Collocated Deduplication with Fault Isolation for Virtual Machine Snapshot Backup

Wei Zhang, Michael Agun, Tao Yang
Department of Computer Science
University of California, Santa Barbara, CA 93106

ABSTRACT

A cloud environment that hosts a large number of virtual machines has a high storage demand for frequent backup of system image snapshots. Deduplication of data blocks can lead a big reduction of redundant blocks when their signatures are identical. However it is expensive and less fault-resilient to perform a global comparison of all data block signatures and let a data block share by many virtual machines. This paper studies a cluster-based design which collocates a lightweight backup service with the cloud service and integrates multiple duplicate detection strategies that localize the deduplication as much as possible within each virtual machine. Our analysis shows that this virtual machine centric scheme uses a small amount of memory resource to conduct duplicate detection and can provide better fault tolerance through deduplication localization. This paper provides a comparative evaluation of this scheme in accomplishing a high deduplication efficiency while sustaining a good backup throughput.

1. INTRODUCTION

In a cluster-based cloud environment, each physical machine runs a number of virtual machines as instances of a guest operating system and their virtual hard disks are represented as virtual disk image files in the host operating system. Frequent snapshot backup of virtual disk images can increase the service reliability. For example, the Aliyun cloud, which is the largest cloud service provider by Alibaba in China, automatically conducts the backup of virtual disk images to all active users every day. The cost of supporting a large number of concurrent backup streams is high because of the huge storage demand. Using a separate backup service with full deduplication support [4, 7] (Figure 1) can effectively identify and remove content duplicates among snapshots, but the solution can be expensive. There is also a large amount of network traffic to transfer data from the host machines to the backup facility before duplicates are removed.

This paper seeks for a low-cost architecture option and takes advantage of the fact that the backup service collocates on the cloud cluster with a minimum resource usage (Figure 1) Cloud providers often wish that the backup service only consumes small or modest resources with a minimal impact to the existing cloud services. We study and evaluate the integration of several deduplication strategies to meet the low-cost collocation requirement. In parallel, we identify deduplication options that yield better fault isolation. We achieve improved fault isolation by accounting for the fact that after deduplication most data blocks are shared by several to many virtual machines. Failure of such popluar blocks would have a catastrophic effect and many snapshots of virtual machines would be affected. The previous work in deduplication focuses mainly on the efficiency and approximation of finger print comparison, and has not addressed fault tolerance along with a low-cost solution. Another issue considered is that deletion of old snapshots compete for computing resources as well so deletion needs to be considered in system design. The additional complexity for deletions come from the dependencies created by multiple links to the same problem, so some solution is needed to determine blocks with no dependent snapshots after deletion.

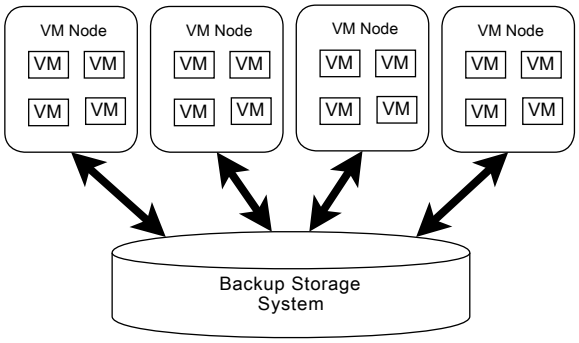


Figure 1: Traditional VM Backup System

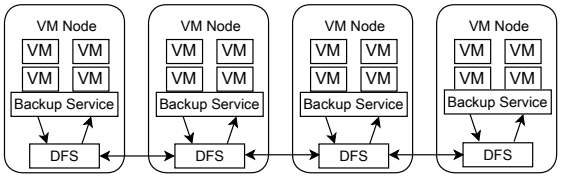


Figure 2: Colocated VM Backup System

The paper studies and evaluates an integrated approach which uses multiple duplicate detection strategies based on version detection, inner VM duplicate search, and controlled cross-VM comparison. This approach is VM centric by localizing duplicate detection within each VM as much as possible and minimize memory usage while delivering a decent deduplication efficiency. Our partially localized solution also brings the benefits of parallelism, utilization, and fault isolation. Our deletion strategy uses a double Bloom filter strategy for periodic mark-and-sweeping of expired data blocks. We have developed a prototype system that runs a cluster of Linux machines with Xen. The backup storage uses a standard distributed file system with data replication and block packaging, and we allocate more replicas for the commonly-shared data blocks among VM snapshots to enhance fault tolerance.

The rest of this paper is organized as follows. Section **??** reviews background and related work. Section **??** discusses the design framework and system architecture. Section **??** analyzes the benefit of our approach for fault isolation. Section **??** is our experimental evaluation that compare with the other approach. Section **??** concludes this paper.

2. BACKGROUND AND RELATED WORK

At a cloud cluster node, each instance of a guest operating system runs on a virtual machine, accessing virtual hard disks represented as virtual disk image files in the host operating system. For VM snapshot backup, file-level semantics are normally not provided. Snapshot operations take place at the virtual device driver level, which means no fine-grained file system metadata can be used to determine the changed data.

