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

Daniel Agun*, Wei Zhang*, and Tao Yang*
* University of California at Santa Barbara,

## 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 a backup service that 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. Another tradeoff of our efficient batched approach, related to reading each segment twice, is that we must maintain a consistent view of the disk between the first and second read. We use Copy on Write (CoW) to acheive this, at the cost of additional disk space. With careful parallelism and buffer management, this multi-stage detection scheme can provide a sufficient throughput for VM backup. To minimize the CoW cost and adjust the balance between total backup time and the backup times for individual VMs, we adopt a multiple-batch approach based on the dual bin packing problem.

## 2 Background and Related Work

At a cloud cluster node providing access to multiple virtual machines, each instance of a guest operating system runs on a virtual machine (VM), accessing virtual hard disks represented as virtual disk image files in the host operating system. For VM snapshot backups, file-level semantics are normally not provided, and snapshot oper-

ations take place at the virtual device driver level. This meands no fine-grained file system metadata can be used to determine the changed data.

Another issue is the sheer size of the backups. to maintain mutiple daily uncompressed snapshots of each VM would be very expensive, even though most of the data remains the same from day to day, and even between VMs common software packages and operating systems mean that the amount of unique data is much smaller. 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. Approximation techniques which reduce the memory requirement at the expense of deduplication efficiency are studied in [4, 7, 16]. 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, efficiently finishing the backup of VM images within a reasonable time window is more important than optimizing individual VM block backup requests, and by batch processing the deduplication requests we can decrease resource usage.

## 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 the virtual disk driver to maintain segment dirty bits, which has a negligible space cost (1 bit per 2 MB). In the second level of deduplication, content blocks of dirty segments are compared with the fingerprints of unique blocks from 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 follow the VM distribution among machines and the detected duplicate summary should be collected following the same distribution. Therefore, there are four all-to-all data redistribution operations involved. One is to map detection requests from VM-based distribution to fingerprint based distribution, and another one is to map duplicate summary from fingerprint-based distribution to VM based distribution. Two additional rounds are required to ensure no duplicate blocks are missed and that both the partition index holder and VM owning machines have correct data references. 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.
- **Round Scheduling**. When all data is scheduled to be deduplicated in one round, the cost of Copy on Write (CoW) can be significant, as explored in other papers[11]. We therefore develop an algorithm to break up the data into multiple rounds to make the best use of the efficiency of batch processing, while minimizing the cost of CoW.

