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. 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 deduplication using signatures and let a data block share by many virtual machines. This paper studies a VM-centric scheme which collocates a lightweight backup service with other cloud services in a cluster and it integrates multiple duplicate detection strategies that localize deduplication as much as possible within each virtual machine. It also 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 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 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 [**?**, **?**] 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.

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 amount 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 parallelism exploitation so backup operations can run simultaneously without a central bottleneck. This VM-centric solution uses a small amount of memory while delivering a decent deduplication efficiency. 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.

The rest of this paper is organized as follows. Section **??** reviews background and related work. Section **??** discusses the design options for snapshot backup with a VM-centric approach. Section **??** analyzes the benefit of our approach for fault isolation. Section **??** 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 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 that can eliminate redundancy globally among different files from different users. Backup systems have been developed to use content fingerprints to identify duplicate content [**?**, **?**]. Offline deduplication is used in [**?**, **?**] to remove previously written duplicates during idle time. Several techniques have been proposed to speedup searching of duplicate fingerprints. For example, the data domain method [**?**] 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 [**?**, **?**]. The approximation techniques are studied in [**?**, **?**, **?**] to reduce memory requirements with the tradeoff of a reduced deduplication ratio.

Additional inline deduplication techniques are studied in [**?**, **?**, **?**]. 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. Figure **??** illustrates the cluster architeture where each physical runs a backup service and a distribued file system (DFS) [**?**, **?**] serves a backup store for the snapshopts. 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.

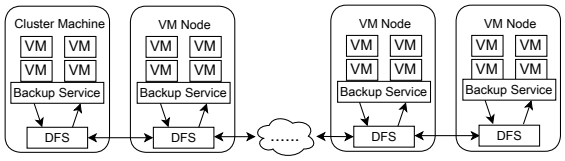


Figure 1: Collocated VM Backup System.

We discuss the design considerations as follows.

- *Deduplication localization, sharing minmization, and fault tolerance.*

  Because a data chunk is compared with signatures collected from other VMs during the deduplication process, only one copy of duplicates is stored in the backup storage and this artificially creates data dependency among different VM users. Content sharing via deduplication affects fault isolation since machine failures happen at daily basis 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 incrase fault isolation and resilience. Thus from the fault tolerance point of view, duplicate sharing among mutiple VMs is discourage. On the other hand, we need to seek for a tradeoff since there are a significant number of duplicates cross VMs.

- **Packaging data chunks as file system blocks.** Since the file block size in the Hadoop and GFS is uniform and large with 64MB as a default setting, the content chunk 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. Packaging that maps data chunks to file system blocks can create data dependence among VMs since a file block can be shared even more VMs. Thus we need to consider a minimum assoocation of a fie system block to VMs in the packaging process.

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 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 [**?**, **?**] along with the fault isolation consideration discussed below.

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 chunk's duplicates. We call this as the VM-oblivious (VO) approach. In comparison, we analyze the design of a VM-centric approach (VC) which differentiates duplicates within a VM and cross VMs, and conducts *Inner-VM* and *cross-VM* detection separately.

4.1 Inner and cross VM deduplication

Inner-VM duplication can be very effective 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 within a VM improves the

system by: increasing data independence 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 coarse-grain inner-VM deduplication.** The first-level deduplication is to follow the standard dirty bit approach, but is conducted in the coarse grain segment level. We use the Xen virtual device driver which supports a feature called "changed block tracking" for the storage device and the dirty bit setting is maintained in a coarse grain level we call it a segment. In our implementation, the segment size is 2MB. 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 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 chunks. 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 chunk fingerprints in 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 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 most popular items.

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 chunk. We manage this accumulation process using an append-store scheme and discuss this in details in Section **??**. 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.

It should be noted that CDS chunks are computed periodically and all CDS chunks are stored in a separate append-store instance. In this way, each file block for non-CDS chunks is associated with one VM and does not contain any CDS chunk.