The previous work for storage backup has extensively studied data deduplication techniques can eliminate redundancy globally among different files from different users. Backup systems have been developed to use content fingerprints to identify duplicate content [4, **?**]. Offline deduplication is used in [**?**, **?**] to remove previously written duplicate blocks during idle time. Several techniques have been proposed to speedup searching of duplicate fingerprints. For example, the data domain method [7] uses an in-memory Bloom filter and a prefetching cache for data blocks which may be accessed. An improvement to this work with parallelization is in [6, **?**]. As discussed in Section **??**, there is no dedicated resource for deduplication in our targeted setting and low memory usage is required so that the resource impact to other cloud services is minimized. The approximation techniques are studied in [1, 2, **?**] to reduce memory requirements with the tradeoff of a reduced deduplication ratio.

Additional inline deduplication techniques are studied in [3, 2, **?**]. All of the above approaches have focused on optimization of deduplication efficiency, and none of them have considered the impact of deduplication on fault tolerance in the cluster-based environment that we have considered in this paper. We will describe the motivation of using the cluster-based approach for running the backup service and then present our solution with fault isolation.

3. DESIGN CONSIDERATION AND OPTIONS

As discussed earlier, collocating the backup service on the existing cloud cluster avoids the extra cost to acquire a dedicated backup facility and reduces the network bandwidth consumption in transferring the un-deduplicated raw data for backup. We discuss the design considerations as follows.

- *Collocating with cluster-based cloud services.* We collocate the backup service with other cloud serices in a cluster and use a distributed file system [**?**, **?**] to store the backup with multiple replicas. There might be a concern that storing many versions of snapshots would go against the purpose of deduplication. The previous research shows the deduplication can compress the backup copies effectively in a 10:1 or even 15:1 rage. Assume on the average the compression ratio is c:1. When the number of versions saved in the backup storage is l on the average, the amount of storage space in the cluster allocated for the backup would be $\frac{l}{l+c}$. For example, when $l = 5$ and $c = 10$, 33% of the cluster storage is used for the backup. When $l = 9$ and $c = 13$, storage usage is 40%. That is acceptable given today's cheap cost for disk storage. In the case of Alibaba, each machine hosts 25 VMs on the average and each VM uses 40GB data. Each machine however can typically provide over 10TB storage. Thus the normal disk usage is about 1TB per machine, and adding 30% to 100% extra space for the backup is not a practical concern, especially with the added benefits of cheaper running costs and lower network bandwidth requirements.

  Since the block size in the Hadoop and GFS is uniform and large with 64MB as a default setting, the content block in a deduplication system is of nonuniform size with 4KB or 8KB on average. We need to build an intermediate layer that supports large snapshot writing with append operations and infrequent snapshot access.

- *Fault tolerance and isolation.* The distributed file system [**?**, **?**] provides the replication support and we can leverage existing open-source software to implement such a layer. When the number of machines failed exceeds the replication degree, there may be some data loss. Storage deduplication worsens the impact of such faults because the majority of each snapshot (e.g. over 90%) is deduplicated and the corresponding data blocks are shared among VMs.

  Previous work in deduplication [1, 3] has studied the solutions to reduce memory requirements. One important issue not considered in the previous work is how content sharing affects fault isolation. Because a content chunk is compared with a content signature collected from other users, this artificially creates data dependency among different VM users. In large scale cloud, node failures happen at daily basis, the loss of a shared block can affect many users whose snapshots share this data block. Without any control of such data sharing, we can only increase replication for global datasets to enhance the availability, but this incurs significantly higher costs. Loss of a small number of shared data blocks can cause the unavailability of snapshots for a large number of virtual machines.

Thus a key goal in our design is restricting the impact of data loss to a limited number of VMs. It is not desirable that small scale of data failure affects the backup of many VMs.

Because of collocation of this snapshot service with other existing cloud services, the computing resources allocated for the snapshot service is limited, especially because of its lower priority. The key usage of resource for backup is memory for storing and comparing the fingerprints. We will consider the approximation techniques with less memory consumption studied in [1, 2] along with the fault isolation consideration discussed below. Excessive use of bandwidth and computing resource for backup can create noticeable contention for resources with the existing cloud, which is not preferred and may not not be acceptible for production system operation. Thus it is desirable that backup for all nodes can be conducted in parallel and in a short period of time, and any centralized or cross-machine communication for deduplication should not become a bottleneck. For example, in an Aliyun cluster with over 1,000 nodes and each hosts over 25 VMs, the system must finish saving daily snapshots of all VMs in a few hours. Often a cloud has a few hours of light workload each day (e.g. midnight), which creates an small window for automatic backup. The backup service can generate hundreds of terabytes of data in such a time window without optimization and thus the resource content is considered carefully as we design a fault-resilient cluster-based solution.

