Low-Cost Data Deduplication for Virtual Machine Backup in Cloud Storage

Wei Zhang*, Tao Yang*, Gautham Narayanasamy*, and Hong Tang†
* University of California at Santa Barbara, †Alibaba Inc.

Abstract

In a virtualized cloud cluster, frequent snapshot backup of virtual disks improves hosting reliability; however, it takes significant memory resource to detect and remove duplicated content blocks among snapshots. This paper presents a low-cost deduplication solution scalable for a large number of virtual machines. The key idea is to separate duplicate detection from the actual storage backup instead of using inline deduplication, and partition global index and detection requests among machines using fingerprint values. Then each machine conducts duplicate detection partition by partition independently with minimal memory usage. Another optimization is to allocate and control buffer space for exchanging detection requests and duplicate summaries among machines. Our evaluation shows that the proposed multi-stage scheme uses a small amount of memory while delivering a satisfactory backup throughput.

1 Introduction

Periodic archiving of virtual machine (VM) snapshots is important for long-term data retention and fault recovery. For example, daily backup of VM images is conducted automatically at Alibaba which provides the largest public cloud service in China. The cost of frequent backup of VM snapshots is high because of the huge storage demand. This issue has been addressed by storage data deduplication [9, 17] that identifies redundant content duplicates among snapshots. One architectural approach is to attach a separate backup system with deduplication support to the cloud cluster, and every machine periodically transfers snapshots to the attached backup system. Such a dedicated backup configuration can be expensive, considering that significant networking and computing resource is required to transfer raw data and conduct signature comparison.

This paper seeks for a low-cost architecture option and considers that a backup service uses the existing cloud computing resource. Performing deduplication adds significant memory cost for comparison of content fingerprints. Since each physical machine in a cluster hosts many VMs, memory contention happens frequently. Cloud providers often wish that the backup service only consumes small or modest resources with a minimal impact to the existing cloud services. Another challenge is that deletion of old snapshots compete for computing resource as well, because data dependence created by duplicate relationship among snapshots adds processing complexity.

Among the three factors - time, cost and deduplication efficiency, one of them has to be compromised for the other two. For instance, if we were building a deduplication system that has a high rate of duplication detection and has a very fast response time, it would need a lot of memory to hold fingerprint index and cache. This leads to a compromise on cost. Our objective is to lower the cost incurred while sustaining the highest de-duplication ratio and a sufficient throughput in dealing with a large number of VM images.

The traditional approach to deduplication is an inline approach which follows a sequence of block reading, duplicate detection, and non-duplicate block write to the backup storage. Our key idea is to first perform parallel duplicate detection for VM content blocks among all machines before performing actual data backup. Each machine accumulates detection requests and then performs detection partition by partition with minimal resource usage. Fingerprint based partitioning allows highly parallel duplicate detection and also simplifies reference counting management. The tradeoff is that every machine has to read dirty segments twice and that some deduplication requests are delayed for staged parallel processing. With careful parallelism and buffer management, this multi-stage detection scheme can provide a sufficient throughput for VM backup.

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.

Backup systems have been developed to use content fingerprints to identify duplicate content [9, 10]. Offline deduplication is used in [6, 2] to remove previously written duplicate blocks during idle time. Several techniques have been proposed to speedup searching of duplicate fingerprints. For example, the data domain method [17] uses an in-memory Bloom filter and

a prefetching cache for data blocks which may be accessed. An improvement to this work with parallelization is in [14, 15]. As discussed in Section 1, there is no dedicated resource for deduplication in our targeted setting and low memory usage is required so that the resource impact to other cloud services is minimized. The approximation techniques are studied in [4, 7, 16] to reduce memory requirement with a tradeoff of the reduced deduplication ratio. In comparison, this paper focuses on full deduplication without approximation.

Additional inline deduplication techniques are studied in [8, 7, 12]. All of the above approaches have focused on such inline duplicate detection in which deduplication of an individual block is on the critical write path. In our work, this constraint is relaxed and there is a waiting time for many duplicate detection requests. This relaxation is acceptable because in our context, finishing the backup of required VM images within a reasonable time window is more important than optimizing individual VM block backup requests.

3 System Design

We consider deduplication in two levels. The first level uses coarse-grain segment dirty bits for version-based detection [5, 13]. Our experiment with Alibaba's production dataset shows that over 70 percentage of duplicates can be detected using segment dirty bits when the segment size is 2M bytes. This setting requires OS to maintain segment dirty bits and the amount of space for this purpose is negligible. In the second level of deduplication, content blocks of dirty segments are compared with the fingerprints of unique blocks from the previous snapshots. Our key strategies are explained as follows.

