Data Deduplication and Zipf-like Distribution in VM Cloud Storage

Wei Zhang, Hong Tang, Hao Jiang, Rachael Zeng, Xiaogang Li, Tao Yang

Abstract

Virtualization has became the engine behind many cloud computing platforms. In a virtualized computing environment, virtual machines operate on virtual disks, so backing up user data is done by constantly taking snapshots of those virtual disks. Because such VM snapshots are huge in terms of both quantity and individual sizes, snapshot storage would cost tremendous amount of space without deduplication. Current snapshot deduplication is mainly done through copy-on-write on fixed-size disk blocks. Such solutions cannot handle the cross VM data duplication because VMs do not share any data. In addition, storing VM images and their snapshots in the same storage engine reduce the underline design flexibility because these two kinds of data have distinct access requirements.

In this paper, we first perform a large scale study in production VM clusters to show that cross VM data duplication is severe due to they have large amount of common data. Then our data analysis finds out that the overall data duplication pattern follows the Zipf's law. Base on these discoveries, we propose a snapshot storage deduplication scheme using variable-size chunking to address the above problem efficiently. We eliminate the majority of cross VM data duplication by pre-select a small set of frequently seen data blocks to be shared globally, and we also remove many cross snapshot duplication by using smaller chunking granuarity and locality. Experiment shows our design can achieve high deduplication ratio with very limited amount of resources in large scale VM cloud environment.

1 Introduction

Virtualization is the engine behind many popular cloud computing platforms. Amazon, Aliyun and many others have provided public VM clouds that host hundreds of thousand virtual machines(VM) which are dynamically created by users. In such a virtualized environment, each instance of guest operating system operates on virutal disks, which are represented as files called virtual disk image (e.g. .vhd, .vmdk) in the host operating system. Because those virtual disk images are stored as files in the external storage, backing up VM data in virtual disks is mainly done by taking snapshots of virtual disk images, which is quite different from traditional file system backup.

The most widely used snapshot method is copy-on-write (CoW). Upon VM image storage system receives a save snapshot request, it freezes the state of that image file, then all consequent write request will result in the write region being copied to a incremental snapshot data file. CoW method can finish a snapshot operation instantly, but it has several disadvantages: first, CoW may affect the general I/O performance due to defered data coping. Second, CoW does not seperate backup data and runtime image data, which have distinct access requirements: runtime image data is directly used by the running VM, thus need high throughput, and is very sensitive to latency, such data must be served with hig cost hardware, but backup data generally only need fair aggregate throughput, is not sensitive to latency, thus can be stored in secondary storage devices. Finally and most important, VM snapshots contain tremendous amount of data duplication, which is nearly impossible to tackle if these two kinds of data are tightly coupled together.

Data deduplication technique can eliminate such redundancy, today many D2D backup systems[2][1]. uses vaiable-size chunking algorithm to detect duplicates in file data. Chunking divides a data stream into variable length chunks, it has been used to conserve bandwidth[9], search and index documents in large repositories[4], scalable storage systems[6], store and transfer large directory trees efficiently and with high reliability[7].

To chunk a file, starting from the first byte, its contents as seen through a fixed-sized (overlapping) sliding window are examined. At every position of the window, a fingerprint or signature of its contents, f, is computed using hashing techniques such as Rabin fingerprints[11]. When the fingerprint meets a certain criteria, such as $f mod D = r$ where D, the divisor, and r are predefined values; that position of the window defines the boundary of the chunk. This process is repeated until the complete file has been broken down into chunks. Next, a cryptographic hash or chunk ID of the chunk is computed using techniques such as MD5 or SHA. After a file is chunked, the index containing the chunk IDs of backed up chunks is queried to determine duplicate chunks. New chunks are written to disk and the index is updated with their chunk IDs. A file recipe containing all the information required to reconstruct the file is generated. The index also contains some metadata about each chunk, such as its size and disk location. How much deduplication is obtained depends on the inherent content overlaps in the data, the average size of chunks and the chunking method[10]. In general, smaller chunks yield better deduplication.

Inline deduplication is deduplication where the data is deduplicated before it is written to disk as opposed to post-process deduplication where backup data is first written to a temporary staging area and then deduplicated offline. One advantage of inline deduplication is that extra disk space is not required to hold and protect data yet to be backed up. However, unless some form of locality or similarity is exploited, inline, chunk-based deduplication, when done at a large scale faces what has been termed the disk bottleneck problem: to facilitate fast chunk ID lookup, a single index containing the chunk IDs of all the backed up chunks must be maintained. As the backed up data grows, the index overflows the amount of RAM available and must be paged to disk. Without locality, the index cannot be cached effectively, and it is common for nearly every index access to require a random disk access. This disk bottleneck severely limits deduplication throughput.