With these considerations in mind, we study a cluster-based backup service with VM-centric deduplication. This service is co-hosted in the existing set of machines and resource usage is friendly to the existing applications. We will first discuss and analyze the integration of the VM-centric deduplication strategies with fault isolation, and then present an architecture and implementation design with deletion support.

4. VM-CENTRIC SNAPSHOT DEDUPLICATION

A deduplication scheme compares the fingerprints of the current snapshot with its parent snapshot and also other snapshots in the entire cloud. The traditional approach compares all fingerprints and stores one copy for all of a block's duplicates. We call this as the VM-oblivious (VO) approach. In comparison, we analyze the design of a VM-centric approach (VC) which differenates duplicates within a VM and cross VMs, and conducts *Inner-VM* and *cross-VM* detection seperately.

4.1 Inner and cross VM deduplication

Inner-VM duplication can be very effecive within VM's snapshots. There are typically a large number of duplicates existing among the snapshots for a single VM. Localizing the snapshot data deduplication witin a VM improves the system by: increasing data independency between different VM backups, simplifying snapshot management and statistics collection, and facilitating parallel execution of snapshot operations. On the other hand, cross-VM duplication can also be desirable mainly due to widely-used software and libraries. As the result, different VMs tend to backup large amount of highly similar data. We integrate those strategies together as follows.

- **Dirtybit-based corase-grain inner-VM deduplication.** The first-level deduplication is to follow the standand dirty bit approach, but is conducted in the coarse grain segment level. We use the Xen virtual device driver which supports changed block tracking for the storage device and the dirty bit setting is maintained in a corase grain level we call it a segment. In our implementation, the segment size is 2MB. Since every write for a block will touch a dirty bit, the device driver maintains dirtybits in memory and cannot affort a small segment size.

  It should be noted that changed block tracking is supported or can be easily implemented in many major virtualization solution vendors. VMware support it directly; the VMWare hypervisor has an API to let external backup application know the changed areas since last backup. Xen doesn't directly support it, However, their open-source architecture allows anyone to extend the device driver, thus enabling changed block tracking. We implement dirty bit tracking this way in Alibaba's platform. The Microsoft SDK provides an API that allows external applications to monitor the VM's I/O traffic, therefore changed block tracking can be implemented externally.

- **Chunk-level fine-grain inner-VM detection.** The best deduplication uses a nonuniform chunk size in the average of 4K or 8K [**?**]. This allows the system to achieve deduplication even when there are insertions/deletions which would affect many fixed-size blocks. Thus the second-level inner-VM deduplication is to perform this chunking for dirty segments, and to compare the snapshot with its parent. We load the fingerprints of block chunks of the corresponding segment from the parent and perform fingerprint matching for further inner-VM deduplication. The amount of memory for maintaining those fingerprints is small, as we only load one segment at a time. For example, with a 2MB segment, there are about 500 fingerprints to compare.

- **Cross-VM deduplication.** This step accomplishes the standand global fingerprint comparison as conducted in the previous work [**?**]. One key observation is that the inner deduplication has removed many of the duplicates. There are fewer deduplication opportunities across VMs while the memory and network consumption for global comparison is more expensive. Thus our approximation is that the global fingerprint comparison only searches for the most popular items.

When a block is not detected as a duplicate to any existing block, this block will be written to the file system. Since the backend file system typically uses a large block size such as 64MB, each VM will accululate small local blocks. We manage this accumulation process using an apendstore scheme and discuss this in details in Section **??**.

4.2 Popular Chunk Management

With cross-VM deduplication, shared data blocks create an artifical dependence among VMs and failure of one shared block affects many VMs. Thus we only maintain the index for a small set of popular chunks. The management for popular data chunks contains two aspects.

- Compute and select top-k most popular chunks. The popularity of a chunk is the number of data blocks from different VMs that are duplicates of this block after the inner VM deduplication. This number can be computed perioidically in a weekly basis. Once the popularity of all data blocks is collected, the system only maintains the top k most popular blocks. For cross-VM comparison, we only store top k items and k is chosen to be relatively small. Our analysis below will show that the algorithm can still deliver competitive deduplication effiency after making these approximations.
- Since k is small and these top k blocks are shared among multiple VMs, we can afford to provide extra replicas for these popular blocks to enhanace the fault resilence. We will provide an analysis of the fault tolerance in the next subsection.

This section analyzes how value k impacts the deduplication efficiency. The analysis is based on VM traces collected using Alibaba's production user data [**?**] which shows that the popularity of data blocks after inner VM deduplication follows a Zipf-like distribution[**?**] and its exponent α is ranged between 0.65 and 0.70.

b	the total amount of data blocks
b_u	the total amount of unique fingerprints after inner VM deduplication
C_i	the frequency for the ith most popular fingerprint
δ	the percentage of duplicates detected in inner VM deduplication
p	the number of nodes in the cluster
D	the amount of unique data on each node
B	the average block size. Our setting is 4K.
s	the average size of data containers stored in the distributed file system . Our setting is 64MB.
m	memory size on each node used by VC
F	the size of an polular data index entry

Table 1: Modeling symbols.