- **Separation of duplicate detection and data backup.** The second level detection requires a global comparison of fingerprints. Our approach is to perform duplicate detection first before actual data backup. That requires a prescanning of dirty VM segments, which does incur an extra round of VM reading. During VM prescanning, detection requests are accumulated. Aggregated deduplicate requests can be processed partition by partition. Since each partition corresponds to a small portion of global index, memory cost to process detection requests within a partition is small.
- **Buffered data redistribution in parallel duplicate detection**. Let *global index* be the meta data containing the fingerprint values of unique snapshot blocks in all VMs and the reference pointers to the location of raw data. A logical way to distribute detection requests among machines is based on fingerprint values of content blocks. Initial data blocks follows the VM distribution among machines and the detected duplicate summary should be collected following the same distribution. Therefore, there are two all-to-all data redistribution operations involved. One is to map detection requests from VM-based distribution to fingerprint based distribution. Another one is to map duplicate summary from fingerprint-based distribution to VM based distribution. The redistributed data needs to be accumulated on the disk to reduce the use of memory. To minimize the disk seek cost, outgoing or incoming data exchange messages are buffered to bundle small messages. Given there are $p \times q$ partitions where p is the number of machines and q is the number of fingerprint-based partitions at each machine, space per each buffer is small under the memory constraint for large p or q values. This counteracts the effort of seek cost reduction. We have designed an efficient data exchange and disk data buffering scheme to address this.

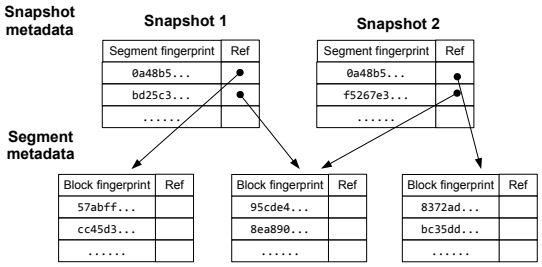

Figure 1: Metadata structure of a VM snapshot.

We assume a flat architecture in which all p machines that host VMs in a cluster can be used in parallel for deduplication. A small amount of local disk space and memory on each machine can be used to store global index and temporary data. The real backup storage can be either a distributed file system built on this cluster or use another external storage system.

The representation of each snapshot in the backup storage has a two-level index structure in the form of a hierarchical directed acyclic graph as shown in Figure 1. A VM image is divided into a set of segments and each segment contains content blocks of variable-size, partitioned using the standard chunking technique with 4KB as the average block size. The snapshot metadata contains a list of segments and other meta data information. Segment metadata contains its content block fingerprints and reference pointers. If a segment is not changed from one snapshot to another, indicated by a dirty bit embedded in the virtual disk driver, its segment metadata contains a reference pointer to an earlier segment. For a dirty segment, if one of its blocks is duplicate to another block in the system, the block metadata contains a reference pointer to the earlier block.