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

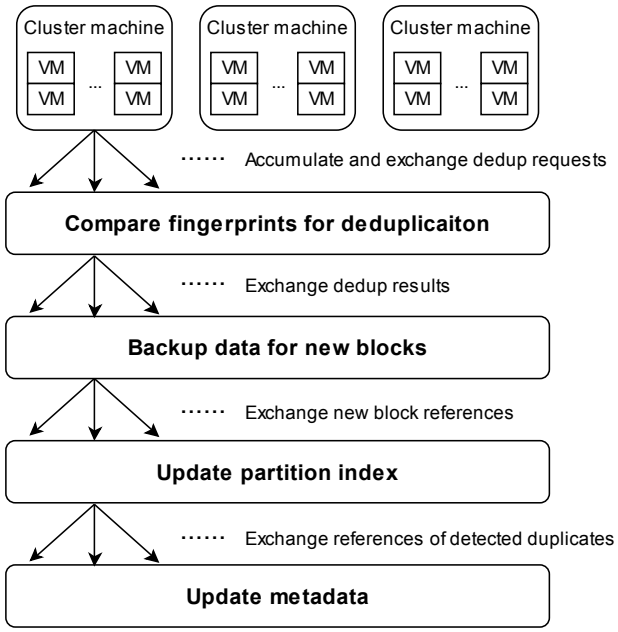


Figure 1: Overall architecture of parallel VM snapshot deduplication

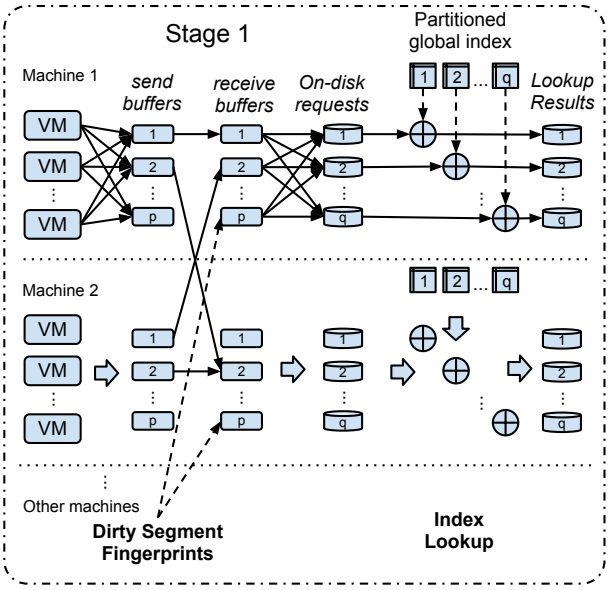


Figure 2: Processing flow of Stage 1.

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.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.

In Stage 1a, 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 **??**, 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 1b 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. One potential issue is that there may be multiple copies of a new block added to the system during the same round. We

call these dup-with-new blocks and the redundandy is discovered and logged during the index lookup. Our experience is that there is significant redundancy during the initial snapshot backup and after that, the percentage of redundant blocks due to concurrent processing is small.

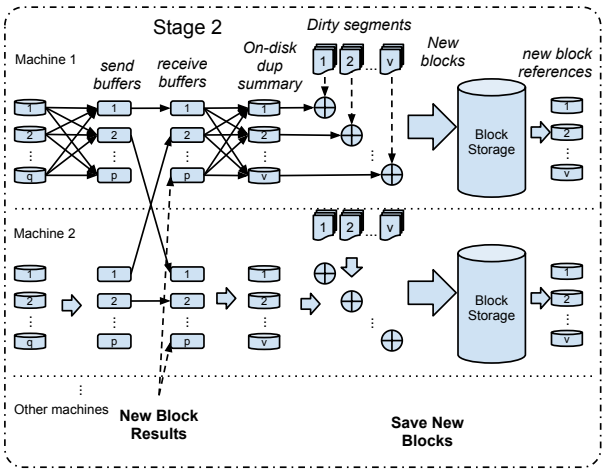


Figure 3: Processing flow of Stage 2.

In stage 2a the deduplication results corresponding to unique (new) data blocks are exchanged using $p$ send and receive buffers to route the responses to the deduplication requesters. Dup-with-new blocks are not exchanged, as they are treated as duplicate blocks, only without data references (the data references to these blocks are added in Stage 3).

Stage 2b is to perform the 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, references to each new data block are saved so that the global index may be updated.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.

In Stage 3a the machines exchange references to new blocks. This is required because it is the VM holder that backs up the data, but the partition index holder that must perform index lookups. Each machine reads the references obtained for all new blocks written, and then sends those references to the partition index holder, and the received references are saved to disk during 3a.

Stage 3b is where the index update occurs; new blocks are added to each partition index, and additionally dup-with-new blocks are re-read to obtain references. The dup-with-new blocks, now with references, are added to the duplicate result lists to be returned in Stage 4.

