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 (VMs) has a high storage demand for frequent backup of system image snapshots and signature-based deduplication of data blocks is necessary to eliminate excessive redundant blocks. Collocating a cluster-based duplicate service with other cloud services reduces network traffic; however it is resource expensive and less fault-resilient to perform a global deduplication and let a data block share by many virtual machines. This paper proposes a VM-centric collocated backup service that localizes deduplication as much as possible within each virtual machine and associates underlying file blocks with one VM for most of cases. Our analysis shows that this VM centric scheme can provide better fault tolerance while using a small amount of computing and storage resource. This paper describes 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 [7, 12] can effectively identify and remove content duplicates among snapshots, but such a 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 that collocates a backup service with other cloud services and uses a minimum amount of resources. We also consider the fact that after deduplication, most data chunks are shared by several to many virtual machines. Failure of shared data chunks can have a catastrophic effect and many snapshots of virtual machines would be affected. The previous work in deduplication focuses on the efficiency and approximation of finger print comparison, and has not addressed fault tolerance together with deduplication. Thus we also seek deduplication options that yield better fault isolation. Another issue considered is that that deletion of old snapshots compete for computing resource as well. sharing of data chunks among oby multiple VMs and their snapshots needs to be detected during snapshot deletion and such dependenec complicates deletion operations.

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 and by packaging data chunks from the same VM into a file system block as much as possible. By narrowing duplicate sharing within a small percent of data chunks, this scheme can afford to allocate extra replicas of these shared chunks for better fault resilience. Localization also brings the benefits of greater ability to exploit parallelism so backup operations can run simultaneously without a central bottleneck. This VM-centric solution uses a small amount of memory while delivering a reasonable deduplication efficiency. We have developed a prototype system that runs a cluster of Linux machines with Xen and uses a standard distributed file system for the backup storage.

The rest of this paper is organized as follows. Section **??** reviews the background and discusses the design options for snapshot backup with a VM-centric approach. Section **??** analyzes the tradeoff and benefit of our approach. Section 4 describes our system architecture and implementation details. Section **??** is our experimental evaluation that compare with the other approaches. Section **??** concludes this paper.

2. BACKGROUND AND DESIGN CONSIDERATIONS

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. Backup systems have been developed to use content fingerprints to identify duplicate content [7, **?**]. As discussed earlier, collocating a 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. Figure 1 illustrates the cluster architecture where each physical runs a backup service and a distributed file system (DFS) [**?**, **?**] serves a backup store for the snapshots. The previous study shows that deduplication can compress the backup copies effectively in a 10:1 or even 15:1 rage. Therefore the portion of space in a cluster allocated for snapshots of data should not dominate the cluster storage usage.

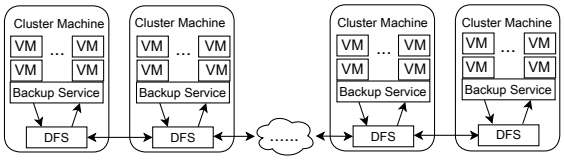


Figure 1: Collocated VM Backup System.

Since it is expensive to compares a large number of chunk signatures for deduplication, several techniques have been proposed to speedup searching of duplicate fingerprints. For example, the data domain method [12] uses an in-memory Bloom filter and a prefetching cache for data chunks which may be accessed. An improvement to this work with parallelization is in [11, **?**]. The approximation techniques are studied in [2, 4, **?**] to reduce memory requirements with the tradeoff of a reduced deduplication ratio.

Additional inline deduplication techniques are studied in [5, 4, 9]. 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.

Our key design consideration is VM dependence minimization during deduplication and file block management.

- *Deduplication localization.* Because a data chunk is compared with signatures collected from all VMs during the deduplication process, only one copy of duplicates is stored in the storage, this artificially creates data dependency among different VM users. Content sharing via deduplication affects fault isolation since machine failures happen periodically in a large-scale cloud and loss of a small number of shared data chunks can cause the unavailability of snapshots for a large number of virtual machines. Localizing the impact of deduplication can increase fault isolation and resilience. Thus from the fault tolerance point of view, duplicate sharing among multiple VMs is discouraged. Another disadvantage of sharing is that it complicates snapshot deletion, which occurs frequently when snapshots expire regularly and leads a cost concern. The mark-sweep approach [**?**] is effective for deletion, and its main cost is to count if a data chunk is still shared by other snapshots. Localizing deduplication can minimize data sharing and simplify deletion while sacrificing deduplication efficiency, and can faciltate parallel execition of snaptshot operations. Thus we seek for a tradeoff.

- **Management of file system blocks.** The file block size in a distributed file system such as Hadoop and GFS is uniform and large (e.g. 64MB), a data chunk in a deduplication service is of a nonuniform size with 4KB or 8KB on average. Packaging data chunks to a file system block can create more data dependence among VMs since a file block can be shared with even more VMs. Thus we need to consider a minimum association of a fie system block to VMs in the packaging process.

Another consideration is the computing cost of deduplication. Because of collocation of this snapshot service with other existing cloud services, cloud providers wish that the backup service only consumes small resources with a minimal impact to the existing cloud services. The key resource for signature comparison is memory for storing the fingerprints. We will consider the approximation techniques with less memory consumption along with the fault isolation consideration discussed below.

We call the classical deduplication approach as VM-oblivious (VO) because they compare fingerprints of snapshots without consideration of VM. With the above considerations in mind, we study a VM-centric approach (called VC) for a collocated backup service with resource usage friendly to the existing applications. In designing a VC duplication algorithm, we have considered and adopted some of the following previously-developed techniques.

- **Version-based change detection**. VM snapshots can be backed up incrementally by identifying file blocks that have changed from the previous version of the snapshot [3, **?**, **?**]. Such a scheme is VM-centric since deduplication is localized. We are seeking for a tradeoff since cross-VM signature comparison can deliver additional reduction [4, **?**, **?**].

- **Stateless Data Routing**. One approach for scalable duplicate comparison is to use a content-based hash partitioning algorithm called stateless data routing [**?**] that divides the deduplication work with an approximation. This work is similar to Exreme Binning[2] and each request is routed to a machine which holds a bloomer filter or can fetch on-disk index for additional comparison. While this approach is VM-oblivious, it motivates us to use a combined signature of a dataset to narrow VM-specific local search.

- **Sampled Index**. One effective approach that reduces memory usage is to use a sampled index with prefetching, proposed by Guo and Efstathopoulos[4]. The algorithm is VM oblivious and it is not easy to adopt for a distributed architecture. To use a distributed memory version of the sampled index, every deduplicate request may access a remote machine for index lookup and the overall overhead of latency latency for all requests can be significant.

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.

3. VM-CENTRIC SNAPSHOT DEDUPLICATION

Our VC design has the following objectives: 1) localizing deduplication while identifying a decent amount of duplicates among VMs to maintain a competitive deduplication efficiency and simplify snapshot management; 2) Minimizing the number of file system blocks shared among VMs and adding extra replicas for shared blocks.