Let b be the total number of data blocks after localized deduplication, b_u be the total number of fingerprints in the global index after complete deduplication, and C_i be the frequency for the ith most popular fingerprint. By Zipf-like distribution, $C_i = \frac{C_1}{i^\alpha}$. Since $\sum_{i=1}^{b_u} C_i = b$,

$$C_1 \sum_{i=1}^{b_u} \frac{1}{i^\alpha} = b$$

Given $\alpha < 1$, C_1 can be approximated with integration:

$$C_1 = \frac{b(1-\alpha)}{b_u^{1-\alpha}}. \quad (1)$$

Data size (GB)	1%	2%	4%
14.6	18.6%	22.1%	31.4%
28.1	19.5%	26.2%	38.8%
44.2	21.7%	26.5%	36%
61.6	23.2%	32.9%	35%
74.2	23.6%	33.6%	37.5%

Table 2: Deduplication effectiveness of top k% of global index

The k most popular fingerprints can cover the following number of blocks.

$$C_1 \sum_{i=1}^{k} \frac{1}{i^\alpha} \approx C_1 \int_1^k \frac{1}{x^\alpha} dx \approx C_1 \frac{k^{1-\alpha}}{1-\alpha}.$$

Deduplication efficiency of VC using top k popular blocks is the percentage of duplicates that can be detected:

$$\begin{aligned} e_k &= \frac{b(1-\delta) + C_1 \frac{k^{1-\alpha}}{1-\alpha}}{b(1-\delta) + sb - b_u} \\ &= \frac{(1-\delta) + s(\frac{k}{b_u})^{1-\alpha}}{1 - \frac{b_u}{b}}. \end{aligned} \quad (2)$$

Let p be the number of physical machines in the cluster, m be the memory on each node used by the popular index, F be the size of an index entry, D be the amount of unique data on each phyiscal machine, and B be the average block size. We store the popular index using a distribued shared memory hashtable such as MemCacheD. Then k and b_u can be expressed as: $k = p * m/F$, and $b_u = p * D/B$.

The overall deduplication effiency is

$$\frac{(1-\delta) + \delta(\frac{m*B}{D*F})^{1-\alpha}}{1 - \frac{b_u}{b}}.$$

where $(\frac{m*B}{D*F})^{1-\alpha}$ represents the percentage of the remaining blocks detected as duplicates after inner VM deduplication.

When the number of machines at each cluster increases, the number of total VMs increases. Then k increases since more memory is available to host the popular block index. But for each physical machine, the number of VMs remains the same, and thus D is a constant. Then the overall deduplication efficiency of VC remains a constant.

4.3 Fault Analysis

We compare the impact of losing d machines to the the VC and VO approaches. The replication degree of the backup storage is r for regular content blocks and $r = 3$ is a typical setting in the distributed file system [**?**, **?**]. in The VC approach, a special replica degree r_c used for CDS blocks where $r_c > r$. Noted that the storage cost for VO with full deduplication is $b_u * r$ while it is

$$r * b * \delta(1 - (\frac{k}{b_u})^{1-\alpha}) + k * (r - r_c).$$

For each rank i block which appears C_i times among V virtual machines. Assigning this block to virtual machines can

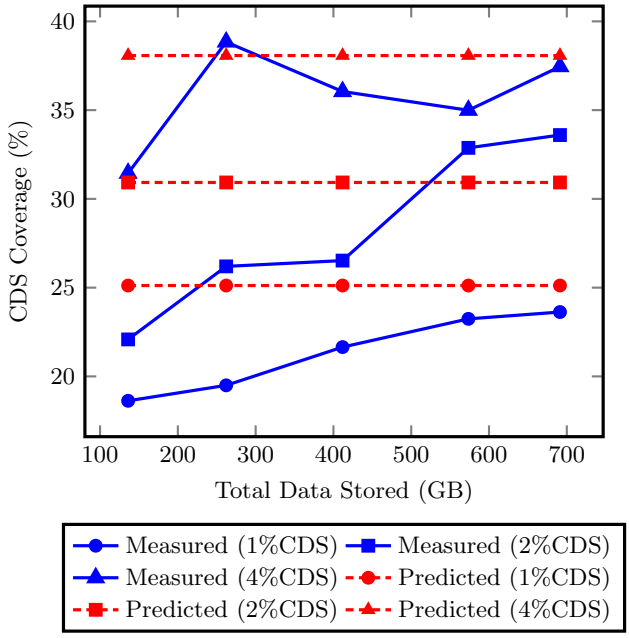


Figure 3: fixed-alpha predicted vs. actual CDS coverage as data size increases.