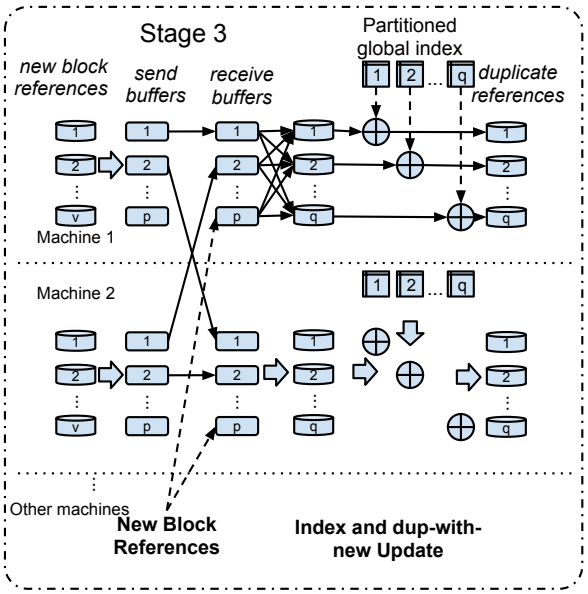


Figure 4: Processing flow of Stage 3.

In the final Stage of our algorithm, Stage 4, All duplicate references, including dup-with-new blocks, are returned to the requesters from the index holders in 4a, And in 4b the snapshot recipes are updated with those pointers.

The above steps can be executed by each machine using one thread to minimize the use of computing resources. The disk storage usage on each machine is fairly small for storing part of the 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. $M$ includes space for network communications and buffering data which is being sent to disk. We have found that as long as the write/read buffers are not unreasonably small (e.g. 4KB), the size of the disk buffers does not have a great impact on backup time, so we allocate small disk buffers (e.g. 128KB read and 2.5MB shared across all write buffers), and then allocate the remaining memory to index space or network buffers, depending on the stage.

- For Stage 1, $M$ is divided for 1) an I/O buffer to read dirty segments; 2) $2p$ send/receive buffers and $q$ disk buffers for deduplication requests.
- 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

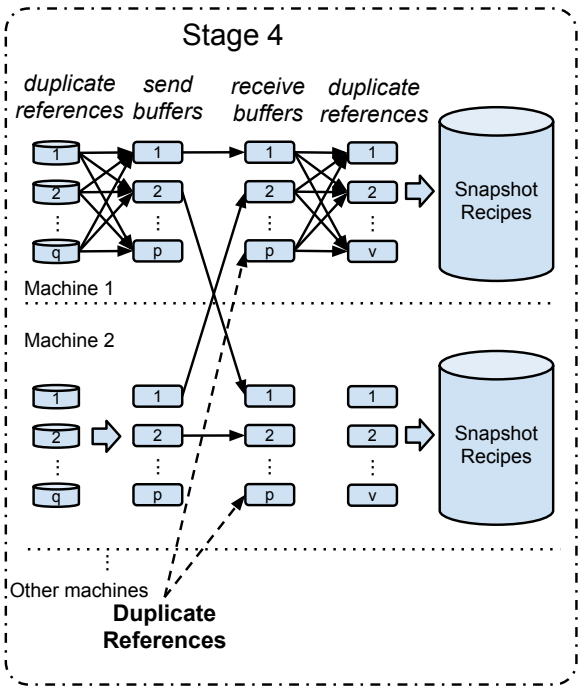


Figure 5: Processing flow of Stage 4.

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

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 the first 3 stages each have two parts, first an all-to-all exchange of data, then local processing to prepare for the next stage.

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.

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 dedup results are written back out to disk. The results are broken into 3 groups for each partition: 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 new block 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 round). 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 initially created in 1b. We then add the new references to the corresponding partition indices 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

## 5 Round Scheduling

The naive way to take advantage of the efficiency of batch processing is to schedule all the work to be done in one round. However, in our case we are backing up the live virtual disks, thus such single round backup presents several problems in our synchronous approach. This first is that in order to preserve a consistent view of VM images, we apply Copy-on-Write (CoW) to protect the dirty segments against incoming disk writes during the backup period of stage 1 and 2. Previous studies have shown that as much as 8% or even more of total capacity must be reserved for CoW [11], while the actual cost of CoW is a factor of the data size, the write rate, and duration of CoW protection. The second problem is that a single round backup brings the highest cost of CoW because of the workload in stage 1a is imbalanced, causing machines with small VMs to wait while CoW protection is turned on. Thus we must minimize the duration that a given VM is undergoing CoW protection.