3.1 Key VM-centric Strategies

- **VM-specific local duplicate search within similar segments.** We start with the standard dirty bit approach in a coarse grain segment level. In our implementation with Xen at an Alibaba platform, the segment size is 2MB and the device driver is extended to support changed block tracking. Since every write for a segment will touch a dirty bit, the device driver maintains dirty bits in memory and cannot afford a small segment size. It should be noted that dirtybit tracking is supported or can be easily implemented in major virtualization solution vendors. For example, the VMWare hypervisor has an API to let external backup applications know the changed areas since last backup. The Microsoft SDK provides an API that allows external applications to monitor the VM's I/O traffic and implement the changed block tracking feature.

  Since the best deduplication uses a nonuniform chunk size in the average of 4K or 8K [**?**], we conduct additional local inner-VM deduplication by comparing dirty segment's chunk fingerprints to similar segments from its parent snapshot. For every segment, the minimum value of all its chunk fingerprints is computed during backup and is recorded in the snapshot recipe. When we are processing a dirty segment, we can easily find similar segments from the parent snapshot recipe and load their segment recipes. Then given a set of data chunks within a dirty segement, we compare these chunk fingerprints with those in similar segments. For example, with a 2MB segment, there are about 500 fingerprints to compare. The amount of memory for maintaining those fingerprints in similar segment s is small, as we only load one dirty segment at a time.

- **Cross-VM deduplication with popular chunks and replication support** This step accomplishes the standard 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 top k most popular items. This dataset is called PDS (common data set). The popularity of a chunk is the number of data chunks from different VMs that are duplicates of this chunk after the inner VM deduplication. This number can be computed periodically on a weekly basis. Once the popularity of all data chunks is collected, the system only maintains the top k most popular chunk signatures in a distributed shared memory.

  Since k is relatively small and these top k chunks are shared among multiple VMs, we can afford to provide extra replicas for these popular chunks to enhance the fault resilience.

- **VM-centric file system block management.** When a chunk is not detected as a duplicate to any existing chunk, this chunk will be written to the file system. Since the backend file system typically uses a large block size such as 64MB, each VM will accumulate small local chunks. We manage this accumulation process using an append-store scheme and discuss this in details in Section 4. The system allows all machines conduct the backup in parallel in different machines, and each machine conducts the backup of one VM at a time, and thus only requires a write buffer for one VM.

  PDS chunks are stored in a separate append-store instance. In this way, each file block for non-PDS chunks is associated with one VM and does not contain any PDS chunks.

We have not adopted the sampled index [4] for popular data chunks and this is because sampling requires the use of prefetching to be effective. For the data sets we have tested, the spatial locality is limited among popular data chunks and on average the number of consecutive data chunks is 7 among popular chinks, which is not sufficient.

3.2 Impact on deduplication efficiency

We analyze how the choice of value k impacts the deduplication efficiency. The analysis is based on the characteristics of the VM snapshot traces studied from Alibaba's production user data. Our previous study shows that the popularity of data chunks after inner VM deduplication follows a Zipf-like distribution[**?**] and its exponent α is ranged between

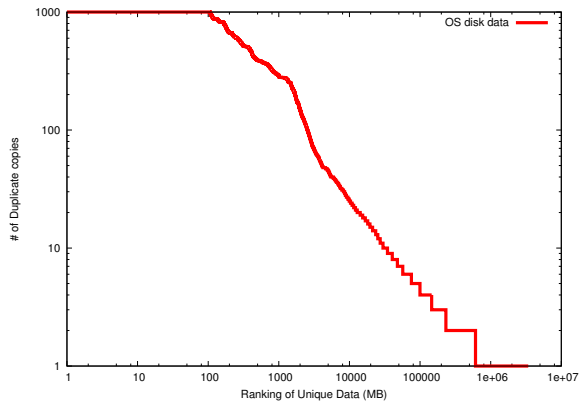


Figure 2: Duplicate frequency versus chunk ranking in a log scale.

0.65 and 0.7. [**?**]. Table 1 lists parameters used in this analysis.

[Need to find a place to put these numbers in: Total number of chunks in 350 snapshots: 1,546,635,485. Total number of chunks after localized dedup: 283,121,924. Total number of unique chunks: 87,692,682.]

As summarized in Table 1, let c be the total number of data chunks. c_u be the total number of fingerprints in the global index after complete deduplication, and f_i be the frequency for the ith most popular fingerprint. By Zipf-like distribution, $f_i = \frac{f_1}{i^\alpha}$. Since $\sum_{i=1}^{c_u} f_i = c$,

$$f_1 \sum_{i=1}^{c_u} \frac{1}{i^\alpha} = c.$$

Given $\alpha < 1$, f_1 can be approximated with integration:

$$f_1 = \frac{c(1-\alpha)}{c_u^{1-\alpha}}. \quad (1)$$

The k most popular fingerprints can cover the following number of chunks after inner VM deduplication:

$$f_1 \sum_{i=1}^{k} \frac{1}{i^\alpha} \approx f_1 \int_1^k \frac{1}{x^\alpha} dx \approx f_1 \frac{k^{1-\alpha}}{1-\alpha}.$$

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

$$\frac{c(1-\delta) + f_1 \frac{k^{1-\alpha}}{1-\alpha}}{c(1-\delta) + \delta c - c_u} = \frac{(1-\delta) + \delta(\frac{k}{c_u})^{1-\alpha}}{1 - \frac{c_u}{c}}. \quad (2)$$

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

k	the number of top most popular chunks
c	the total amount of data chunks
c_u	the total amount of unique fingerprints after inner VM deduplication
f_i	the frequency for the ith most popular fingerprint
δ	the percentage of duplicates detected in inner VM deduplication
σ	the number of unique non-PDS chunks over the number of the PDS chunks.
p	the number of machines in the cluster
V	the number of VMs on each machine
D	the amount of unique data on each machine
C	the average data chunk size. Our setting is 4K.
s	the average size of file system blocks in the distributed file system. The default is 64MB.
m	memory size on each node used by VC
E	the size of an popular data index entry
N_1	the average number of non-PDS file system blocks in a VM
N_2	the average number of PDS file system blocks in a VM
N_o	the average number of file system blocks in a VM for VO

Table 1: Modeling parameters and symbols.

The overall deduplication efficiency of VC is

$$\frac{(1-\delta) + \delta(\frac{m*C}{D*E})^{1-\alpha}}{1 - \frac{c_u}{c}}.$$

where $(\frac{m*C}{D*E})^{1-\alpha}$ represents the percentage of the remaining chunks detected as duplicates after inner VM deduplication. Figure 3 shows measured PDS coverage in our 35 VM dataset. You can see from the graph that the coverage actually increases as the data size increases, which indicates that α increases with the data size (a fixed-α model would predict constant coverage). This makes the PDS even more effective, as the coverage of the PDS increases as more data is added.

As illustrated in Figure **??**, 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 chunks 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.