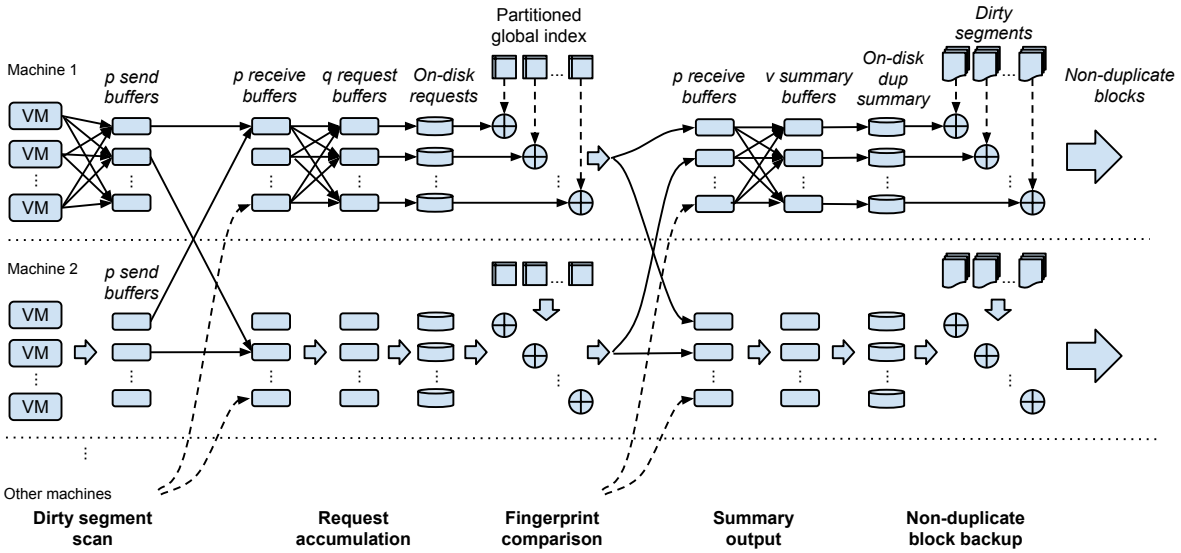

Figure 2: Processing flow of Stage 1 (dirty segment scan and request accumulation), Stage 2 (fingerprint comparison and summary output), and Stage 3 (non-duplicate block backup).

The data flow of our multi-stage duplicate detection is depicted in Figure 2. In Stage 1, each machine independently reads VM images that need a backup and forms duplicate detection requests. The system divides each dirty segment into a sequence of chunk blocks, computes the meta information such as chunk fingerprints, sends a request to a proper machine, and accumulates received requests into a partition on the local temporary disk storage. The partition mapping uses a hash function applied to the content fingerprint. Assuming all machines have a homogeneous resource configuration, each machine is evenly assigned with q partitions of global index and it accumulates corresponding requests on the disk. There are two options to allocate buffers at each machine. 1) Each machine has $p \times q$ send buffers corresponding to $p \times q$ partitions in the cluster since a content block in a VM image of this machine can be sent to any of these partitions. 2) Each machine allocates p send buffers to deliver requests to p machines; it allocates p receive buffers to collect requests from other machines. Then the system copies requests from each of p receive buffers to q local request buffers, and outputs each request buffer to one of the request partitions on the disk when this request buffer becomes full. Option 2, which is depicted in Figure 2, is much more efficient than Option 1 because $2p + q$ is much smaller than $p \times q$, except for the very small values. As a result, each buffer in Option 2 has a bigger size to accumulate requests and that means less disk seek overhead.

Stage 2 is to load disk data and perform fingerprint comparison at each machine one request partition at a time. At each iteration, once in-memory comparison between an index partition and request partition is completed, duplicate summary information for segments of each VM is routed from the fingerprint-based distribution to the VM-based distribution. The summary contains the block ID and the reference pointer for each detected duplicate block. Each machine uses memory space of the request partition as a send buffer with no extra memory requirement. But it needs to allocate p receive buffers to collect duplicate summary from other machines. It also allocates v request buffers to copy duplicate summary from p receive buffers and output to the local disk when request buffers are full.

Stage 3 is to perform real backup. The system loads the duplicate summary of a VM, reads dirty segments of a VM, and outputs non-duplicate blocks to the final backup storage. Additionally, the global index on each machine is updated with the meta data of new chunk blocks. When a segment is not dirty, the system only needs to output the segment meta data such as a reference pointer. There is an option to directly read dirty blocks instead of fetching a dirty segment which can include duplicate blocks. Our experiment shows that it is faster to read dirty segments in the tested workload. Another issue is that during global index update after new block creation, the system may find some blocks with the same fingerprints have been created redundantly. For example, two different VM blocks that have the same fingerprint are not detected because the global index has not contained such a fingerprint yet. The redundancy is discovered and logged during the index update and can be repaired periodically when necessary. Our experience is that there is a redundancy during the initial snapshot backup and once that is repaired, the percentage of redundant blocks due to concurrent processing is insignif-

icant.

The above steps can be executed by each machine using one thread to minimize the use of computing resource. The disk storage usage on each machine is fairly small for storing part of global index and accumulating duplicate detection requests that contain fingerprint information. We impose a memory limit M allocated for each stage of processing at each machine. The usage of M is controlled as follows and space allocation among buffers is optimized based on the relative ratio between the cross-machine network startup cost and disk access startup cost such as seek time. Using a bigger buffer can mitigate the impact of slower startup cost.

- For Stage 1, M is divided for 1) an I/O buffer to read dirty segments; 2) $2p$ send/receive buffers and q request buffers.
- For Stage 2, M is divided for 1) space for hosting a global index partition and the corresponding request partition; 2) p receive buffers and v summary buffers.
- For Stage 3, M is divided for 1) an I/O buffer to read dirty segments of a VM and write non-duplicate blocks to the backup storage; 2) summary of duplicate blocks within dirty segments.