To evaulate our CoW cost, we assume the incoming writes point to random disk locations, which would incur a CoW operation over a segment if the write location fall into a dirty segment for the first time (consequential writes to that segment will be re-directed to the new copy so there's no extra cost). Therefore the number of dirty segments needs to be copied represents the CoW cost of a VM.

Let $t$ be the time that a VM is under Cow protection, $w$ be the write rate during that period, $n$ be the total number of segments and $d_1$ be the percentage of clean segments that don't need to be protected, then the total number of dirty segments is

$$n_d = n(1-d_1)$$

and the total number of writes fall into dirty segments is

$$w_d = tw(1-d_1)$$

Then the total number of dirty segments that need to be copied is:

$$Cost = n_d(1-e^{-w_d/n_d}) = n(1-d_1)(1-e^{-tw/n})$$

From the above formula it is clear that decreasing CoW time $t$ will reduce the CoW protection cost, the model we derive is similar to the results meaured by [11].

In addition to backup time for individual VMs, another cost consideration is total backup time for the system. It is generally preferred to run backups during low demand hours to make use of otherwise unused time and minimize the impact on users. This leaves a window of several hours (e.g. from 1am to 4am) to complete the backup. The more rounds there are the shorter each round can be and therefore the smaller the VM backup times and the total CoW costs. The more rounds there are however the greater the overheads and some of the efficiency gained from batch processing is lessened. We balance these costs by developing an algorithm to schedule VMs into rounds.

*START OF DUAL BIN PACKING BASED ALGORITHM* From our requirements 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

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

Figure 6: Iterated A scheduling algorithm, requires good choice of A to be effective

```
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
```

Figure 7: Naïve round packing heuristic based on VM size and current round runtime

as possible are to filled to at least some minimum level. This fits our purpose because we want to maximize the number of rounds to minimize VM backup times, with some constraint to keep the efficiency benefits of batch processing and keep the total backup time low. In our dual-bin-packing solution the constraint is to keep the total cost of the schedule under a time limit rather than mandating a minimum bin level. We have adapted 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 items to be scheduled and the number of bins, using binary search to arrive at the best number of bins. A(I,k) is defined to return the optimality of packing set I into k 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 is defined as:

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

where A(machines,k) returns the total backup time of the schedule for k rounds
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 does provide for shorter rounds (see Table **??**), but the efficiency of the packing is poor due to several issues 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 different round where that machine is currently empty. You can see this in Figure 8, where VM1D must be scheduled to one of two rounds which currently have the same aver-

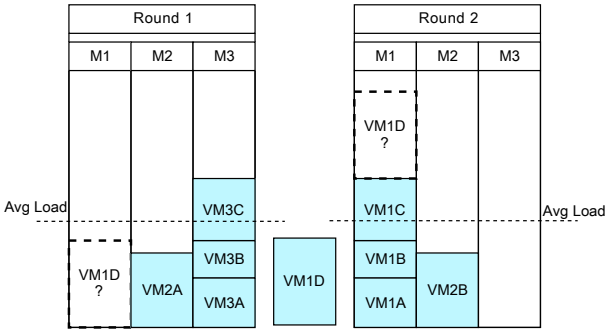


Figure 8: Choosing a round to schedule a VM to. Though the rounds look identical in terms of total/average load, Round 1 is a much better choice.

age/total load. The effect of VM1D on round 1 would be much less than the effect on round 2 because we wouldn't be increasing the maximum machine load of the round, so we should schedule to round 1. Therefore a better scheduling heuristic is one based on predicted runtime after adding the VM, and picking the round with the lowest runtime after adding the 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 round picking 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 such that they can be backed up by any machine in the cluster then the problem of load balancing becomes a simpler but much different problem, but 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, given equivalent equivalent options.