4.2 Popular Chunk Management

With cross-VM deduplication, shared data chunks create an artificial dependence among VMs. Thus we only maintain the index for the most popular chunks to reduce the inter-VM dependence while still accomplishing competitive deduplication efficiency.

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 chunks from different VMs that are duplicates of this chunk after the inner VM deduplication. This number can be computed periodically in a weekly basis. Once the popularity of all data chunks is collected, the system only maintains the top k most popular chunks. 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 efficiency after making these approximations.

- Since k is 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. We will provide an analysis of the space need and 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 chunks after inner VM deduplication follows a Zipf-like distribution[**?**] and its exponent α is ranged between 0.65 and 0.70.

[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.] Let b be the total number of data chunks after localized deduplication, b_u

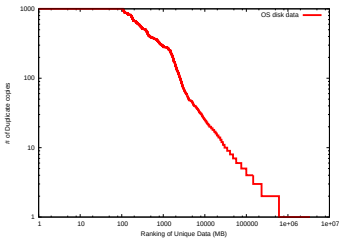


Figure 2: Duplicate frequency versus chunk ranking in a log scale.

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:

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

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

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

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

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

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 physical machine, and B be the average chunks size. We store the popular index using a distributed 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 efficiency of VC 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 chunks 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 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.

b	the total amount of data chunks
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
σ	the number of unique non-CDS chunks over the number of the CDS chunks.
p	the number of machines in the cluster
D	the amount of unique data on each machine
B	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
F	the size of an popular data index entry
N_1	the average number of non-CDS file system blocks in a VM
N_2	the average number of CDS file system blocks in a VM
N_o	the average number of file system blocks in a VM for VO

Table 1: Modeling symbols.

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

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

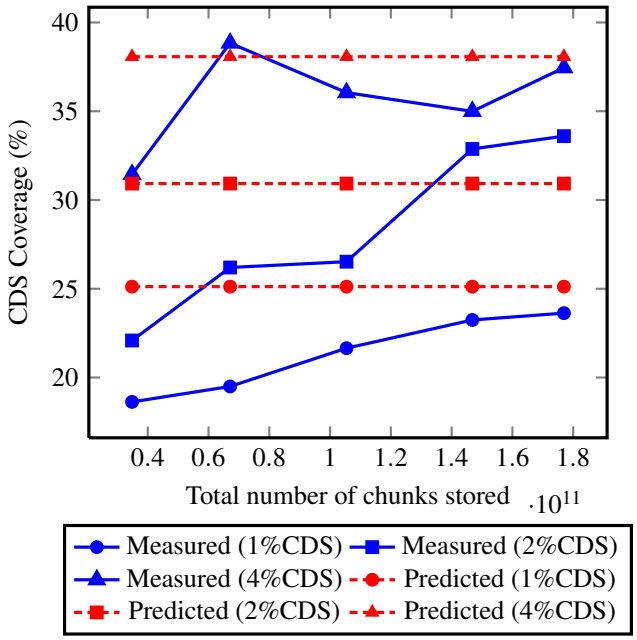
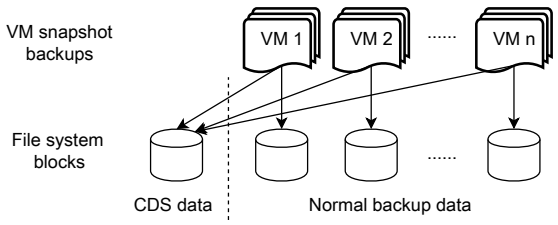
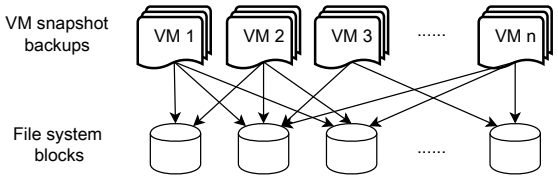


Figure 3: fixed-alpha predicted vs. actual CDS coverage as data size increases.