be viewed as a classical ball-bin assignment approximated by a Poisson distribution. Namely the virtual machines that share this block is approximated as $1 - e^{C_i/V}$. Thus the average number of VMs that share a block is:

$$(1-\delta) + \delta \sum_{1}^{u_b} \frac{1 - e^{C_i/V}}{u_b}.$$

Any failure of a block would impact the above number of virtual machines. In VO, the number of VMs that share a block is:

$$\sum_{1}^{u'_b} \frac{1 - e^{C'_i/V}}{u'_b}.$$

Next we discuss how a failure of d machines impacts the VM snapshots in VC. When $d < r$, there is no loss of data in the snapshot storage. When $r_c > d \geq r$, some of data blocks in the storage are lost, and we compute the number of VMs that could suffer the loss of their snapshots. When $d \geq r_c$, some of CDS blocks are affected. In VC, the blocks in CDS are stored in a container manner (superblock), and they are shared by many VMs. The number of data containers stored with replication degree r for referenced by each VM is:

$$N_1 = \frac{b}{v * s}(1-\delta) + b/(v*s)\delta(1 - (\frac{k}{b_u})^{1-\alpha})$$

The number of CDS data containers stored with replication degree r_c for referenced by each VM is:

$$N_2 = \frac{b}{v * s}\delta(\frac{k}{b_u})^{1-\alpha}.$$

When there are $r \leq d < r_c$ machines failed and the probability of a container failure depends on if all replicas of this container reside in the failed machines.

$$\begin{aligned} &Probability(\text{ a VM fails in VC}) \\ &= 1 - Probability(\text{ a VM has no data container loss}) \\ &= 1 - Probability(\text{ A container has no data loss})^{N_1} \end{aligned} \quad (3)$$

$$\begin{aligned} &Probability(\text{A container no data loss}) \\ &= 1 - Probability(\text{ this container is in d failed machines}) \\ &= 1 - \frac{\binom{d}{r}}{\binom{p}{r}}. \end{aligned} \quad (4)$$

When $r_c \leq d$, the CDS data containers in VC can also have a loss.

$$\begin{aligned} &Probability(\text{ a VM fails in VC}) \\ &= 1 - Probability(\text{ A container has no data loss})^{N_1} * \\ &\quad Probability(\text{ A CDS container has no data loss})^{N_2} \\ &= 1 - (\frac{\binom{d}{r}}{\binom{p}{r}})^{N_1} * (\frac{\binom{d}{r_c}}{\binom{p}{r_c}})^{N_2} \end{aligned} \quad (5)$$

By comparison, the probability that a VM has data loss is close to 1 once there are $d \geq r$ failed machines. That is because there are more data blocks shared among VMs. Any failure of these blocks causes many more VMs to fail.

The rate of increase in node failures can be seen in Figure 5 (for a 100 node cluster). The graph assumes 10X40GB snapshots per VM with 18% unique data after local dedup and 64MB filesytem chunks (which results in 1792 filesystem chunks per VM). We assume 18% unique data after local dedup because that is what our snapshots have shown.

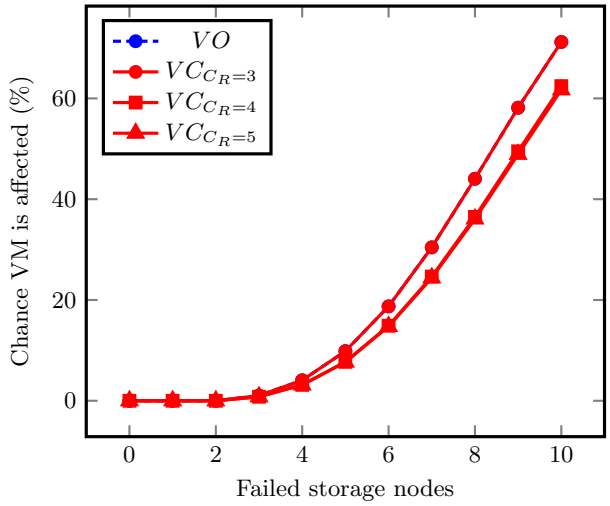


Figure 4: Percent chance that a VM is affected as storage nodes fail

Figure 6 shows the average number of links to a given block in the global index as more VMs are added. The first 15

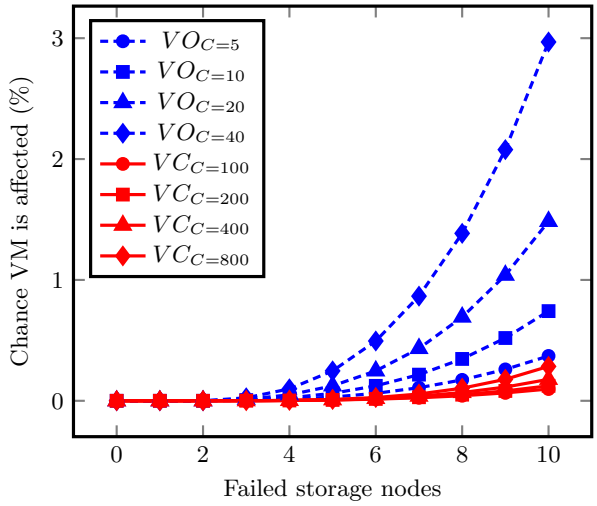


Figure 5: Percent chance that a VM is affected as storage nodes fail

VMs are all windows machines, so that explains the initial higher values, but for the most part the average number of links is between 1.5 and 2.