3.3 Storage Space and Impact on Fault Tolerance

The replication degree of the backup storage is r for regular file 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 PDS blocks where $r_c > r$. Notice that the

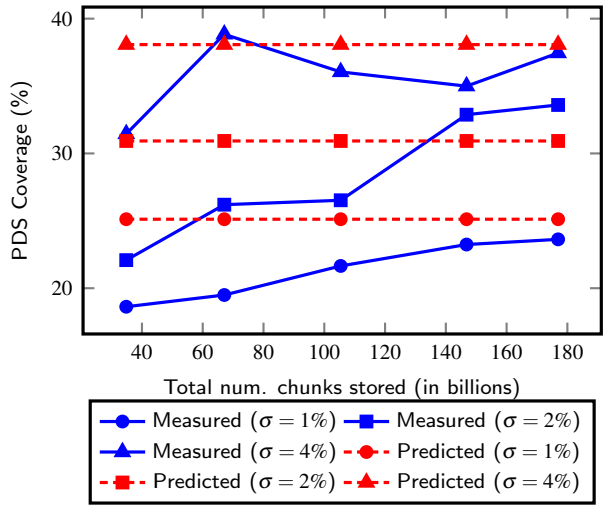
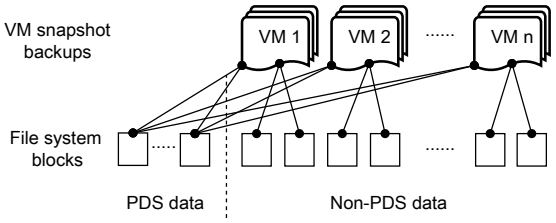
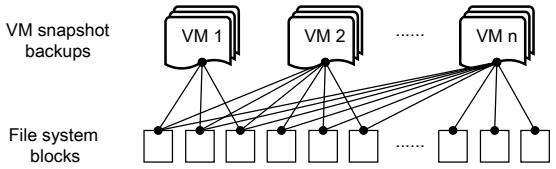


Figure 3: fixed-alpha predicted vs. actual PDS coverage as data size increases.



(a) Sharing of data under VM-oblivious dedup model



(b) Sharing of data under VM-centric dedup model

Figure 4: Difference of sharing data under VO and VC approaches

ratio of non-PDS data size vs PDS data size for each VM is

$$\sigma = \frac{c * \delta(1 - (\frac{k}{c_u})^{1-\alpha})}{k}.$$

Thus storage cost for VO with full deduplication is $c_u * r$ and for VC, it is

$$k * r_c + k * \sigma * r.$$

In our experiment with Alibaba data, the ratio σ is 162. Thus allocation of extra replicas for PDS only introduces a small amount of extra space cost. Figure **??** shows the storage cost ratio of VC and VO when r=3, and r_c varies from 3 to 10. The result shows that the storage cost for adding extra replication for PDS is insignificant.

Next we compare the impact of losing d machines to the the VC and VO approaches.

In characterizing the reliability of VM backups in our model, we consider the likely hood that a file system block fails, given some number of storage machine failures. Every time a filesystem block fails, we say that we have lost data for that virtual machine, so it is no longer available. In reality it may be only one snapshot that is affected, but it is the user who must decide which snapshots are important, so we consider the worst case. We use filesystem blocks rather than a deduplication data chunk as our unit of failure because the DFS keeps filesystem blocks as its base unit of storage.

To compute the probablity of losing snapshots of a virtual machine, we estimate the number of file system blocks per VM in each apporach. We can build a bipartite graph representing the association from unique file system blocks to their corresponding VMs. An association edge is drawn from a file block to a VM if this file is used by this VM. For VC, each VM has an averge number of N_1 file system blocks for non-PDS data. It also refers an average of N_2 file system blocks for PDS data For VO, each VM has an average of N_o file system blocks and let V_o be the averge number of VMs shared by each file system block. Figure **??** illustates the bipartite association.

In VC, each non-PDS file system block is associated with one VM while PDS file system blocks are shared among VMs (at most V VMs). Thus,

$$V * N_1 * s = pD\frac{\delta}{\delta+1} \text{ and } V * N_2 * s \leq pD\frac{1}{\delta+1} * V.$$

For the VO approach,

$$V * N_o * s = pDV_o.$$

Then

$$N_1 = \frac{pD\delta}{Vs(\delta+1)}, N_2 \leq \frac{pD}{s(\delta+1)}, \text{ and } N_o = \frac{pDV_o}{sV}.$$

Since each file block (with default size $s = 64MB$) contains many chunks (on average 4KB), each file block contains the hot low-level chunks shared by many VMs, and it also contains rare chunks which are not shared. Figure 5 shows the number of VMs sharing by each file block. In our experiment, we find that $V_o \approx 0.2V$ when backing up VMs one by one. We can observe in Figure 6 that $N_1 + N_2 < N_o$. This is likely because the PDS FSBs tightly pack data used by many VMs, which decreases the overall number of FSBs required to backup a VM. If the backup for multiple VMs is conducted concurrently, there would be more VMs shared by each file block on average.

Figure 5 shows the average number of VMs sharing a filesystem block (FSB) as VMs are added. Though we don't expect the linear trend to continue indefinitely for much larger datasets, it should continue to increase, which has important implications on VM backup availability in the precense of failures. Below we show the significant impact the number of links has on VM backup reliability.

The snapshot availability of is a VM is the likelyhood that

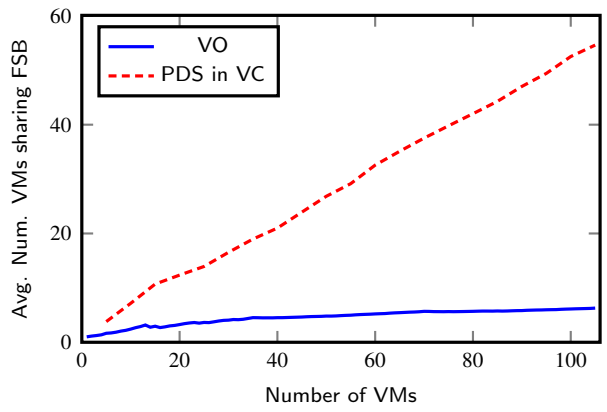


Figure 5: Measured Average number of VMs sharing a 64MB FSB with global dedup (VO), and in a 2% PDS for VC.

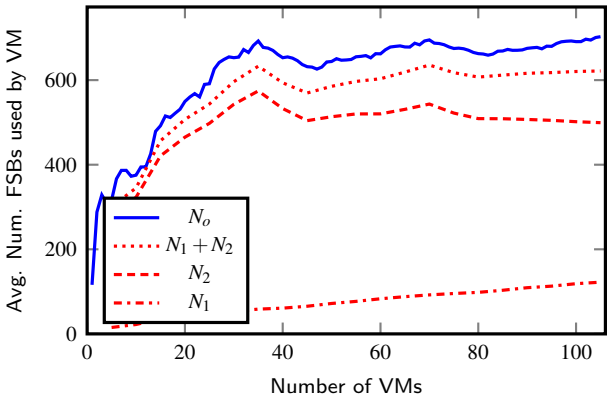


Figure 6: Measured Average number of 64MB FSBs used by a single VM. For VC both the number of PDS and Non-PDS FSBs used are shown.

there is no data loss for all its file blocks. With replication degree r, the likelyhood of a file block block is the probability that all of its replicas appear in d failed machines. Namely, $\binom{d}{r}/\binom{p}{r}$. This can be seen in Figure 7 for a 100 machine cluster for 3 different replication factors.