(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

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 CDS blocks where $r_c > r$. Notice that the ratio of non CSD data size vs CDS data size for each VM is

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

Thus storage cost for VO with full deduplication is $b_u * r$ 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 CDS 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 CDS is insignificant. Now we compare the fault resilience of VC and VO.

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 node 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 (in our case there are 16384 blocks in a filesytem block on average, with 4KB block size and a 64MB block size).

To compute the probablity of losing 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, let N_1 be the averge number of file system blocks for non-CDS data for each VM and N_2 be the averge number of file system blocks for CDS data for each VM. For VO, let N_o be the averge number of file system blocks for each VM and let V_o be the averge number of VMs shared by each file system block. Figure **??** illustates the bipartite association.

Each non-CDS file system block is associated with one VM while CDS file system blocks are shared among VMs.

$$V * N_1 * s = pD\frac{\delta}{\delta + 1}$$

For CDS file blocks, there are at most V VMs sharing, thus

$$V * N_2 * s \leq pD\frac{1}{\delta + 1} * V$$

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 **??** shows the number of VMs shared by each file block. In our experiment, we find that $V_o \approx 0.2V$ when backuping up VMs one by one. We can observe that $N_1 + N_2 << N_o$. If the backup for multiple VMs is conducted concurrently, there would be more VMs sharing for each file block on average.

Figure **??** shows the average number of VMs shared by a block in the global index as more VMs are added. The first 15 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.

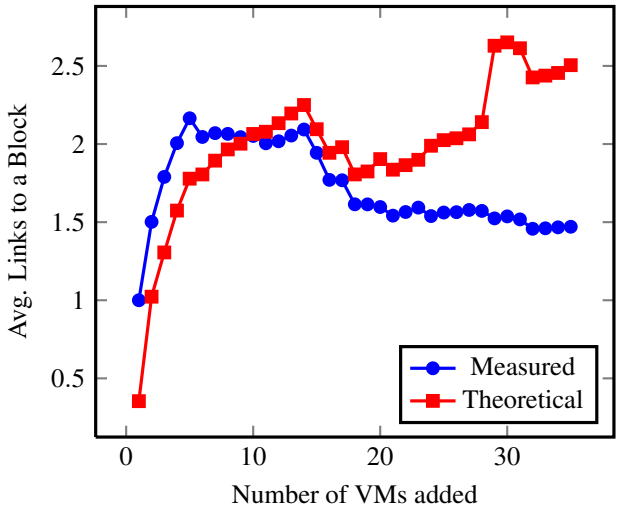


Figure 5: Average number of VMs sharing a block in the global index

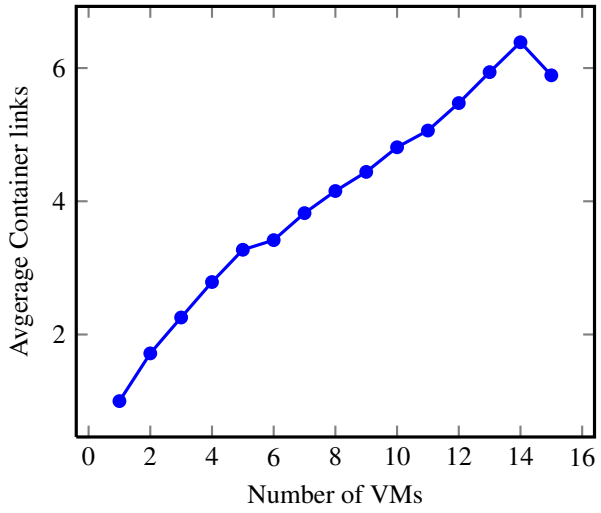


Figure 6: Measured Average number of VMs sharing a container in the global index (for VO)

The availability of is a VM is the likelyhood that 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, $\binom{d}{r}/\binom{p}{r}$.