This problem has been addressed by many previous studies. Zhu[12] tackle it by using an in-memory Bloom Filter and prefetch groups of chunk IDs that are likely to be accessed together with high probability. Lillibridge[8] break list of chunks into large segments, the chunk IDs in each incoming segment are sampled and the segment is deduplicated by comparing with the chunk IDs of only a few carefully selected backed up segments. These are segments that share many chunk IDs with the incoming segment with high probability. Deepavali[3] uses Broder's theorem[5] to find similar files and group them into the same physical location (bins) to deduplicate against each other.

In cloud storage, we are solving data deduplication problem in a different context of data stream deduplication (the D2D case). Because our snapshot storage servive is co-located with runtim VMs, we have very limited amount of system resource to perform deduplication. For example, on our typical 8-core, 48GB memory, 12TB disk server, there lives over 20 VMs who are hungrily eating system resources: some are running map-reduce jobs, some are busy web servers, some are back-end databases. Any behavior that affects user VMs performance or stability is unacceptable.

2 Zipf-like Distribution and Common Data Set

In Aliyun's VM cloud, each VM has one OS disk, plus one or more data disks. During VM's ceation, its OS disk is directly copied from user's choosen base image. Through its lifetime user may change configurations, install software, or write user data into OS disk, but most of the OS and software related data shall keep unchanged. Previous study has also supported this[**?**]. Therefore, we can let all OS disks share these common data in their snapshot backups.

Furthermore, our study on user's data disks has shown that the duplication pattern of variable-sized data blocks follows *Zipf-like* distribution. As a result, the major portion of deduplication effect will emerge from eliminating the duplication of frequently seen data.

2.1 OS CDS

We define *OS CDS* to be the unchanged data across the base VM image and the snapshots of all OS disks which are derived from it. Such data are brought by the operating system and popular software installations, they are rarely modified or deleted, and can be easily extracted using base image data as a hint.

To see the data duplication of OS CDS, we choose 7 major OSes in our VM cloud platform, which are: Win2008 Server 64 bits, Win2003 Server 32 bits, Win2003 Server 64 bits, RHEL, CentOS, Ubuntu Server and Debian (all Linux distributions are 64 bits). Then we pick 5 VMs for each OS, and 10 snapshots for each VM. These VMs have been actively used by their owners, their OS disks are heavily modified, some users even store huge amount of user data on their OS disks. For example, some Ubuntu OS disks have only 2GB of data while some have 50GB.

We take the following steps to evaluate the effect of OS CDS: for each OS, the base image and all OS disk snapshots are splitted into variable-sized blocks using TTTD algorithm. Then for every data block in base image, we check if this block appear in all 50 snapshots.

The above analysis is only an estimation of OS and software related common data, some user installed software may not be discovered by this method because they are not included in the base image. However, as we can see in the next section, the overall duplication of user generated data follows the Zipf-like distribution, thus all popular data can be discovered by statistical analysis.

2.2 Zipf-like Distribution and Data CDS

Zipf's law has been shown to characterize use of words in a natural language, city populations, popularity of website visits and other internet traffic data.

Here we present the first fully consistent empirical study showing that data duplication pattern in very large scale storage follows Zipf-like distribution. More specifically, we show that the duplication count of any unique data block is inversely proportional to its rank in the frequency table, with α smaller than unity. As far as we know, this is the first study of large scale real user data to disclose the Zipf-like distribution pattern in data storage area.

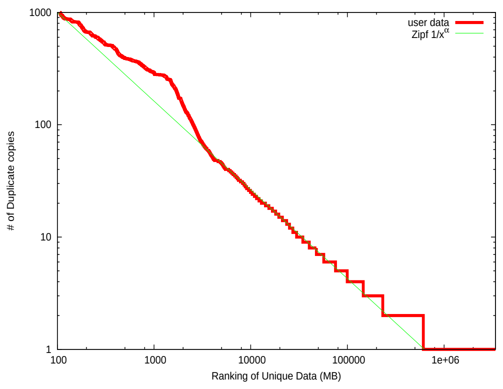

Figure 1: Number of duplicate copies vesus block ranking

2.2.1 The model

Consider a general storage system that holds user's files without any deduplication. Let N be the total number of unique data blocks after content-defined chunking and deduplication, $C_N(i)$ be the number of duplicate copies that the ith data block being stored in storage, given all the unique data blocks be ranked in order of their number of duplicate copies. Let S_i be the size of ith block, then the actual ranking of the ith block is reflected by $\sum_1^{i-1} S_i$.