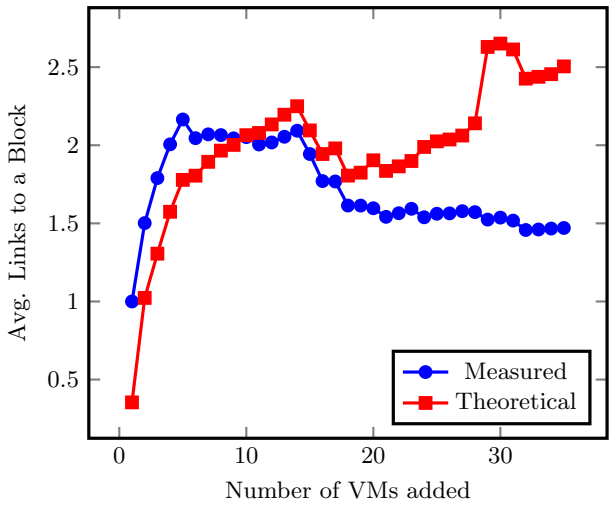


Figure 6: Average number of VMs sharing a block in the global index

5. ARCHITECTURE AND IMPLEMENTATION DETAILS

Our system runs on a cluster of Linux machines with Xen-based VMs. A distributed file system (DFS) manages the physical disk storage and we use QFS [**?**]. All data needed for VM services, such as virtual disk images used by runtime VMs, and snapshot data for backup purposes, reside in this distributed file system. One physical node hosts tens of VMs, each of which access its virtual machine disk image through the virtual block device driver (called TapDisk[5] in Xen).

5.1 Components of a cluster node

As depicted in Figure 7(a), there are four key service components running on each cluster node for supporting backup and deduplication: 1) a virtual block device driver, 2) a snapshot deduplication component, 3) an append store client which provides facilities to store and access snapshot data, and 4) a CDS client to support CDS index access. We will further discuss our deduplication scheme in Section **??**.

We use the virtual device driver in Xen that employs a bitmap to track the changes that have been made to virtual disk. When the VM issue a disk write, the bits corresponding to the segments that covers the modified disk region are set, thus letting snapshot deduplication component knows these segments must be checked during snapshot backup. After the snapshot backup is finished, snapshot deduplication component acknowledges the driver to resume the dirty-bits map to a clean state. Every bit in the bitmap represents a fix-sized (2MB) region called *segment*, indicates whether the segment is modified since last backup. Hence we could treat segment as the basic unit in snapshot backup similar to file in normal backup: a snapshot could share a segment with previous snapshot it is not changed. As a standard practice, segments are further divided into variable-sized chunks (average 4KB) using content-based chunking algorithm, which brings the opportunity of fine-grained deduplication by allowing data sharing between segments.

The representation of each snapshot has a two-level index data structure. The snapshot meta data (called snapshot recipe) contains a list of segments, each of which contains segment metadata of its chunks (called segment recipe). In a snapshot recipes or a segment recipe, the data structures includes reference pointers to the actual data location.

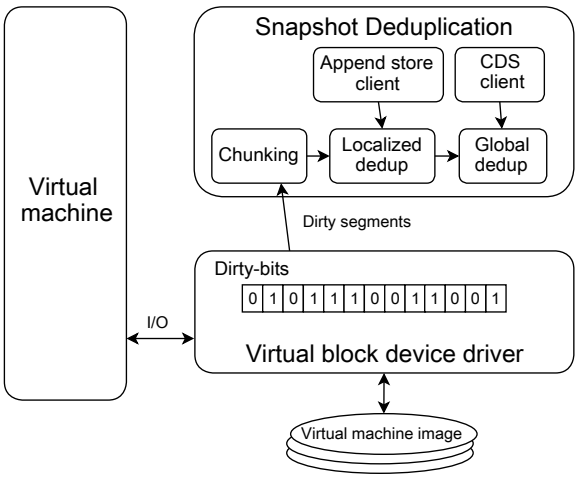
5.2 Append store for backup data

The Append Store (AS) is our underlining storage engine for storing snapshot data in the distributed file system after deduplication.

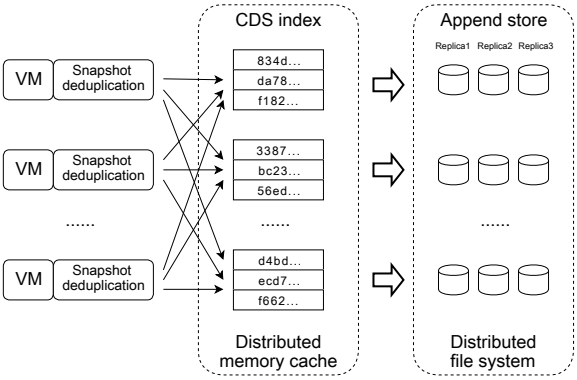
AS supplies three interfaces: *get(ref)* accepts a data reference and retrieves data, *put(data)* accepts data and returns a reference to be stored in metadata recipes, *delete(ref)* deletes the data pointed by the reference. Under the hood, small var-sized data are grouped and stored into larger data containers. Each VM has its snapshot data stored in its own Append Store, specified by the VM ID. We split every Append Store into multiple data containers so that reclaiming the disk space would not result in rewriting all the data at the same time.