When there are $r \leq d < r_c$ machines failed, the avaiability of a VM in the VC approach is

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

When $r_c \leq d$, the CDS file block in VC can also have a loss. The 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 availability of its VM mbox in the VO approach is

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

The figure shows that even as the number of VMs sharing a CDS file block increase by 100 times, the availability only decreases slightly. The key factor placed is that $N_1+N_2 << N_o$. The VM-centric approach packs data blocks under one VM as much as possible and performs deduplication mainly within the same VM. The sharing part in CDS is replicated with a higher degree, which also signficantly increses the availability even a CDS file blcok is shared by every VM.

Figure **??** shows the advantages of increasing the replication factor for CDS blocks. 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.

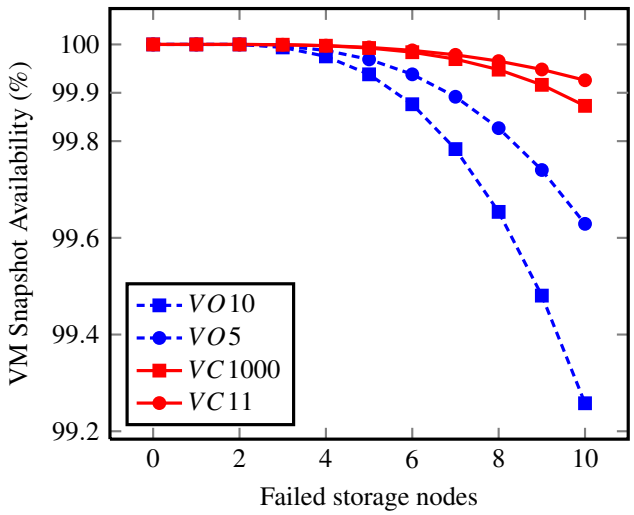


Figure 7: Availability of VM backups as nodes fail for VO and VC models for varying average block link counts (i.e. average number of VMs that use a block)

5. A COMPARISON WITH OTHER APPROACHES

Sampled Index. One alternative approach to avoiding a single global index is to use a sampled index. This is the approach taken by Guo and Efstathopoulos[**?**]. We compare their solution to ours by running their algorithm on our 350 traces using a sampling rate of 1 out of 101 blocks, and always including the first block in a container (which we fix at 16MB in size). The results of running this test are shown in Table **??**, and the memory usage comparison can be found Sec. **??**. Our results show that using a sampled

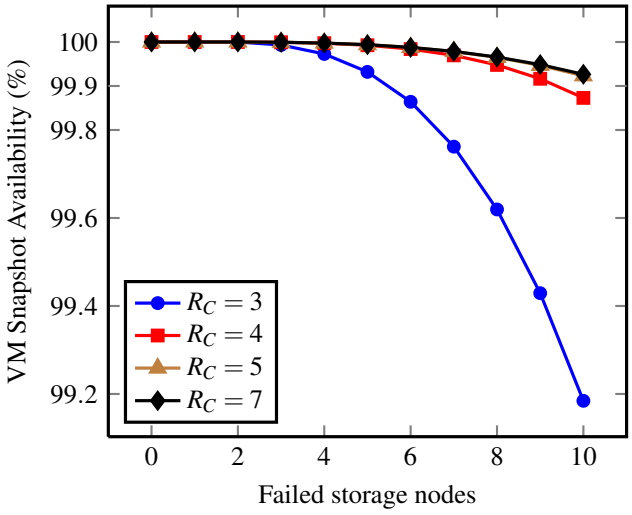


Figure 8: Availability of VM backups as nodes fail in the VC model for different CDS Replication factors (Non-CDS replication fixed at 3, and average CDS block links set to 1000)

index achieves a very high rate of deduplication, and so a sampled index might be a good way to do dedup in a single node setup.