Snapshot deletion. Each VM will keep a limited number of automatically-saved snapshots and expired snapshots are normally deleted. We adopt the idea of mark-and-sweep [7]. A block or a segment can be deleted if its reference count is zero. To delete useless blocks or segments periodically, we read the meta data of all snapshots and compute the reference count of all blocks and segments in parallel. Similar to the multi-stage duplicate detection process, reference counting is conducted in multi-stages. Stage 1 is to read the segment and block metadata to accumulate reference count requests in different machines in the fingerprint based distribution. Stage 2 is to count references within each partition and detect those records with zero reference. The backup data repository logs deletion instructions, and will periodically perform a compaction operation when its deletion log is too big.

4 Performance Analysis and Comparison

The system keeps at most x copies of snapshots for each VM on average. The total size of global content fingerprints is $x * s * v/c * u * (1 - d1) * (1 - d2)$ where c is the average chunk size and u is the meta data size of each chunk fingerprint. In practice $c = 4K$ and u/c is about 100. $x = 10$ in the case of Alibaba cloud.

Define $r = sv(1 - d_1)/c$ which is the average number of detection requests made by each machine, and the number of detection requests each machine must handle. We assume the load, i.e., amount of data to backup at each machine, is not balanced, so there is also r_max, which is the number of requests at the most heavily loaded machine. This is the case in the Alibaba cluster, with some machines being terabytes while the average machine is 40GB.

Here we develop a model for the total backup time, which becomes important in trying to minimize the CoW cost during backup *CoW should be explained and justified before this point in the paper*. The backup process can be broken into 4 stages, where each stage has two parts, first an all-to-all exchange of data, then local processing to prepare for the next stage.

In Stage 1a, the dirty segments are read from the virtual disk, the hash of each block is computed, and dedup requests are sent to the machine hosting the blocks' respective partitions. Due to the synchronization in each stage, Stage 1a needs to wait for the most heavily loaded machine to read all it's data - so the read stage depends on r_{max}, while the write stage, which is balanced by the hash partitioning, depends on r. We first read r blocks from disk, send r dedup requests, and then save the received requests to temporary files (one for each parition). The time for the first stage can be expressed as:

$$r_{max}c/b_r + \alpha_n \frac{ur_{max}}{m_n} + ru/b_w$$

In Stage 1b, each partition index is read from disk, then the dedup requests (from Stage 1a) for that paritition are processed and the results are written back out to disk. The results are broken into 3 groups: duplicate blocks, new blocks, and dup-with-new blocks, which are duplicates of blocks that are new to this batch.

Let n be the total number of index entries at each machine before the backup was started. $n = (vx)(1 - d_1)(1-d_2)\frac{e}{c}$. Since each machine holds a constant number q paritions, and the paritions are uniform in size as they are from the hash of the block, n is very even across the machines, even when the vm load is imbalanced.

The cost of Stage 1b is:

$$ru/b_r + ne/b_r + r\beta + re/b_w$$

In Stage 2a the new block results from Stage 1 are sent to the requesters, and in Stage 2b the new blocks are written out to the storage system. We will now mostly be dealing with the $r(1 - d_2)$ blocks that are new to the system (or $r_{max}(1-d_2)$ when we must wait for the most heavily loaded machine). In 2b the dedup results must be read and the actual disk blocks for each new block must be re-read before they can be sent to the block store. To avoid seeking we have found it faster to simply re-read all the dirty data and ignore the duplicate blocks rather than find only the new blocks on disk. The CoW locking on the filesystem cannot end until after the dirty data is re-read in Stage 2b.

The cost of Stage 2a is:

$$r(1-d_2)e/b_r + \alpha_n \frac{er_{max}(1-d2)}{m_n} + r_{max}(1-d_2)e/b_w$$

and the cost of Stage 2b is:

$$r_{max}c/b_r + r_{max}(1-d_2)e/b_r$$

After the new blocks have been written to the block store, and references to them have been obtained, those references must be returned to the parition index holder so that those blocks may be deduped in the future (and also so dup-with-new requests can be handled in this run). This Stage 3a, to read the new index entries, return them to the parition index holder, and save the received references, costs:

$$r_{max}(1-d_2)e/b_r + \alpha_n \frac{er_{max}(1-d_2)}{m_n} + r(1-d_2)e/b_w$$

Stage 3b consists of updating the partition index with the new block references from Stage 3a, and also updating the dup-with-new results from Stage 1. First we load the new references from Stage 2b into memory, and then dedup the dup-with-new requests against the new block references. Once the dup-with-new results have chunk references, we can add the dup-with-new results to the duplicate result files. We then add the new references to the corresponding partition index to handle future dedup requests. Stage 3b costs:

$$r((1-d_2)+d_3)e/b_r + rd_3\beta + r((1-d_2)+d_3)e/b_w$$

In the final stage (Stage 4), all duplicate references (including the dup-with-new references from 3b) are returned to the requesters, so that the snapshot recipes may be updated with references to those blocks. This process costs:

$$rd_2e/b_r + \alpha_n \frac{er_{max}d_2}{m_n} + re/b_w$$

4.1 A Comparison with Other Approaches

The memory space requirement for the data domain approach with bloom filter is:

$$x * rku(1-d2)/r$$

where r is the bloom filter with about 1:10 ratio in practice. The disk space used is

$$x * r * k * u * (1-d_2).$$

5 Round Scheduling

The simplest way to take advantage of the efficiency of batch processing is to schedule all the work to be done in one round. This works well if all of the data being backed up is just copies (e.g. a separate backup system which data is sent to over the network), however in our case we are backing up the original virtual disk, which may still be in use during the snapshot. This adds extra complexity to the cost analysis, because we must now also consider the cost of maintaining a constistent view during the snapshot process. We use the Copy on Write (CoW) provided by the virtual disk manager. With CoW, the duration of the backup affects how much data must be copied. Other studies have shown that as much as 8% or even more of total capacity must be reserved for CoW [11]. The actual cost of CoW is a factor of the data size, the write rate, and duration CoW is taking place. The data size isn't something we can change, nor the write rate, but we can minimize the duration that a given VM is undergoing CoW. We assume a poisson distribution of writes (which closely fits the meaured results from [11]), and then try to minimize the CoW cost using our performance and CoW model. Although the single batch schedule completes the backup in the smallest amount of time, it also has the greatest CoW cost because the most processing must be done before any VM can release the CoW lock.

Our basic CoW cost model is:

$$CoW = n(1-e^{-m/n})$$

where

$$m = tw(1-d_1)c$$

n is the number of dirty segments, t is the time under CoW, w is the write rate during CoW, d_1 is the % of clean data (which doesn't go under CoW), and c is a constant determining how likely writes are to touch dirty segments which haven't yet been copied vs. dirty segments which have already been copied during the current backup. With this CoW cost model and the earlier performance model, we can estimate the CoW cost of a given backup schedule. Note that CoW ends in Stage 2b, so only the time up to the end of 2b counts towards CoW. *we still need to pick a value for c*

The way to decrease the CoW cost is to break up the backup into multiple rounds, where in each round the CoW cost is minimized. The more rounds there are the shorter each round can be and therefore the smaller the CoW cost. The more rounds there are however the greater the backup overheads and some of the efficiency gained from batch processing is lessened. We balance these costs by setting a time limit on the whole backup job, and then develop an algorithm to schedule VMs into rounds.

With these goals a model that closely fits our goals is the dual version of the bin packing problem. In standard bin packing the goal is to fit all of the items into as few bins as possible, without overfilling any bins. In the dual

version of the problem however as many bins as possible are to filled to at least some minimum level. In our problem the constraint is to keep the total cost of the schedule under a time limit rather than a minimum bin level. We adapt an algorithm for dual bin packing[3] to fit our VM scheduling problem. The algorithm adapted is called iterated A and works by iteratively callng a bin packing heuristic A with the VMs to be scheduled and the number of rounds, using binary search to arrive at the best number of rounds. A(I,N) is defined to return the optimality of packing set I into N bins using A. We take this basic idea and look at several VM packing heuristics to arrive at an efficient packing algorithm. More formally, our adaptation of the iterated A algorithm can be defined as:

justify choice of initial UB (right now it is mostly arbitrary)

```
Set UB=min(n,2*v)
Set LB=1
while UB>LB
  set N = (UB+LB+1)/2
  if A(machines,N) > T, set UB=N-1
  else set LB=N
Halt
```

where A(I,N) returns the total backup time of the schedule
This algorithm returns the packing generated by A(machines,UB) after loop finishes

The general algorithm relies on a good choice of A to arrive at an efficent packing. Our first VM packing heuristic, A0, is a naïve approach very close to the unadapted algorithm from the dual bin packing paper. For both algorithms we develop, in case of ties, choose the left-most item.