When there are $r \leq d < r_c$ machines failed and then there is no PDS data loss, the snapshot avaiability of a VM in the VC approach is and is

$$(1-\frac{\binom{d}{r}}{\binom{p}{r}})^{N_1}.$$

When $r_c \leq d$, both non-PDS and PDS file blocks in VC can have a loss. The snapshot availability of a VM in the VC approach is

$$(1-\frac{\binom{d}{r}}{\binom{p}{r}})^{N_1} * (1-\frac{\binom{d}{r_c}}{\binom{p}{r_c}})^{N_2}.$$

The snapshot availability of a VM in the VO approach is

$$(1-\frac{\binom{d}{r}}{\binom{p}{r}})^{N_o}.$$

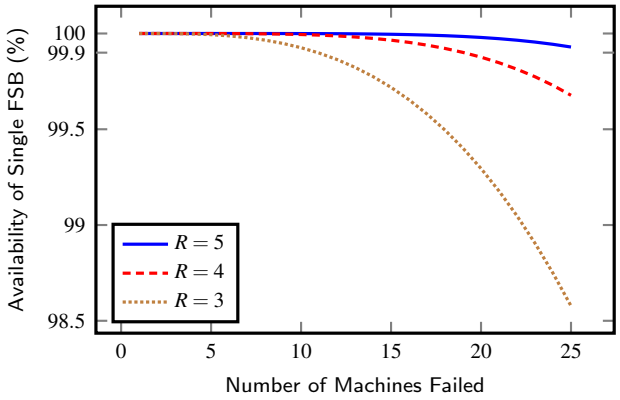


Figure 7: Availability of Individual FSBs in 100 Machine Cluster with different replication factors.

4. 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[10] in Xen).

4.1 Components of a cluster node

As depicted in Figure 8, 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 to store and access snapshot data, and 4) a PDS client to support PDS index access. We will further discuss our deduplication scheme in Section 3.

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 reset the dirty-bits map to a clean state. Every bit in the bitmap represents a fix-sized (2MB) region called a *segment*, indicates whether the segment is modified since last backup. Hence we think of segment as the basic unit in snapshot backup similar to file in normal filesystem 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 a 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 snapshot and segment recipes, the data structures includes reference pointers to the actual data location to eliminate the need for additional indirection.

4.2 A VM-centric snapshot store for backup data

We build the snapshot storage on the top of a distributed file system. Following the VM-centric idea for the purpose of fault isolation, each VM has its own snapshot store, containing new data chunks which are considered to be non-duplicates. There is also a special store containing all PDS chunks shared among different VMs. This separation of PDS chunks allows us to change the replication degree of of those popular file blocks in the underlying file system Extra replication of this store is added and analyzed in Section **??**. As shown in Fig.9, we explain the data structure of snapshot stores as follows.

- The PDS snapshot contains a set of commonly used data chunks and is accessed by its offset and size in the corresponding DFS file. The PDS index uses the offset and size as a reference in its index structure.

- Each non-PDS snapshot store is divided into a set of containers and each container is approximately 1GB. The reason we divide the snapshot into containers is to simplify the compaction process conducted periodically. Chunks without any reference from other snapshots can be removed by this process, and the compaction routine can work on one container at a time and copy used data chunks to another container. By limiting the size of a container, we limit the cost of the compaction process.

  – Each container is divided into a set of chunk data groups. Each group is composed of a set of data chunks and is the basic unit for the snapshot in data access and retrieval. In writing a chunk group, data chunks in this group are compressed together and then stored. When accessing a particular chunk, its chunk group is retrieved from the disk storage and uncompressed. Given the high spatial locality in snapshot data accessing [**?**, **?**], retrieval of data chunks by group naturally works well with prefetching to speedup snapshot access. A typical chunk group contains 100 to 1000 chunks, with an average size of 200-600 chunks. Chunk grouping also reduces the container index size as we discuss below. Given the average chunk size of 4KB, the index size for a 1GB container reduces from 10MB to 100KB when the chunk group size is 100.

  – Each data container is represented by three data files in the DFS: 1) the container data file holds the actual content of data chunks, 2) the container index file is responsible for translating a data reference into its location within a container, and 3) a chunk deletion log file saving all the deletion requests within the container. A VM snapshot store typically has a small number of containers because each container is fairly large, with an average size of 1GB, and maintains a limited number of snapshots (e.g. 10 in the Alibaba case). New snapshot data chunks can be effectively compressed in chunk groups in addition to deduplication.

  – We maintain a chunk counter and assign the current number as a chunk ID (called CID) within this container as a reference of a new chunk added to a container. Since data chunks are always appended to the snapshot store, a CID is monotonically increasing. A data chunk reference stored in the index of snapshot recipes is composed of two parts: a container ID (2 bytes) and CID (6 bytes). When a snapshot chunk is to be accessed, the recipe for the snapshot will point to either a data chunk in the PDS or a reference number with a container ID and CID. With a container ID, the corresponding container index file is accessed and the chunk group is identified using this CID. Once the chunk group is loaded to memory, its header contains the exact offset of the corresponding chunk and the content is then accessed from the memory buffer.

The snapshot store supports three API calls.