The problems for sampling are in distributing the algorithm and in deduplicating extremely large bodies of data. The algorithm stores the entire (sampled) index in memory, which is required for high throughput to avoid the disk-bottleneck problem - since the index itself has no locality information, lookups are essentially random, so the index must be somehow optimized to prevent excessive disk seeking (the sampling algorithm does this by storing the index in memory). To distribute this index, each node would either require a complete copy of the index which would have a prohibitive cost in memory, or something like memcached must be used, which leaves the same problem as the disk bottleneck with every index check potentially using the network. The difficulty in storing extremely large bodies of data is related, as once 1PB is stored in the system, assuming a sampling rate of 1/101 with 22 byte index entires, 55GB of (in memory) index are required.

Our solution uses more total memory (??how much??), but is more scalable both in terms of capactity and distributing the deduplication. The only part of our algorithm which requires significant network traffic is the CDS deduplication, but this is done after Levels 1 and 2 (dirty bits and comparison to parent), and so is only done for a small percent of blocks (?? 5-10% ??).

Scalable Data Routing. Another approach to avoiding a global index is to use a content-based hash partitioning algorithm to break up the deduplication work. This approach is taken by Dong et al. in their Scalable Data Routing paper, and is similar to Exreme Binning[**?**][**?**]. We compared our solution to content-based hash partitioning by running the algorithm on our set of traces. We used 2MB superchunks and 4KB chunks to partition the data, and use the minhashes of the superchunks for routing. The results of this are shown in Table **??**. Routing each chunk to a bin provides good deduplication efficiency, and only requires each storage node to keep an index of local data and the bin mapping, but misses significant opportunities in our indended use case.

The problem with the Data Routing algorithm is in the very high network traffic. Our intended use case is a backup system which runs alongside a number of VMs, in order to save costs. Data Routing makes no use of the inherent locality in such a system, and therefore puts a much higher network burden on machines which are also running 35 VMs each. This will reduce the available network bandwidth to users of the VMs, and/or reduce the possible deduplication throughput.

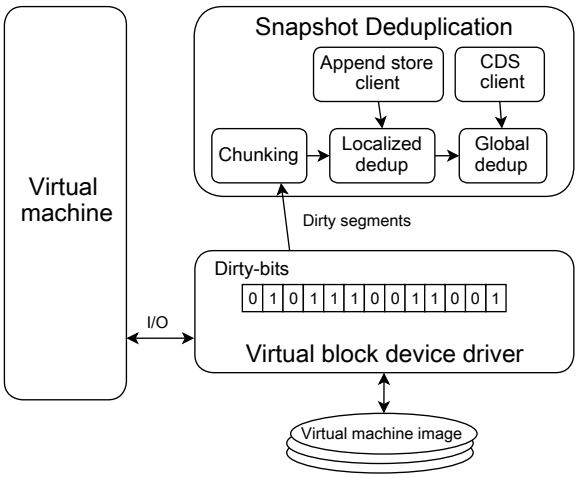
6. ARCHITECTURE AND IMPLEMENTATION DETAILS

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[**?**] in Xen).

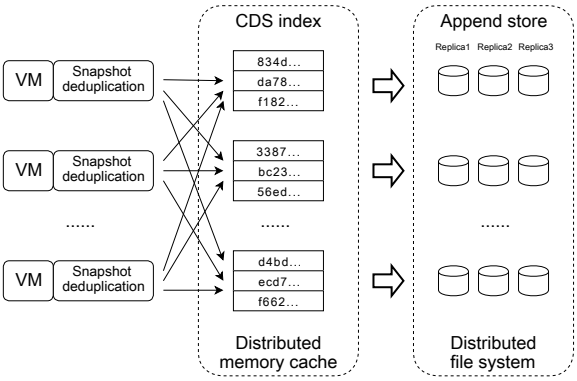
6.1 Components of a cluster node

As depicted in Figure **??**, 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.



(a) Node architecture from VM point of view



(b) Cluster architecture

Figure 9: System architecture.

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.