2.2.2 Methodology

1323 users' virtual data disks were scanned using TTTD content-defined chunking, and we performed global perfect deduplication to caculate the number of duplicate copies of each individual unique block. We choose 2KB, 4KB, 16KB as the minimum, average and maximum block size, all variable-sized blocks are compared by their SHA-1 hash arther than real data.

We choose user's data disks rather than OS disks in thie experiment for several reasons: First, the data in OS disks are instinctively highly similar, because most of the VM users only make some common or unique but tiny changes to their OSes, so the data duplication pattern in OS disks cannot reflect the real distribution of general user data. Second, the data disks are way more important in terms of data safty and backup because they are what users really care about.

2.2.3 Result

Figure 1 shows the block duplication count vesus its ranking by that count. The almost straight line proves the block duplication in user generated data follows Zipf-like distribution, with α close to 0.78. A tiny hottest fraction of data lead to majority of data duplication, as a result, we can target at this small set of commonly seen data to design our fast and lightweight deduplication process, just like what people did in CDN and web proxy.

2.3 Discussion

By storing the hash index of a small set of hottest data (CDS), we expect to acheive great deduplication effect with minimal cost. But one problem of the CDS is what we called *blind spot*: the update of CDS is always behind new data's arrival, as a result, some data must have been written to the backup store before being collected into CDS, thus such duplication may comprise the effect of CDS. For instance, when a MySQL update is released, some users may apply it very quickly, so the new hot data bring by this update will not be filtered by CDS. Only until later our CDS update includes those new data, then the rest of MySQL users get benefited. We can take a few measurements to reduce the impact of blind spot. First, the data in blind spot will not accumulate, because snapshots backup is a paid service, so over the long term, data in blind spot will be removed from backup store as VMs and old snapshots keep being deleted. Second, for the major OS updates, we can anticipate them in advance by putting the new hot data into CDS before users widely adopt them.

Another problem associated with CDS is that some data in CDS may no longer be popular or even disappear as time goes. For example, an MySQL software update may invalidate some data that we previously hold in the CDS. If some data blocks in CDS are no longer referenced, our map-reduce process is able to detect this situation, so invalid data will be removed and new data will be added during the next CDS update. But if some users choose to keep their old MySQL version, then the corresponding old and new data will co-exist in the CDS.

3 Our Approach

3.1 Aliyun's VM Cloud

Aliyun provides the largest public VM cloud in China which is based on the open-source Xen technology. A typical VM cluster in our cloud environment consists of from hundreds to thousands physical machines, each of which can host tens of virtual machines. A few varieties of mainstream OS are supported, including several editions of Windows Server, major Linux distributions, and FreeBSD. During the VM creation, user choose his flavor of OS distribution, our system will copy the corresponding pre-configured base VM image to his VM as the OS disk, and an empty data disk is created and mounted onto his VM as well.

All these virtual disks are represented as virtual machine image files in our underline runtime VM storage system. The runtime I/O between virtual machine and its virtual disks is tunneled by the TapDisk driver. To avoid network latency and congestion, our distributed file sys-

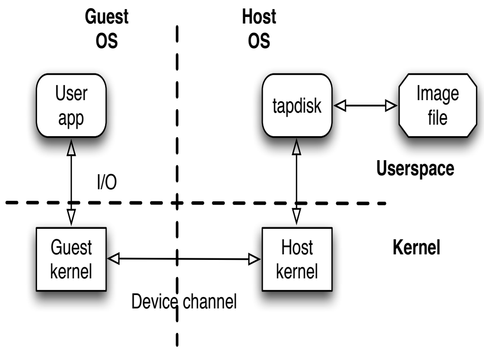

Figure 2: I/O channel between VM and image files

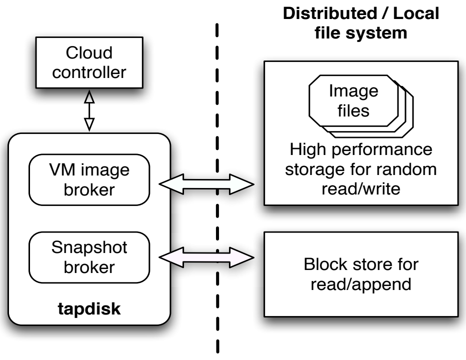
Distributed / Local file system
Cloud controller
Image files
High performance storage for random read/write
VM image broker
Snapshot broker
Block store for read/append
tapdisk

Figure 3: Storage architecture

tem carefully selects the location of primary replica of VM's image files such that they are located on the same physical machine of VM instance.