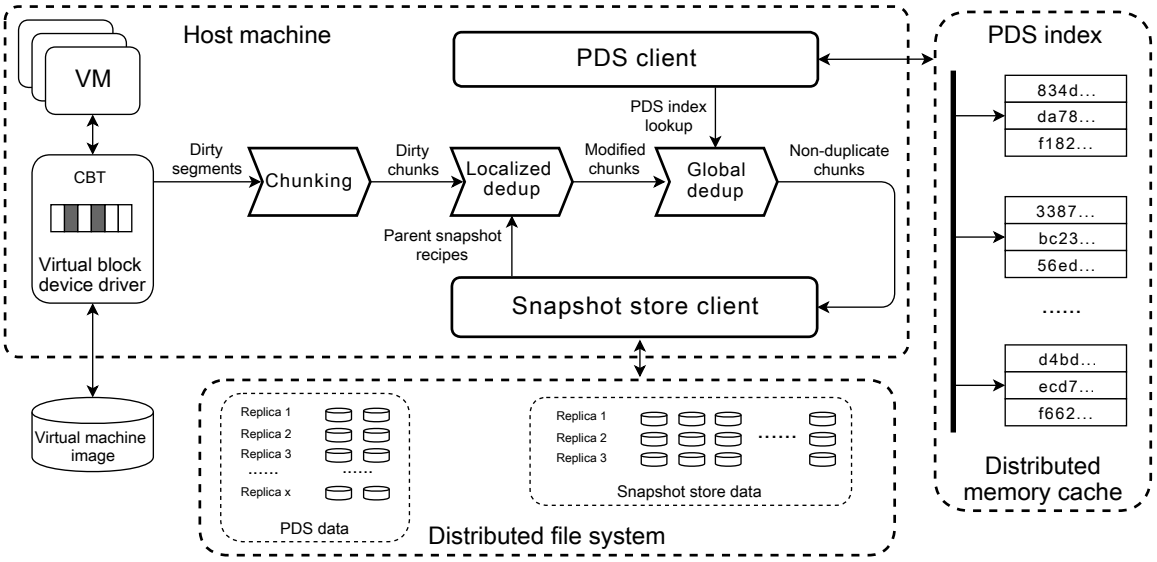


Figure 8: System Architecture

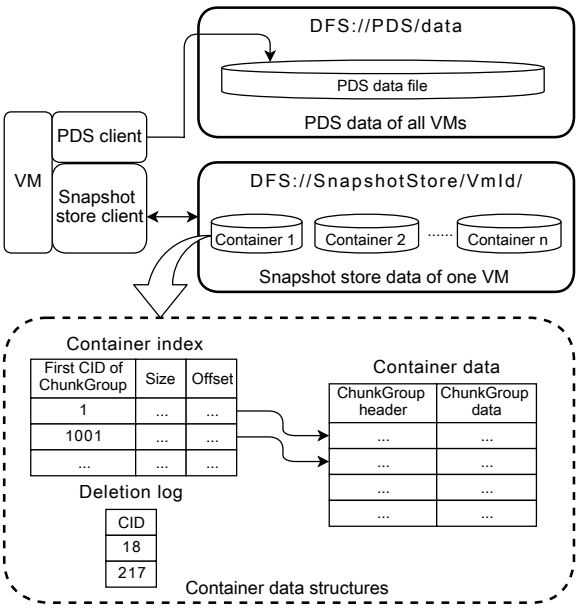


Figure 9: Data structure of VM snapshot stores.

- *Put(data)* places data chunk into the snapshot store and returns a reference to be stored in the recipe metadata of a snapshot.

  The write requests to append data chunks to a VM store are accumulated in the client side. When the number of write requests reaches the group size g, the snapshot store client compresses the accumulated chunk group, adds a chunk group index to the beginning of the group, and then appends the header and data to the corresponding VM file. A new container index entry is also created for each chunk group and is written the corresponding container index file.

  The writing of PDS data chunks is conducted periodically when there is a new PDS calculation. Since the PDS dataset is small, a new PDS file is created during the periodical update.

- *Get(reference)*. The fetch operation for the PDS data chunk is straightforward since each reference contains the offset and size within the PDS underlying file. We also maintain a small data cache for the PDS data service to speedup the process.

  To read a non-PDS chunk using a reference, the snapshot store client first loads the corresponding VM's container index file specified by the container ID, then searches the chunk group that covers the chunk by the group CID range. After that, it reads the whole chunk group from DFS, decompresses it, and seeks to the exact chunk data specified by the CID. Finally, the client updates its internal chunk data cache with the newly loaded content to anticipate future sequential reads.

- *Delete(reference)*. A data chunk can be deleted when a snapshot expires or gets deleted explicitly by a user.

We will discuss the snapshot deletion in the following subsection. When deletion requests are issued for a specific container, those requests are simply logged into the container's deletion log initially and thus a lazy deletion strategy is exercised. Once CIDs appear in the deletion log, they will not be referenced by any future snapshot and can be safely deleted when needed. Periodically, the snapshot store picks those containers with an excessive number of deletion requests to compact and reclaim the corresponding disk space. During compaction, the snapshot store creates a new container (with the same container ID) to replace the existing one. This is done by sequentially scanning the old container, copying all the chunks that are not found in the deletion log to the new container, and creating new chunk groups and indices. Every chunk's CID however is directly copied rather than re-generated. This process leaves holes in the CID values, but preserves the sorted order. As a result, all data references stored in upper level recipes are permanent and stable, and the data reading process is as efficient as before. This CID stability also ensures that recipes do not depend directly on physical storage locations.

4.3 VM-centric Approximate Snapshot Deletion with Leak Repair

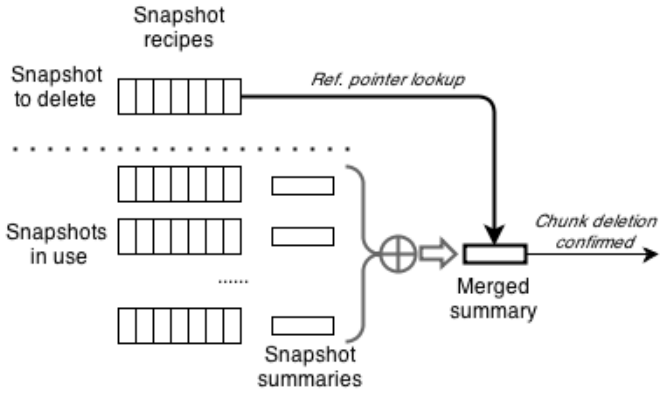


Figure 10: Approximate deletion

In a busy VM cluster, snapshot deletions are as frequent as snapshot creations. However, data deduplication complicates the deletion process because space saving relies on the sharing of data, thus making it difficult to decide which chunks are safe to delete. In a traditional deduplication system, deletion often require looking at global scope to resolve the data dependency between logical backup entities and physical data chunks. Our VM-centric snapshot storage design simplifies the deletion process since we only need to locate unreferenced chunks within each VM's snapshot store to free up space when deleting a snapshot. The PDS data chunks are commonly shared all VMs and we do not consider their reference counting during snapshot deletion.

While we can use the standard mark-and-sweep technique [**?**], it still takes significant time to conduct this process every time there is a snapshot deletion. In the case of Alibaba, snapshot backup is conducted automatically and there are about 10 snapshot stored for every user. When there is a new snapshot created every day, there will be a snapshot expired everyday to maintain a balanced storage use. Given the large number of snapshot deletion requests, we seek a fast solution with a very low resource usage to delete snapshots.

With this in mind, we develop an *approximate* deletion strategy to trade deletion accuracy for speed and resource usage. Our method sacrifices a small percent of storage leakage to effectively identify unused chunks in $O(n)$ time, with n being the logical number of non-PDS chunks to be deleted from a VM snapshot store. The algorithm contains three aspects.

- **Computation for snapshot fingerprint summary.** Every time there is a new snapshot created, we compute a bloom-filter with z bits as the summary of reference pointers for all non-PDS chunks used in this snapshot.

- **Approximate deletion with fast summary comparison.** Then when there is a snapshot deletion, we need to identify if chunks to be deleted from one snapshot are still used by other snapshots. This is done approximately and quickly by comparing the reference pointers of deleted snapshots with the merged reference bloom-filter summary of other live snapshots. The merging of live snapshot bloom-filter bits uses the logical OR operator and the merged vector still takes z bits. Since the number of live snapshots is limited for each VM (e.g. 10 in the Alibaba's production system), the time cost of this comparison is small. However, there is a small false-positive ratio which would identify unused data as in use, resulting in temporary storage leakage.

- **Periodic repair of leakage** Since there are certain unused chunks which are not deleted during the approximate deletion, leakage repair is conducted periodically. In this phase we load the reference pointers of all chunks from a VM snapshot store as a table in memory, then scan through all the snapshots' segment recipes. For each reference pointer that is in use by a snapshot, we mark the corresponding entry in the table as in use. Finally those unmarked (and therefore unused) reference pointers in the table represent chunks that can be safely deleted.