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). Our improved hueristic to pick VMs is to predict

```
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
```

Figure 9: More effective round packing heuristic based on minimizing remaining work and predicted round time

the one-round runtime after removing that VM, and pick the VM whose removal most decreases the single round time (i.e. the VM that has the greatest impact on overall backup time). By picking VMs this way we take into account the machine load in picking which VM to schedule next, 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 significantly improves our simulated results, also 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).

*END OF DUAL BIN PACKING BASED ALGORITHM*

The problem of scheduling VM backup tasks to specific rounds is a variant of bin packing, with some additional constraints. We want to efficiently schedule VMs to rounds, minimizing average VM backup time (and thereby minimizing CoW), while also taking into account resource usage and total jobspan (which are minimized in the single round schedule). We solve the problem with a 2-part algorithm. First, we have a round packing heuristic which takes a number of rounds and a set of VMs, and returns a schedule for the backup process. Second, we have a higher level algorithm that determines the optimal number of rounds, given the constraints.

Our round packing heuristic uses a descending sorted order worst fit packing, Scheduling VMs machine by machine, and is described in Fig 10. For our higher-level scheduling algorithm, we minimize Resource usage and average backup time, while ensuring that the jobspan constaint is met (if possible). The overall scheduling algorithm is described in Fig 11, where T is the alotted time for backup. our cost function is defined as

```
for each Machine m
  sort VMs in descending order by size
  while there is an unscheduled VM
    pick the first unscheduled VM a
    pick the round b with the least
      amount of VM data for m
    schedule VM a to round b
Halt
```

Figure 10: Round packing heuristic A

```
Set UB=min(n,2*v)
set k=1
while k < UB
  sch = A(machines,k)
  if jobspan(sch) > T
    return A(machines,min(1,k-1))
  cost = cost(sch);
  if k=1 OR cost < min_cost
    min_cost = cost;
    min_sch = sch;
return min_sch;
```

Figure 11: Scheduling Algorithm

$cost(s) = \alpha \times Resource\ Usage(s) + (1 - \alpha) \times Avg\ Backup\ Time(s)$.

Using the above algorithm we can show that for any round $r$, the amount of data scheduled to $r$ across all VMs has the upper bound of twice the optimal amount of data for a perfectly balanced schedule.

Proof: first we define $load(m, r)$ to be the amount of data from VMs on Machine $m$ to round $r$, $Machines$ to be the set of all machines, and $Rounds$ to be the set of all rounds. now define $2b_m$ to be the upper bound for data backed up in a single round, where $b_m = min(\frac{\sum\{VMs\ on\ m\}}{|Rounds|}, largest\ VM\ on\ m)$, and $2b$ to be the global upper bound for each round, where $b = \sum\{b_m | m \in Machines\}$. We would like to prove the actual global load is less than $2b$. we will do this by proving that for each machine $2b_m$ is the upper bound, which from the definition of $b$ can be extended to a global uppoer bound of $2b$. More formally, we would like to show:

$$(\forall m \in Machines, \forall r \in Rounds) load(m, r) < 2b_m$$

Now let $x$ be a VM from Machine $m$ to be scheduled. For each round $r$ there are two possibilities:

- $load(m, r) < b_m \implies load(m, r) + x < b_m + x$
- $load(m, r) \geq b_m \implies$
  $(\exists k \in Rounds) load(m, k) < b_m$

since we never schedule to a round where $load(m,r) \geq b_m$, $load(m,r)$ must be less than $b_m + x$, where $x$ is the largest VM on $m$. By Definition $b_m > x$, so $load(m,r) < 2b_m$ for each machine, and therefore the global load for each round is less than $2b$. The expected case can be improved though by packing the largest VMs first and always scheduling to the round with the least data.

