there are les with more than two thousand duplicates. We only present here charts for the le 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.

4.2 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 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 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 trans-parent. 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 4.1, which represents the total items found at the data set before eliminating duplicated data.

	Translation Table's Size
4 KB	225.3 MB
8 KB	117.9 MB
12 KB	82.3 MB
File	12.79 MB

Table 4.2 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. 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:

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

N_u is the number of different items at the physical storage, Hash's 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 4.3: 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

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. However, if we try to reduce Hash-to-Address table's size, there is always a trade of. 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 of between metadata's size and computational overhead generated.

As stated before, some redundancy is necessary at the storage. In this survey, 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 ad-dresses 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.

V. ARCHITECTURE OF DEDUPLICATION

This chapter introduces architecture to detect and share duplicated data from VMs running in the same physical host and sharing the same storage.

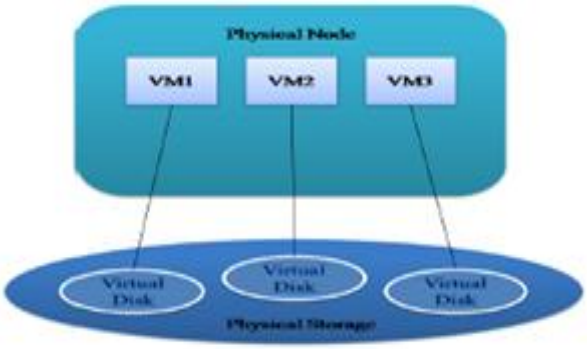

Figure 5.1 describes this scenario.

5.1 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.

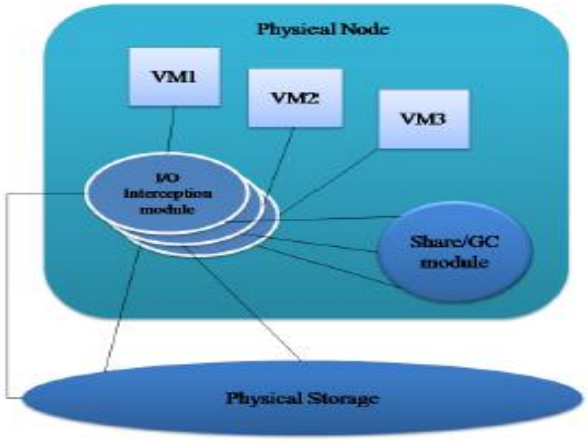

Figure 5.2 shows a description of the architecture

5.2 Intercepting

I/O Requests the I/O Interception module is responsible for intercepting I/O re-quests, 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. For each VM there is an independent module intercepting I/O re-quests. Data structures described above are not shared between VMs. This means that each VM has its own Translation Table and Dirty Addresses Table.

5.3 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.

The Share module is the same for all VMs, but processes concurrently the algorithm described above for each VM. This way, a con-currency 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.

5.4 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 modification. GC module also is responsible for collecting unused blocks that are produced by sharing and copy-on-write operations. 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.

VI. CONCLUSION

This review 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 les from several researchers. One of the techniques detects duplicated les 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.

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.

To conclude, this review 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.

VII. REFERENCES

[1]. M. Armbrust, A. Fox, R. Grifith, 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. Xian 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: Pro-ceedings 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, Secu- rity, 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 Computing and Communications, pages 5_13. IEEE Computer Society, 2008.
[6]. T. E.Denehy and W. W. Hsu. Duplicate management for reference data. Technical report, IBM Research, 2003.
[7]. R. P. Goldberg. Survey of virtual machine research. Computer, 7(6):34_45, 1974.
[8]. B. W. Kernighan and D. M. Ritchie. The C Programming Language. Prentice Hall, second edition, 1988.
[9]. 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.
[10]. U. Manber. Finding similar files in a large file system. In Usenix Winter 1994 Technical Conference, pages 1_10. USENIX Association, 1994.
[11]. 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.
[12]. G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In 2009 USENIX Annual Technical Conference. USENIX Association, 2009.
[13]. A. Muthitacharoen, B. Chen, D. Mazieres, and D. M. Eres. A low-bandwidth network files system. In in Proceedings of the Symposium on Operating Systems Principles (SOSP'01), pages 174_187. ACM, 2001.
[14]. D. Nurmi, R. Wolsky, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youse_, 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.
[15]. 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.
[16]. S. Quinlan and S. Dorward. Vent a new approach to archival storage. In FAST '02: Proceedings of the Conference on File and Storage Technologies, pages 89_101. USENIX Association, 2002.
[17]. J. E. Smith and R. Nair. The architecture of virtual machines. Computer, 38(5):32_38, 2005.
[18]. (Unattributed). Amazon ebs documentation. http://aws.amazon.com/ebs/. accessed 15th September, 2009.