6.2 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.**??**, every data container is represented

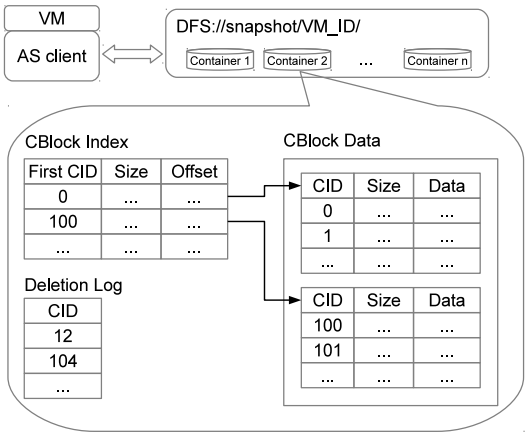


Figure 10: Architecture of Append Store

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.

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.

6.3 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.

7. EVALUATION

We have implemented and evaluated a prototype of our VO scheme on a Linux cluster of 8-core AMD FX-8120 at 3.1 GHz with 16 GB RAM. Our implementation is based on Alibaba's Xen cloud platform [**?**, **?**]. Each machine is equiped with a a distributed file system (QFS) running in the cluster manages six 3TB disks with default replication degree set to 3. The CDS replication is set to 5. Objectives of our evaluation are: 1) Study the deduplication efficiency of the VC approach and compare with an alternative design. 2) Evaluate the backup throughput performance VC for a large number of VMs. 3) Examine the impacts of VC for fault isolation.

7.1 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 its production data, each VM has about 40GB of storage data usage 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 [**?**, **?**] 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% will fall into CDS, as discussed in Section **??**.

Since it's impossible to perform large scale analysis without affecting the 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 this 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 CDS deduplication against user-related data. The overall size of dataset2 is 23.5 TB.

7.2 Deduplication Efficiency

Figure **??** shows the overall impact of 3-level deduplication on dataset1. The X axis shows the overall impact in (a), impact on OS disks in (b), and impact on data disks in (c). Each bar in the Y axis shows the data size after deduplication divided by the original data size. Level-1 elimination can reduce the data size to about 23% of original data, namely it delivers close 77% reduction. Level-2 elimination is applied to data that could pass level-1, it reduces the size further to about 18.5% of original size, namely it delivers additional 4.5% reduction. Level-3 elimination together with level 1 and 2 reduces the size further to 8% of original size, namely it delivers additional 10.5% reduction. Level 2 elimination is more visible in OS disk than data disk, because data change frequency is really small when we sample last 10 snapshots of each user in 10 days. Nevertheless, the overall impact of level 2 is still significant. A 4.5% of reduction from the original data represents about 450TB space saving for a 1000-node cluster.

Figure **??** shows the impact of different levels of deduplication for different OS releases. In this experiment, we tag each block in 350 OS disk snapshots from dataset1 as "new" if this block cannot be deduplicated by our scheme and thus has to be written to the snapshot store; "CDS" if this block can be found in CDS; "Parent segment" if this block is marked unchanged in parent's segment recipe. "Parent block" if this block is marked unchanged in parent's block recipe. With this tagging, we compute the percentage of deduplication accomplished by each level. As we can see from Figure **??**, level-1 deduplication accomplishes a large percentage of elimination, this is because the time interval between two snapshots in our dataset is quite short and the Aliyun cloud service makes a snapshot everyday for each VM. On the other hand, CDS still finds lots of duplicates that inner VM deduplication can't find, contributing about 10% of reduction on average.

It is noticeable that level-1 deduplication doesn't work well for CentOS, a significant percentage of data is not eliminated until they reach level-3. It shows that even user upgrade his VM system heavily and frequently such that data locality is totally lost, those OS-related data can still be identified at level-3.