A0:

```
sort VMs in descending order by size
while there is an unscheduled VM
  pick the first unscheduled VM a
    pick the round with the current
    lowest runtime b
  schedule VM a to round b
Halt
```

A0 decreases CoW cost (see Table **??**), but has room for improvement. The first issue is that the round with the current lowest runtime is chosen. Because backup time is dependent on a combination of the highest machine load and average machine load, if we add a VM to the currently most loaded machine in a round it will increase runtime much more than if we add the VM to a currently empty machine on an equally heavily loaded round *(insert figure to show how this happens)*. Therefore a better scheduling would be obtained if we pick the round with the lowest runtime after we simulate adding the new VM to that round. This new round picking heuristic takes into account that the same VM might have different affects on different rounds. This aspect of must be considered because we assume a VM must be backed up by the machine that hosts it. If the VMs are located on a DFS and can be backed up by any machine in the cluster then the problem of load balancing becomes a simpler but much different problem, and isn't considered here. We also pick the most heavily loaded (i.e. most data remaing) machine in case of ties so we can make the most backup progress in a round.

Another improvement we can make to A0 is to the VM picking heuristic. For the same reason as given above, the largest VM may not always have the greatest impact on time (e.g. picking a slightly smaller VM on the most heavily loaded machine has a greater impact than selecting the largest VM on a lightly loaded machine). A better way to pick VMs than just by size is to simulate removing VM's, then model the new single round time, and pick the VM whose removal most decreases the single round time. By picking VMs this way we take into account the machine load in picking which VM to remove, and minimize the time the remaining VMs will take to backup

After making the two above changes we arrive at VM packing heuristic A1.

A1:

```
while there is an unscheduled VM
  pick the VM a whose removal will
    most decrease single round time
    tie-breaker:VM on most heavily
    loaded round
  pick the round b whose runtime will
    be lowest after adding VM a
  schedule VM a to round b
Halt
```

A1 significantly improves our simulated results, bringing our CoW costs much closer to the best case than the worst case, as can be seen in Figure **??**. Our current implementation of the algorithm focuses on the deduplication and so doesn't make use of CoW, but we can see that our simulated times closely match measured runtimes (hopefully). *I haven't actually implemented the schedulers into the dedup yet to test this*

6 Evaluation

We have implemented and evaluated a prototype of our multi-stage deduplication scheme on a cluster of dual quad-core Intel Nehalem 2.4GHz E5530 machines with 24GB memory. Our implementation is based on Alibaba's Xen cloud platform [1, 16]. Objectives of our evaluation are: 1) Analyze the deduplication throughput

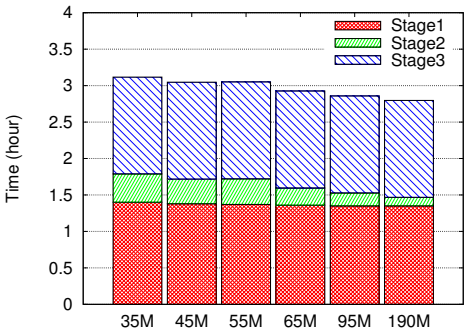

Figure 3: Parallel time when memory limit varies.

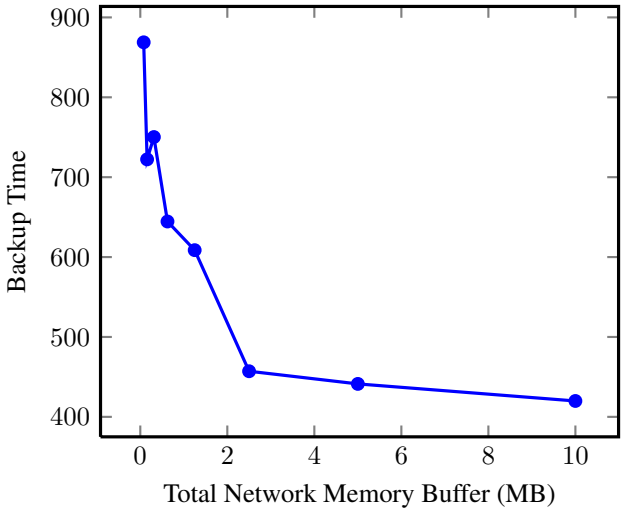

Figure 4: Backup time for verying amounts of memory allocated to network communication. Other Settings: 10 Machines, 5x40GB VMs per machine, write buffer 6.25MB, Read Buffer 128KB

and effectiveness for a large number of VMs. 2) Examine the impacts of buffering during metadata exchange.