## 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 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.

*The names of the algorithms are just the temporary names I've been using for coding, DBP2 is the relative of the dual bin packing algorithm, BPL is the machine by machine largest VM to shortest round algorithm*

Figure 12 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

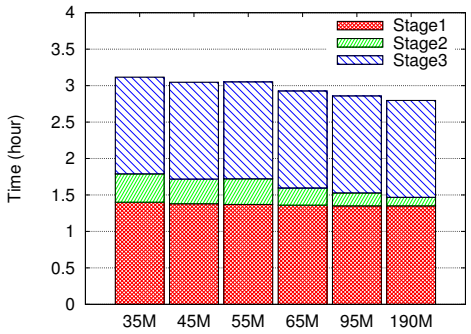


Figure 12: Parallel time when memory limit varies.

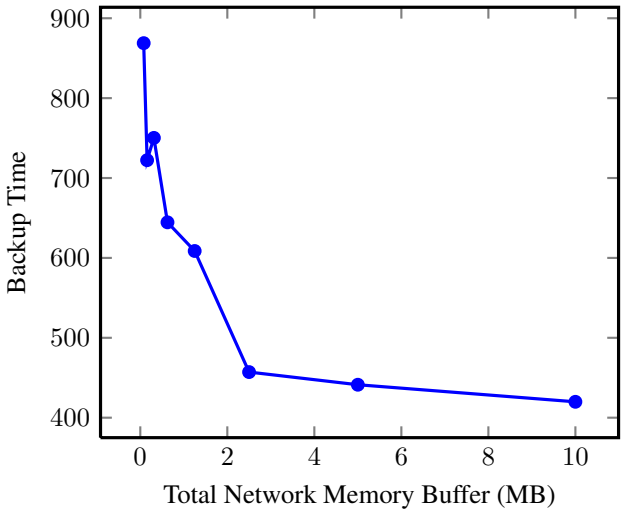


Figure 13: Backup time for verying amounts of memory allocated to network communication. Other Settings: 10 Machines, 5x40GB VMs per machine, write buffer 6.25MB, Read Buffer 128KB

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

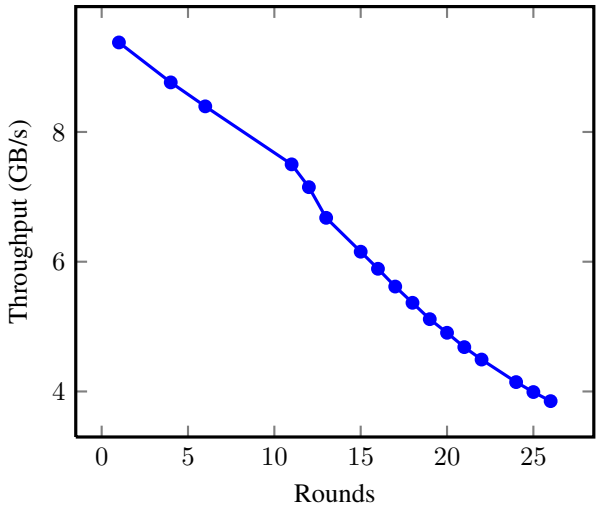


Figure 14: simulated results for average throughput for individual VMs in 100 node cluster.

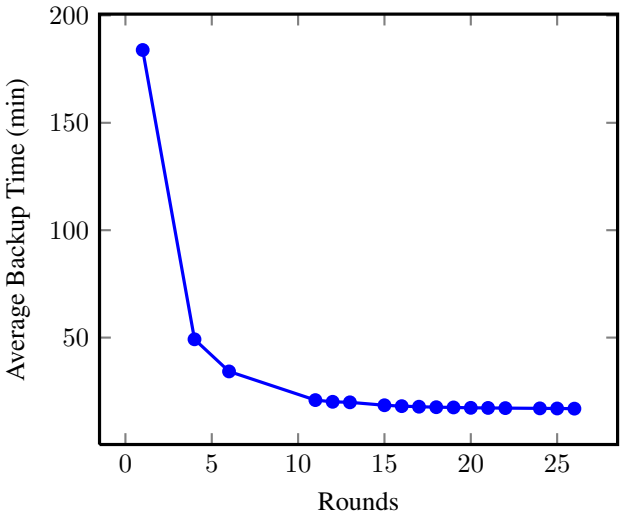


Figure 15: simulated results for average vm backup time for individual VMs in 100 node cluster.