As shown in Fig.8, every data container is represented as three data files in DFS: the data file holds all the actual data, the index file is responsible for translating data reference into data locations, and a deletion log file remembers all the deletion requests to the container.

A data reference is composed of two parts: a container ID (2 bytes) and CID (6 bytes). Append Store assign every piece of data a CID for its internal data referencing. When new data is appended, its CID is the current largest CID in that container plus one. As a result, all the data locations are naturally indexed by this self-incremental CID, no extra sorting is needed.



(a) Node architecture from VM point of view



(b) Cluster architecture

Figure 7: System architecture.

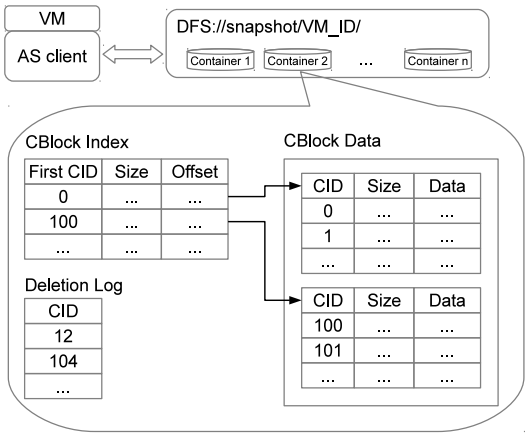


Figure 8: Architecture of Append Store

Append Store groups multiple chunk data (i.e., 100) into larger units, called *CBlock*. CBlock is the basic unit for append store's internal read/write/compression. There is one index entry in the container index corresponding to every CBlock. It keeps the first chunk's CID in that CBlock, and the CBlock data's size and location.

Using CBlock brings us several advantages: First, the write workload to DFS master is greatly reduced; second, grouping small chunks gives better compression. Third, reading a CBlock (200 - 600 KB) typically cost the same amount of disk seek as reading a 4KB chunk. Finally, this greatly reduces the size of index. Let m be the number of chunks in each CBlock, then the overall index size is reduced to $1/m$. In our implementation, using $m = 100$ reduces the index for a 1GB container from 10 MB to 100 KB.

In order to read a chunk data by reference, Append Store client first loads the container index file specified by the container ID, then search the CBlock index to find the entry that covers the chunk by CID. After that, it reads the whole CBlock data from DFS, decompress it, seek the exact chunk data specified by CID. Finally, the client updates itws internal chunk data cache with the newly loaded contents to anticipate future sequential reads.

Write requests to append store are accumulated. When the number reaches m, the AS client forms a CBlock by assigning every chunk a CID, compress the CBlock data, and append it to the CBlock data file. Then a new CBlock index entry is appended to CBlock index.

Append store adopts lazy delete strategy. The deletion requests are appended into every container's deletion log file with the CID of data to be deleted. CIDs in deletion log are guaranteed to be referenced by nobody and can be safe deleted in future. Periodically, snapshot management system asks append store to compact containers in order to reclaim disk space. The actual compaction will only take place when the number of deleted items reached $d\%$ of container's capacity. During compaction, append store creates a new container (with the same container ID) to replace the existing one. This is done by sequentially scan the old container, copying all the chunks that are not found in deletion log to the new container, creating new CBlocks and indices. However, every chunk's CID is plainly copied rather than re-generated. This does not affect the sorted order of CIDs in new container, but just leaving holes in CID values. As the result, all data references stored in upper level recipes are unaffected, and the data reading process is as efficient as before.

5.3 Snapshot Summaries for Approximate Deletion

In a busy VM cluster, snapshot deletions are as frequent as snapshot creations. Our system adopts lazy delete strategy so that all snapshot deletions are scheduled in the backup time window at midnight. Therefore, snapshot deletions must be fast enough to fit in time window and efficient enough to satisfy our resource constraints. However, there is no simple solution can achieve these goals with high reliability. Hence we designed a two-phase *approximate deletion* strategy to trade deletion accuracy for speed and resource

usage. Our method sacrifaces a tiny percentage of storage leakage to effectively identify unused blocks in $O(n)$ speed, with n being the logical amount of blocks to be deleted.

Snapshot Summaries

Approximate Deletion Phase-1 The goal of approximate deletion phase-1 to fast identify unused blocks which are no longer referenced by other snapshots after a snapshot deletion. Instead of scanning the entire append store indices, we merge the type-1 summaries of all valid snapshots. Since each VM has uniform bloom filter parameters to create snapshot summaries, such merged summeries give us a compact representation of all block fingerprints that are still in use. Thus by the property of bloom filter, if a fingerprint is not found in merged summaries, we are certain that block is no longer used by any valid snapshot, it would be then added to append store's deletion log. However, there is a small false-positive probility which would indetify unused data as in use, resulting in temporary storage leakage.