We have performed a trace-driven study using a 1323 VM dataset collected from a cloud cluster at Alibaba's Aliyun. 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. Each VM image is divided into 2 MB fix-sized segments and each segment is divided into variable-sized content blocks with an average size of 4KB. The signature for variable-sized blocks is computed using their SHA-1 hash.

The seek cost of each random IO request in our test machines is about 10 milliseconds. The average I/O usage of local storage is controlled about 50MB/second for backup in the presence of other I/O jobs. Noted that a typical 1U server can host 6 to 8 hard drives and deliver over 300MB/second. Our setting uses 16.7% or less of local storage bandwidth. The final snapshots are stored in a distributed file system built on the same cluster.

The total local disk usage on each machine is about 8GB for the duplicate detection purpose, mainly for global index. Level 1 segment dirty bits identify 78% of duplicate blocks. For the remaining dirty segments, block-wise full deduplication removes about additional 74.5% of duplicates. The final content copied to the backup storage is reduced by 94.4% in total.

Figure 3 shows the total parallel time in hours to backup 2500 VMs on a 100-node cluster a when limit M imposed on each node varies. This figure also depicts the time breakdown for Stages 1, 2, and 3. The time in Stages 1 and 3 is dominated by the two scans of dirty segments, and final data copying to the backup storage is overlapped with VM scanning. During dirty segment reading, the average number of consecutive dirty segments is 2.92. The overall processing time does not have a significant reduction as M increases to 190MB. The aggregated deduplication throughput is about 8.76GB per second, which is the size of 2500 VM images divided by the parallel time. The system runs with a single thread and its CPU resource usage is 10-13% of one core. The result shows the backup with multi-stage deduplication for all VM images can be completed in about 3.1 hours with 35MB memory, 8GB disk overhead and a small CPU usage. As we vary the cluster size p, the parallel time does not change much, and the aggregated throughput scales up linearly since the number of VMs is $25p$.

Table 2 shows performance change when limit M=35MB is imposed and the number of partitions per machine (q) varies. Row 2 is memory space required to load a partition of global index and detection requests. When $q = 100$, the required memory is 83.6 MB and this exceeds the limit M =35MB. Row 3 is the parallel time and Row 4 is the aggregated throughput of 100 nodes. Row 5 is the parallel time for using Option 1 with $p \times q$ send buffers described in Section 3. When q increases, the available space per buffer reduces and there is a big increase of seek cost. The main network usage before performing the final data write is for request accumulation and summary output. It lasts about 20 minutes and each machine exchanges about 8MB of metadata per second with others during that period, which is 6.25% of the network bandwidth.

7 Conclusion Remarks

The contribution of this work is a low-cost multi-stage parallel deduplication solution. Because of separation of duplicate detection and actual backup, we are able

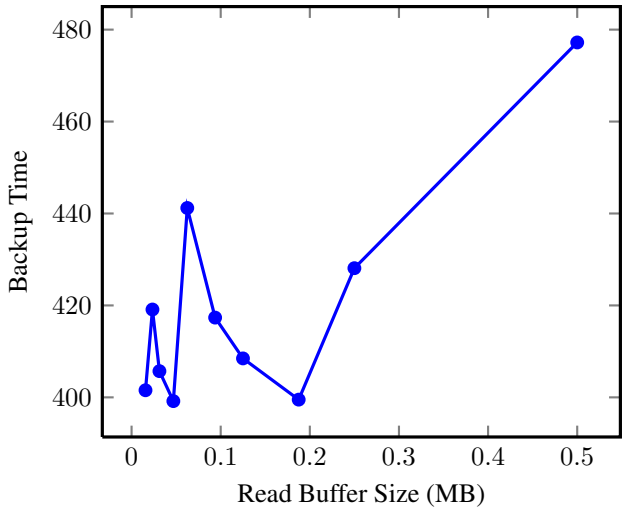

Figure 5: Backup time for verying amounts of memory allocated to disk read buffering. Other Settings: 10 Machines, 5x40GB VMs per machine, network buffers 2.5MB, write buffer 6.25MB