$M$=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 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.

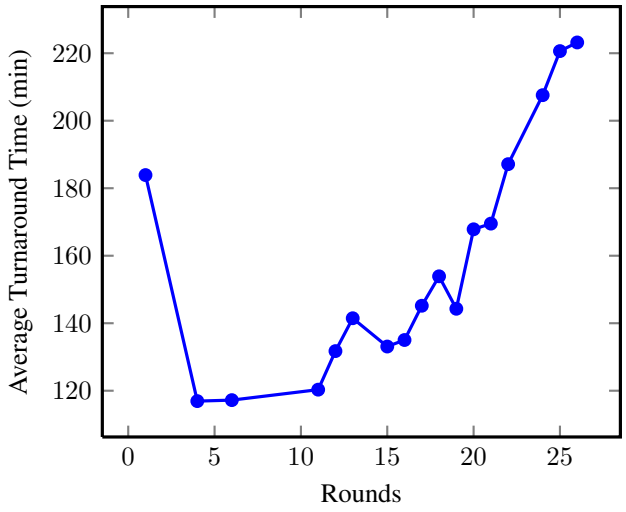


Figure 16: simulated results for average vm turnaround time for individual VMs in 100 node cluster.


[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 storage 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.


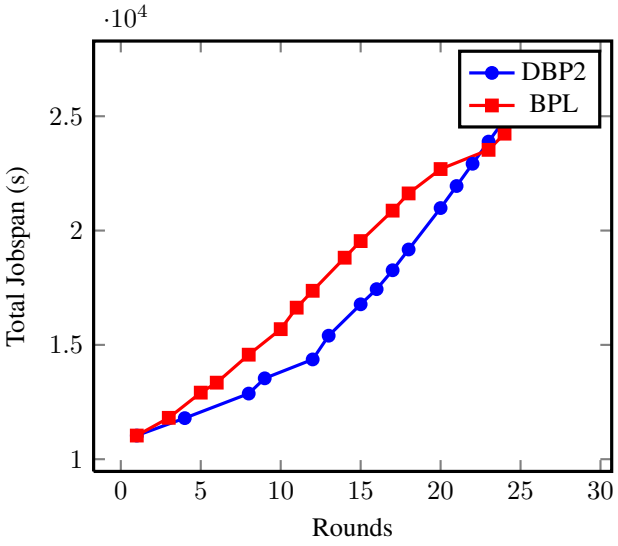


Figure 17: Jobspan for DBP2 and BPL

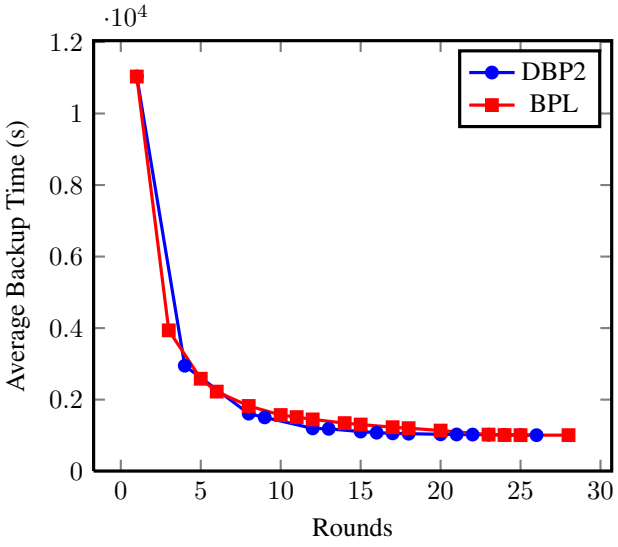


Figure 18: Average Backup Time for DBP2 and BPL

| | |
|---|---|
| $p$ | 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 $m_n/p$ |
| $\alpha_n$ | the latency for sending a message in a cluster |
| $\beta$ | 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 |