In general we see a stable data reduction ratio for all OS varieties, ranging from 92% to 97%, that means the storage cost of 10 full snapshots combined is still smaller than the original disk size. And compare to today's widely used copy-on-write snapshot technique, which is similar to our level-1 deduplication, our solution cut the snapshot storage cost by 64%.

8. EXPERIMENTS

Our main test-bed is an cluster of 6 machines, each of which is equiped with a 8-core AMD FX-8120 at 3.1 GHz with 16 GB RAM, running Linux. A distributed file system (QFS) runs in the cluster manages six 3TB disks with default replication degree set to 3. Our data set consists of 350 virtual machine image snapshots taken from Alibaba.com's public VM cloud in China. We select 35 VMs from the most popular 7 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.

8.1 System Performance

Single VM Backup We start the evauluation of our system by examine the normal warmed-up backup scenario - taking a snapshot of a VM which already has old backups exist. To simulate this scenario, we pick one 40GB VM from the data set, make an initial snapshot backup, then evaulate the system performance on the second snapshot backup. We repeat such procedure under different CDS memory usage settings, the results of backup time break down are shown in firure **??**.

It's easy to see that the time of I/O dominates the backup process, our deduplication procedure only takes a tiny fraction of the total backup time. Larger memory permits us to use larger CDS index to cover more of global index, as a result the amount of data to be writen is reduced and so do the writing time. Inside the deduplication procedure, the time costs of level-1 is plainly zero, and the costs of level-2 and level-3 are almost identical under different settings because the number of fingerprints to process are just the same.

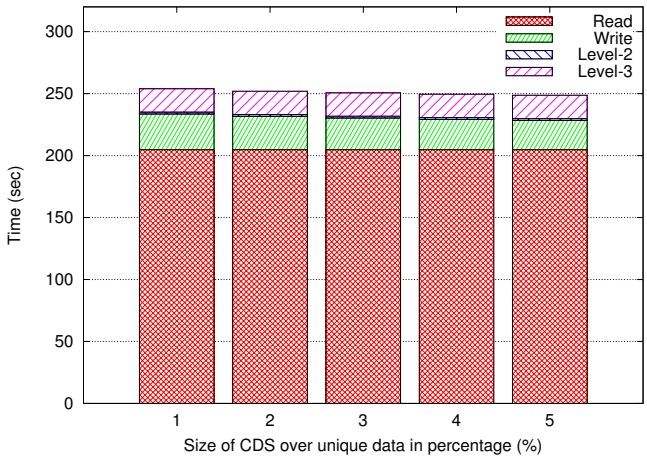
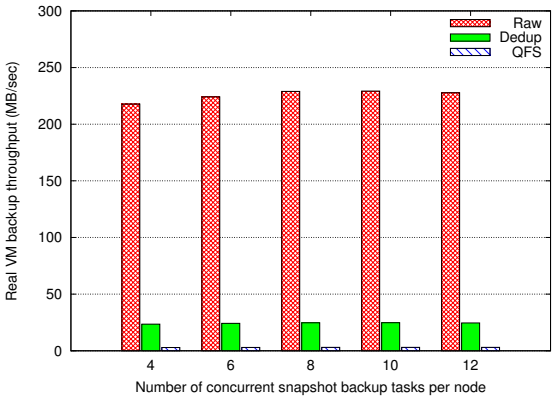
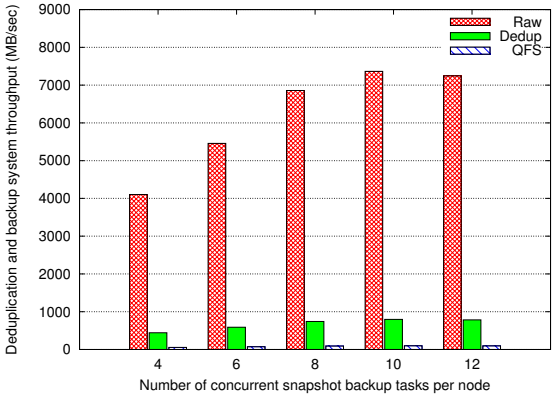