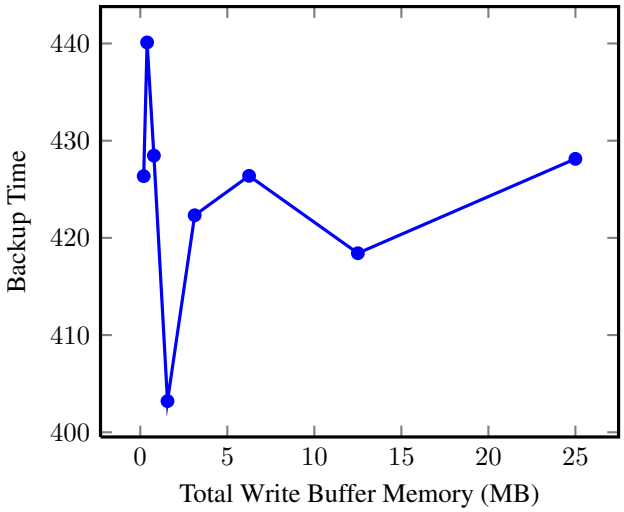

Figure 6: Backup time for verying amounts of memory allocated to network communication. Other Settings: 10 Machines, 5x40GB VMs per machine, network buffer memory 2.5MB, Read Buffer 128KB

to evenly distribute fingerprint comparison among clustered machine nodes, and only load one partition at time at each machine for in-memory comparison.

The proposed scheme is resource-friendly to the existing cloud services. The evaluation shows that the overall deduplication time and throughput of 100 machines are satisfactory with about 8.76GB per second for 2500 VMs. During processing, each machine uses 35MB memory, 8GB disk space, and 10-13% of one CPU core with a single thread execution. Our future work is to conduct more experiments with production workloads.

Acknowledgment. We thank Michael Agun, Renu Tewari, and the anonymous referees for their valuable comments. This work is supported in part by NSF IIS-1118106. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

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

[2] C. Alvarez. NetApp Deduplication for FAS and V-Series Deployment and Implementation Guide. NetApp. Technical Report TR-3505 , 2011.

[3] S. Assmann, D. Johnson, D. Kleitman, and J.-T. Leung. On a dual version of the one-dimensional bin packing problem. *Journal of Algorithms*, 5(4):502 – 525, 1984.

[4] 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.

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

[6] EMC. Achieving storage efficiency through EMC Celerra data deduplication. White Paper, 2010.

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

[8] 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.

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

[10] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *USENIX ATC'08*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[11] H. Shim, P. Shilane, and W. Hsu. Characterization of incremental data changes for efficient data protection. In *USENIX ATC '13*, pages 157–168.

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

[13] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. In *FAST'09*, pages 225–238, 2009.

[14] 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.

[15] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance de-duplication stor-

age system for backup and archiving. In *IEEE IPDPS*, pages 1–12, 2010.

[16] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for vm snapshots in cloud storage. In *IEEE CLOUD'12*, pages 550–557.

[17] 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.

p	the number of machines in the cluster
v	the number of VMs per machine. At Alibaba, $v = 25$
x	is the number of snapshots saved for each VM. At Alibaba, $x = 10$
t	the amount of temporary disk space used per machine for deduplication
m	the amount of memory used per machine for deduplication. Our goal is to minimize this
s	the average size of virtual machine image. At Alibaba, from our collected data, $s = 40GB$
d_1	the average deduplication ratio using segment-based dirty-bit. $d_1 = 77\%$
d_2	the average deduplication ratio using content chunk fingerprints after segment-based deduplication. For Alaba dataset tested, $d_2 = 50\%$
d_3	the average number of dup-with-new blocks, as a fraction of r (defined below)
b_r	the average disk bandwidth for reading from local storage at each machine
b_w	the average disk bandwidth for writing to local storage at each machine
b_b	average write bandwidth to back-end storage (block store)
q	the number of buckets to accumulate requests at each machine. (total number of buckets is $p * q$)
c	the chunk block size in bytes. In practice $c = 4KB$
u	the record size of detection request per block. In practice, u=40. That includes block ID and fingerprint
e	the size of a duplicate summary record for each chunk block
m_n	the memory allocated to network send & receive buffering. Total network memory is $2m_n$, wich each buffer of size m_n/p
α_n	the latency for sending a message in a cluster
β	time cost for in-memory duplicate comparison

Table 1: Modeling parameters and symbols.

Table 2: Performance when M=35MB and q varies.

#Partitions (q)	100	250	500	750	1000
Index+request (MB)	83.6	33.5	16.8	11.2	8.45
Total Time (Hours)	N/A	3.12	3.15	3.22	3.29
Throughput GB/s	N/A	8.76	8.67	8.48	8.30
Total time (Option 1)	N/A	7.8	11.7	14.8	26