  The cost of leak repair mainly come from holding a table of reference pointers and the scan of all snapshots' metadata. Consider each reference pointer consumes 8 bytes plus 1 byte as mark field, a VM that has 40GB backup data with about 10 million chunks will need 90MB of memory to construct such a table. Also, scanning all snapshots metadata is many

times slower when compared to the approximate deletion which only scans single snapshot's metadata. It's worth mention that compared to previously-developed mark-and-sweep techniques, our leak repair still has advantages because the scope of repair is restricted within a single VM's snapshot store due to our VM-centric design, and we perform leakage repair periodically.

We now estimate the storage leakage and how often leak repair needs to be conducted. Assuming a VM always maintain h snapshots in the backup, and it creates and deletes one snapshot everyday. Let u be the total number of chunks brought by the initial backup, Δu be the average number of additional unique chunks added from one snapshot to the next snapshot. Then the total number of unique chunks used is about:

$$U = u + (h-1)\Delta u.$$

Each bloom filter vector has z bits for each snapshot and let j be the number of hash functions used by the bloom filter. The probablity that a particular bit is still in all h summary vectors is still 0 is $(1-\frac{1}{z})^{jU}$. Notice that when a chunk appears multiple times in these summary vectors, it does not increase the probability of a particular bit being 0 in all h vectors. Then the false-positive rate ε is:

$$\varepsilon = (1-(1-\frac{1}{z})^{jU})^{j}.$$

For each snapshot deletion, the amount of chunks need to be deleted is nearly identical to the number of newly add chunks Δu. However, some of chunks among them are not detected as unused in our approximation algorithm, thus forms the storage leakage. Let R be the total number of runs of approximate deletion since last repair. we estimate the total leakage L after R runs as:

$$L = R\Delta u\varepsilon$$

When leakage ratio L/U exceeds a pre-defined threshold τ, we need to execute a leak repair:

$$\frac{L}{U} = \frac{R\Delta u\varepsilon}{u+(h-1)\Delta u} > \tau \Longrightarrow R > \frac{\tau}{\varepsilon}\frac{u+(h-1)\Delta u}{\Delta u}$$

For example in our tested dataset, each VM keeps $h = 10$ snapshots and each snapshot has about 1-5% of new data. Thus $\frac{\Delta u}{u} = 0.05$ at most. With 40GB snapshot size, $u \approx= 10$ millions. Then $U = 10.45$ millions. We choose $\varepsilon = 0.01$ and $\tau = 0.1$. Then from the bloom filter false positive formula, $z = 10U = 100.45$ million bits. $R = 290$. Then we would expect leak repair be triggered once for every 290 runs of approximate deletion. When one machine hosts 25 VMs and there is one snapshot deletion per day per VM, there would be only one full leak repair for one VM scheduled for every 12 days. Each repairs uses about 90MB memory on average as discussed earlier and takes a short period of time.

5. EVALUATION

We have implemented and evaluated a prototype of our VC scheme on a Linux cluster of machines with 8-core AMD FX-8120 at 3.1 GHz with 16 GB RAM. Our implementation is based on Alibaba's Xen cloud platform [1, **?**]. Each machine is equipped with a a distributed file system (QFS) and overall the cluster manages six 3TB disks with default replication degree set to 3. The PDS replication is set to 5. Objectives of our evaluation are: 1) Study the duplicate detection and removal effectiveness of the VC approach and compare with an alternative VO design. 2) Examine the impacts of VC for fault isolation and tolerance. 3) Evaluate the backup throughput performance VC for a large number of VMs. 4) Assess the resource usage of VC in terms of memory, storage, and network bandwidth.

We will also compare our VC approach with with a VO approach using stateless routing with binning (SRB). SRB executes a distributed deduplication by routing a data chunk to a machine [**?**] using the chunk hashing function discussed in [**?**]. Within such a machine, the data chunk is compared with a local index. This minHash paritioning algorithm helps provide a greater degree of parallelism, but ignores the significant amount of parallelism within a single VM.

5.1 Settings

We have performed a trace-driven study using a 1323 VM dataset collected from 100 Alibaba Aliyun's cloud nodes [**?**]. The production environment tested has about 1000 machines with 25 VMs on each machine. For each VM, the system keeps 10 automatically-backed snapshots in the storage while a user may instruct extra snapshots to be saved. The backup of VM snapshots is completed within a few hours every night. Based on our study of production data, each VM has about 40GB of storage data on average including OS and user data disk. All data are divided into 2 MB fix-sized segments and each segment is divided into variable-sized content chunks [6, 8] with an average size of 4KB. The signature for variable-sized blocks is computed using their SHA-1 hash. Popularity of data blocks are collected through global counting and the top 1-2% will fall into the PDS, as discussed in Section **??**.

Since it's impossible to perform large scale analysis without affecting production VM performance, we sampled two data sets from real user VMs to measure the effectiveness of our deduplication scheme. Dataset1 is used study the detail impact of 3-level deduplication process, it compose of 35 VMs from 7 popular OSes: Debian, Ubuntu, Redhat, CentOS, Win2003 32bit, win2003 64 bit and win2008 64 bit. For each OS, 5 VMs are chosen, and every VM come with 10 full snapshots of it OS and data disk. The overall data size for these 700 full snapshots is 17.6 TB.

Dataset2 contains the first snapshots of 1323 VMs' data disks from a small cluster with 100 nodes. Since inner-VM deduplication is not involved in the first snapshot, this data set helps us to study the PDS deduplication against user-related data. The overall size of dataset2 is 23.5 TB.

5.2 Impact on Fault Tolerance

Figure 11 shows the reliability of VM backups as storage nodes fail for different amounts of sharing of filesystem blocks in the index. We show 6.3 for the VO sharing because that is what we measured in our dataset, and while we expect it to continue to grow (thereby decreasing fault tolerance), it should do so slowly, so we treated VO FSB sharing as a contstant factor of 6.3, as it doesn't change our results. for VC we use 1250, because that is what we estimate at 2500 VMs based on the linear growth up to 105 VMs. We also show 2500 VMs sharing each PDS FSB because that is the absolute worst case for 2500 VMs. Our results show that even the VC worst case provides better VM reliability than our optimistic amount of sharing in VO. The key factor placed is that $N_1 + N_2 < N_o$, caused by the fact that the VM-centric approach localizes deduplication and packs data blocks for one VM as much as possible. The extra replicaton for PDS blocks also signficantly increases the snapshot availability even when a PDS file block is shared by every VM.

Figure 12 shows the advantages of increasing the replication factor for PDS blocks. We use our estimate of 1250 VMs sharing each PDS block, which is what we expect for 2500 VMs, given the growth up to 105 VMs. It is easy to see though thatr increasing the replication of just the PDS blocks (which are the most popular blocks) can have a positive impact on the overall reliablity of the VM backups. Increasing the PDS replication is cheap because is makes up a very small percent of the stored data, but the impact is significant. These figures together show the advantages of the VM-centric model, and the advantages that separating the replication factor for popular blocks can have on reliability.