Figure 11: Time to backup a 40GB VM snapshot under different CDS memory usage settings

Parallel VM Backup We then evaluate the performance of writing snapshots in parallel. To begin, on each node we let 4 VMs writing snapshot concurrently, and gradually increase number of VMs to 12 to saturate our system capability. We observed the per-node throughput peaked at 2700



(a) Real VM backup performance



(b) Deduplication and storage system performance

Figure 12: Throughput per-node with concurrent snapshot backup tasks

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 greatly reduce the amount of data that need to be writen to the file system. The main bottleneck here is our QFS only manages one disk per node, making it inefficient to utilize the benefits of parallel disk access. We expect our achitecture can perform even better in production cluster which normally has more than ten disks on each node.

8.2 Read Snapshot

8.3 Comparison with Other Approaches

8.3.1 Sampled Index

One alternative approach to avoiding a single global index is to use a sampled index. This is the approach taken by Guo and Efstathopoulos[**?**]. We compare their solution to ours by running their algorithm on our 350 traces using a sampling rate of 1 out of 101 blocks, and always including the first block in a container (which we fix at 16MB in size). The results of running this test are shown in Table **??**, and the memory usage comparison can be found Sec. **??**. Our results show that using a sampled index achieves a very high rate of deduplication, and so a sampled index might be a good way to do dedup in a single node setup.

The problems for sampling are in distributing the algorithm and in deduplicating extremely large bodies of data. The algorithm stores the entire (sampled) index in memory, which is required for high throughput to avoid the disk-bottleneck problem - since the index itself has no locality information, lookups are essentially random, so the index must be somehow optimized to prevent excessive disk seeking (the sampling algorithm does this by storing the index in memory). To distribute this index, each node would either require a complete copy of the index which would have a prohibitive cost in memory, or something like memcached must be used, which leaves the same problem as the disk bottleneck with every index check potentially using the network. The difficulty in storing extremely large bodies of data is related, as once 1PB is stored in the system, assuming a sampling rate of 1/101 with 22 byte index entires, 55GB of (in memory) index are required.

Our solution uses more total memory (??how much??), but is more scalable both in terms of capactity and distributing the deduplication. The only part of our algorithm which requires significant network traffic is the CDS deduplication, but this is done after Levels 1 and 2 (dirty bits and comparison to parent), and so is only done for a small percent of blocks (?? 5-10% ??).

8.3.2 Scalable Data Routing

Another approach to avoiding a global index is to use a content-based hash partitioning algorithm to break up the deduplication work. This approach is taken by Dong et al. in their Scalable Data Routing paper, and is similar to Exreme Binning[**?**][**?**]. We compared our solution to content-based hash partitioning by running the algorithm on our set of traces. We used 2MB superchunks and 4KB chunks to partition the data, and use the minhashes of the superchunks for routing. The results of this are shown in Table **??**. Routing each chunk to a bin provides good deduplication efficiency, and only requires each storage node to keep an index of local data and the bin mapping, but misses significant opportunities in our indended use case.

The problem with the Data Routing algorithm is in the very high network traffic. Our intended use case is a backup system which runs alongside a number of VMs, in order to save costs. Data Routing makes no use of the inherent locality in such a system, and therefore puts a much higher network burden on machines which are also running 35 VMs each. This will reduce the available network bandwidth to users of the VMs, and/or reduce the possible deduplication throughput.

9. CONCLUSION

In this paper we propose a VM-centric deduplication scheme

for snapshot backup in VM 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 amount of duplicated 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 of cases. Our solution accomplishes the majority of potential global deduplication saving while still meets stringent cloud resources requirement. Our analysis shows that this VM centric scheme can provide better fault tolerance while using a small amount of computing and storage resource.

[Talk about more what we learn from Evaluation] Evaluation using real user's VM data shows our solution can accomplish 75% 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.