Approximate Deletion Phase-2 We designed the second phase of approxmite deletion to solve the temporary storage leakage problem mentioned above. In this phase we scan the entire append store indices, using the merged type-2 snapshot summaries to check if any of them are not referenced by existing snapshots. We cannot simply repeat the phase 1 multiple times to reduce temporary storage leakage, because:

1. After several runs of phase-1, it is proven that the merged type-1 summaries cannot sieve remaining unused blocks, due to the false-positive property of bloom filter.

2. The recipes of deleted snapshots have been removed from the system, thus we are not able to obtain the deleted block fingerprints from any metadata, the only way to discover them is to scan the append store indices.

Discussion For one VM's snapshots, let L_{temp} be the amount of temporary storage leakage, D_{del} be the average amount of data that should be physically deleted in one snapshot deletion, P_1 be the false-positive rate of merged bloom filter type-1, N_1 be the number of runs of deletion phase-1 during the gap of two deletion phase-2 operations, then we have:

$$L_{temp} = N_1 * P_1 * D_{del} \tag{6}$$

If we let approxmite deletion phase-2 be triggered when L_{temp}/D_{del} is accmulated to exceed certain threshold t, then:

$$\frac{L_{temp}}{D_{del}} = N_1 * P_1 > t \Rightarrow N_1 > \frac{t}{P_1} \tag{7}$$

Let P_2 be the false-positive rate of merged snapshot type-2 summaries, N_2 be the number of runs of deletion phase-2, L_{perm} be the permanent storage leakage resulting from the approxmite deletion phase-2, it can be calculated as follows:

$$L_{perm} = N_2 * P_1 * P_2 * D_{del} \tag{8}$$

For example, letting $P_1 = 0.01$ and $t = 1.0$, we would expect deletion phase-2 be triggered once for every $T/P_{bl} = 100$ runs of deletion phase-1. On a node that hosts 20 VMs and each VM deletes one snapshot per day, there would be only 1 deletion phase-2 scheduled for every 5 days, which is sufficiently small to prevent the heavy I/O workload of deletion phase-2 from disturbing normal VM opeations.

6. IMPLEMENTATION

6.1 Snapshot Backup

6.2 Snapshot Read

6.3 Snapshot Deletion

The following steps would take place during an approximate deletion:

1. **Creating bloom filter** Scan all the living snapshot recipess and their segment recipes, for every reference pointing to append store, add it to the bloom filter.

2. **Check existance** For every data reference in the deleted snapshot recipe and its segment recipes, check the existance of that data reference in bloom filter. If not found, it is safe to delete that piece of data from append store because no living snapshots has referenced it.

The overall time of running a approximate deletion for one snapshot deletion would be scanning all the living snapshots and deleted snapshots, since operations on the in-memory bloom filter can be done in parallel and is much faster than loading recipes from DFS:

$$T = (N_{SS} + 1) * T_{scan\_recipes} \tag{9}$$

Using the example and analysis in previous section, this approximate deletion can be done in 5 minutes. Memory usage of the bloom filter depends on its false-positive probility P_{bl}, when set P_{bl} to 0.01, the memory footprint of approximate deletion is about 15 MB. In evaluating our approach, we also looked at other methodologies. One methodology which provides a very low memory footprint system is maintaining a subsampled index[2]. By indexing a subsampling of the data and pulling entire containers into a cache on an index hit, as long as there is locality between data in the store and data in inputs to be deduplicated, and the input data causes index hits, a high rate of deduplication can be acheived. In our small scale testing, subsampling performed quite well (acheiving over 98deduplication for deduplicating highly similar VM traces). The main disadvantage of subsampling is that it doesn't scale easily to many nodes. subsampling performance doesn't scale as well because it requires global data to perform all deduplication (each index hit could pull in a container from anywhere), and because index size is directly proportional to storage size, which requires linear increase in memory as target storage size increases. With a sampling rate of 1/101 4KB blocks and 22 byte entries, the index requires a minimum of 55GB RAM per 1PB of storage, not including cache. By performing most deduplication locally, the performance becomes much more scalable to many nodes as most deduplication doesn't require network traffic, and could even theoretically allow

most of the dedulication to proceed during a complete network partition. Performing local deduplication also reduces the amount of global data required for high efficency, and our tests show that the CDS size increases at a sublinear rate (see Sec.**??**).

7. CONCLUSION

8. REFERENCES

[1] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–9, 2009.

[2] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. page 25, June 2011.

[3] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST*, pages 111–123, 2009.

[4] 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, Berkeley, CA, USA, 2002. USENIX Association.

[5] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. page 22, Apr. 2005.

[6] J. Wei, H. Jiang, K. Zhou, and D. Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–14, May 2010.

[7] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.