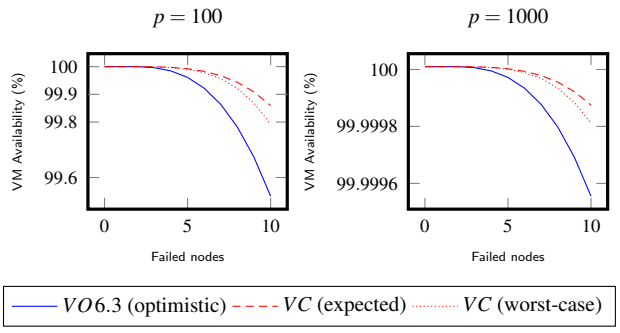


Figure 11: Availability of VM backups for VO and VC models for varying degrees of block sharing. Non-PDS replication fixed at 3 and PDS replication increased to 4. VC worst-case is $V = 25 * p$, and expected-case is $\frac{V}{2}$

5.3 Deduplication Efficiency

Figure **??** shows the deduplication efficiency for SRB and VC. We define deduplication efficiency to be the percent of duplicate chunks which are detected and deduplicated. The figure also compares several PDS sizes chosen for VC. "$\sigma = 2\%$" means that we allocate the space of distributed

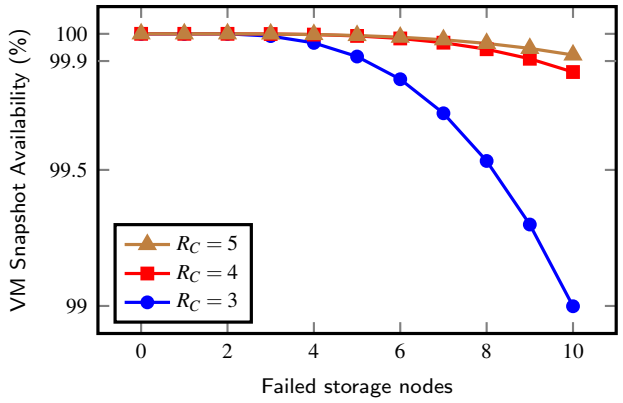


Figure 12: Availability of VM backups as nodes fail in the VC model for different PDS Replication factors (Non-PDS replication fixed at 3, and average PDS block links set to 1000)

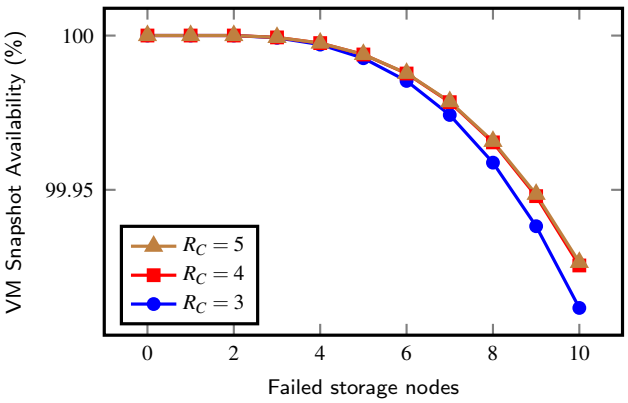


Figure 13: Availability of VM backups as nodes fail in the VC model for different PDS Replication factors (Non-PDS replication fixed at 3, and average PDS block links set to 20)

PDS replication degree	Total space (GB)
3	3065
4	3072
5	3079
6	3086
7	3093

Table 2: Storage space cost under different PDS replication degree

shared memory to accommodate 2% of data chunks shared among VMs, as defined in Table 1. Namely With $\sigma = 2\%$, the deduplication efficiency can reach over 90% while with $\sigma = 4\%$, the deduplication efficiency can reach 92%. The loss of efficiency in VC is caused by the restriction of the total physical memory available in the cluster to support the PDS. The loss of efficiency in SRB is caused by the routing of data chunks which restricts the search scope for global comparison. In all cases, VC provides similar or better deduplication efficiency than SRB.

This result shows VC can remove a competitive amount of duplicates. In general, our experiments show that dirty-bit detection at the segment level can reduce the data size to about 23% of original data, which leads about 77% reduction. Local search within a segment can further reduce the data size to about 18.5% of original size, namely it delivers additional 4.5% reduction. The PDS-based cross-VM detection with $\sigma = 2\%$ can reduce the size further to 8% of original size, namely it delivers additional 10.5% reduction.

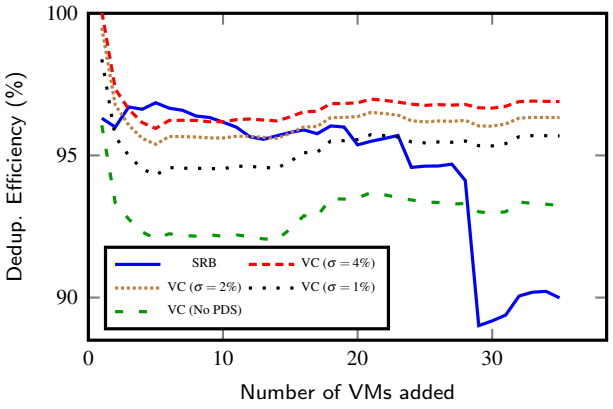


Figure 14: Deduplication Efficiency ($efficiency = \frac{d-du}{d-du_{complete}}$) **comparison between Stateless Routing with Binning and VC. VC uses the minhash algorithm for Level-2.**

5.4 Resource Usage

Storage cost of replication.

Table 2 shows the total storage space required by VC as the PDS replication degree is changed. The increase in storage cost is minimal because the PDS makes up a very small

Tasks	CPU	Mem (MB)	Disk W (MB/s)	Network
1	19%	18.1	32.1	
2	35%	31.8	53	
4	63%	54.1	84	
6	77%	71.9	88.5	
8	82%	90.5	89.2	
10	85%	97.2	90.4	
12	91%	95.6	91.5	

Table 3: Resource usage under concurrent backup tasks

percent of the total storage space, but the increase in reliability can be much more significant (see Figure 12).

5.5 Processing Time and Throughput

Figure 15 shows the average processing time of each segment using VC and SRM in handling our dataset. It has a breakdown of time for reading data and updating the metadata, network latency to visit a remote machine, and index access for finger printer comparison. SRB has a higher index access and fingerprint comparison because once a chunk is routed to a machine, it relies on this local machine to access its index (often on the disk) and perform comparison. VC is faster for index access because it conducts in-memory local search first and then accesses the PDS which is on distributed shared memory. SRB spends slighter more time in reading data and updates because it also updates the on-disk local meta data index. Overall, VC is faster than SRB, though data reading dominates the processing time for both algorithms.

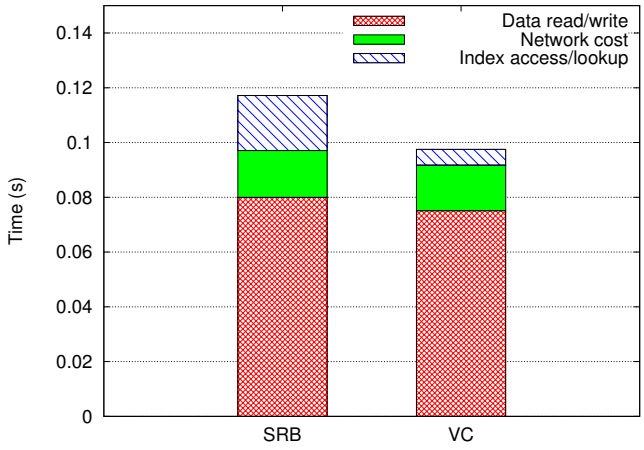


Figure 15: Time to backup a dirty segment under SRB and VC approach

Figure 16 reports the average bacup time for a VM in VC with a varying PDS size. Again it shows that data reading dominates the backup process; our deduplication procedure only takes a small fraction of the total backup time. The change of σ does not significantly affect the overall backup speed as PDS lookup takes only a small amount of time.

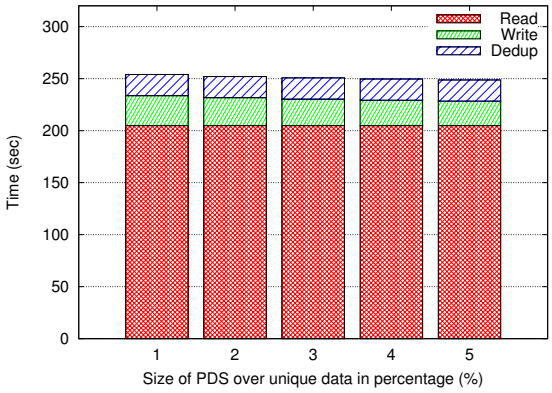


Figure 16: Backup times for varying PDS sizes

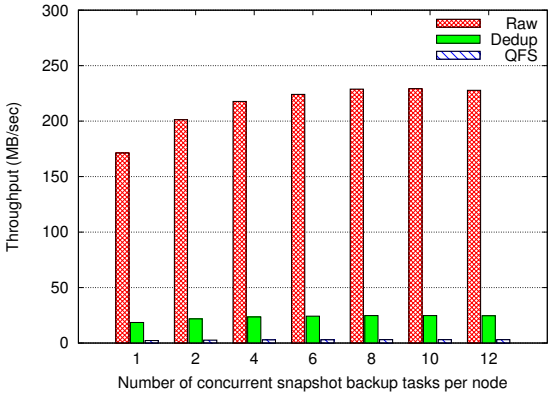
Figure 17 shows the node throughput when all machine nodes run the backup procedure in parallel. To begin, on each node we write snapshots for 4 VMs concurrently, and gradually increase number of VMs to 12 to saturate our system capability. We observed the per-node throughput peaked at 2700 MB/s when writing 10 VM snapshots in parallel, which is far beyond our QFS file system capability. The reason behind it is our efficient deduplication architecture and compression which greatly reduce the amount of data that needs to be written to the file system. The main bottleneck here is that our QFS installation only manages one disk per node, which prevents it from fully utilizing the the benefits of parallel disk access. We expect our architecture can perform even better in production clusters, which often have ten or more disks on each node.

5.6 Approximate Deletion

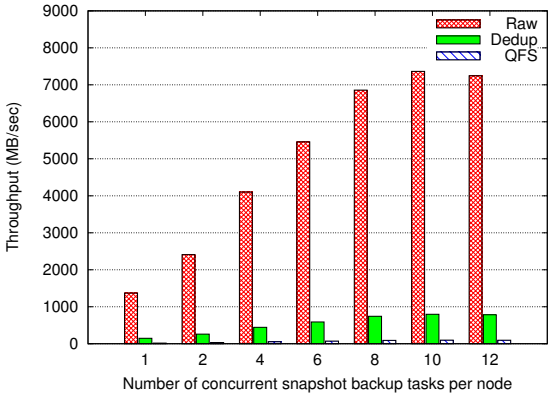
Figure18 shows the average accumulated storage leakage resulted by daily approximate deletions among all VMs. We see the storage leakage ratio is satisfactory: after 9 snapshot approximate deletions, there is only 1MB space leaked over every 10GB of stored data. In addition, the leakage ratio grows linearly with the number of approximate deletions as our analysis, which gives us a reliable prediction of when to trigger a accurate mark-and-sweep delete operation.

6. CONCLUSION

In this paper we propose a VM-centric deduplication scheme for VM snapshot backup in the cloud for maximizing fault isolation and tolerance. Inner-VM deduplication localizes backup data dependency and exposes more parallelism while cross-VM deduplication with a small common data set effectively covers a large percent of duplicate data. The scheme organizes the write of small data chunks into large file system blocks so that each underlying file block is associated with one VM for most cases. by keeping most of the FSBs only referenced by 1 VM, we isolate faults and improve overal fault tolerance. Our solution accomplishes reasonable and competitive deduplication efficiency while still meeting



(a) Real VM backup performance



(b) Deduplication and storage system performance

Figure 17: Throughput per-node with concurrent snapshot backup tasks

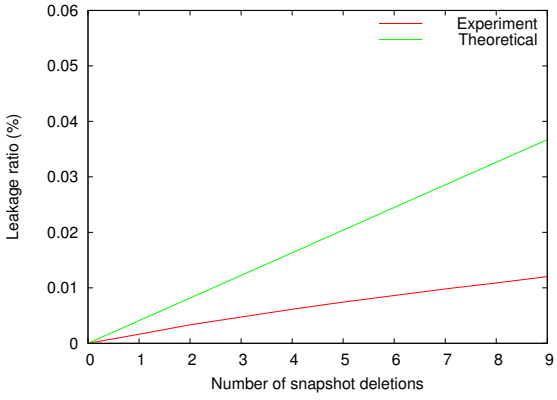


Figure 18: Accumulated storage leakage by approximate snapshot deletions

stringent cloud resource usage requirements. Our analysis shows that this VM centric scheme can provide better fault tolerance than VM-oblivious global deduplication schemes while using a small amount of computing and storage resources.

[Talk about more what we learn from Evaluation] Evaluation using real user's VM data shows our solution can accomplish 92% of what complete global deduplication can do. Compare to today's widely-used snapshot technique, our scheme reduces almost two-third of snapshot storage cost. Finally, our scheme uses a very small amount of memory on each node, and leaves room for additional optimization we are further studying.

7. REFERENCES

[1] Alibaba Aliyun. http://www.aliyun.com.

[2] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE MASCOTS '09*, pages 1–9, 2009.

[3] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *USENIX ATC'09*, 2009.

[4] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *USENIX ATC'11*, pages 25–25, 2011.

[5] 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'09*, pages 111–123, 2009.

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

[7] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *FAST '02*, pages 89–101, 2002.

[8] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981.

[9] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *FAST'12*, 2012.

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

[11] J. Wei, H. Jiang, K. Zhou, and D. Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *IEEE MSST'10*, pages 1–14, May 2010.

[12] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08*, pages 1–14, 2008.