Inside the TapDisk driver, we maintain an array of *dirty bits* to record the dirty area of runtime VM image file since the last snapshot, each bit represent a 2MB fix-sized data segment. During a snapshot operation, we only need to look at the dirty region rather than scan over the whold image file, which would be extremely slow.

The snapshot storage system also resides in our distributed file system, just like the runtime VM storage. But the difference is that snapshot storage do not need to be co-located with VM instances, in fact they can even live in a different cluster to improve the data safty. When cluster A is offline, we can quickly restore its VMs from cluster B to reduce the impact to our users.

The detail design and implementation of our distributed file system and various storage subsystems would be too complicated to be intorduced here, and also beyond the scope of this paper. In the remaining section we will brief the model and interface of our snapshot storage.

3.2 Snapshot Representation

Each snapshot is a two-level index data structure in the form of HDAG. At the bottom level is variable-sized data blocks, like many other deduplication storage systems, we use 4KB as the average block size. In the middle is the segment layer, a segment contains hundreds of blocks, it records the list of block hash, data reference and some other meta information in a data structure we called segment recipe. Then the segment recipe itself, can be serialized into a data block, so the hash of segment recipes and their meta data formed the snapshot recipe. We take this two-level structure because in practice we observed that at each snapshot only a small fraction of dirty bits array are marked as set, so snapshot recipes can share the unmodified segment recipes to reduce the space cost of snapshot meta data.

Instead of using variables-sized segment like many others did, we take the simpler approach to let every segment being aligned to the boundary of dirty bits array. This is because *vhd* file format will keep the allocated inner blocks at the same positions until deletion, which means the locality of snapshot data is almost natively aligned. Enforcing a boundary at every 2MB will only break 0.2% of total data blocks which is tiny.

3.3 Snapshot Storage

The snapshot storage system sits on top of our distributed file system, it can be abstracted as a block store which operates a few *append-only* files in the underline file storage. Each virtual disk has its own data store, all data blocks after deduplication will be stored into that system.

We let each virtual disk has its own block store rather than sharing for several reasons: first, all data written to block store are already being processed by deduplication process, thus no sharing is necessary unless we want to perform additional deduplication inside the block store. Second, such seperation will facilitate VM data stastics, deletion and migration. Finally, this reduces the complexity of concurrent snapshot operations.

The main interface of our snapshot storage is *get*, *put* and *delete*. Get interface accepts a piece of data, write it to the underline data file, and return a reference to the caller. This reference then can be used in the put interface to retrive or delete the data, thus the caller of put interface must preserve the data reference for future use. Since all the underline data structures is append only, upon a delete request, the corresponding data will only be marked rather than being deleted. A compaction will take place when deleted data has accumulated to certain threshold, thus reclaiming the disk space .

In addition to above data access interfaces, the snapshot storage also supports *scan* and *quota* methods. Scan allows us to traverse all the data blocks that store in the data store, and quota is used to acknowledge user how much space he has actually used.

3.4 CDS Operations

The CDS is divided into CDS meta and CDS data. THe structure of CDS meta is not different from the segment recipe we discussed above, except that it is partitioned into many small slices so that each node is easy to load its own slice. At the runtime, our CDS meta resides in a global shared memory cache for deduplication lookups. The data of CDS is stored in a special block store, which doesn't belong to any VM disk.

CDS meta assignment is done statically, each slice of CDS meta has one primary and two backup nodes. During the system bootup, snapshot storage controller will read the assignment, load its primary portion of CDS meta from distributed file system. Loading the backup portion of CDS is due upon a query arrives for it. Everyday the CDS assignment and CDS data will be checked for updates and reloaded if necessary.

Behind the scene there is a offline map-reduce job, which will periodically scan the block stores in the entire cluster and perform a "word counting". High frequency blocks are then added to the CDS if its count is greater than threshold, which is pre-defined by the system memory restrictions.

3.5 Dedup Process

Our deduplication process can be summarized as four steps:

Dedup Dirty bits: Copy the unchanged portion of snapshot recipe from parent snapshot, base on dirty bits.

Dedup Locality: Divide dirty segments into blocks, compare to the corresponding segment recipe at the same position of parent snapshot, if duplicate, copy the data reference into segment recipe.

Dedup CDS: Check if this data exist in the CDS, if yes, copy the returned data reference. Otherwise, write data block to block store, save the returned reference into segment recipe.

Dedup Save meta: write down the segment recipes and snapshot recipe when finish.

4 Results

In our snapshot deduplication architecture, CDS is the key to achieve greater deduplication than incremental backup solutions. Our basic assumption of CDS us that VM disks, especially OS disks, have huge amount of data in common, and such common data can be represented by a relatively smaller data set because of their high appearence frequency. As a result, the major portion of snapshot deduplication effect shall emerge from

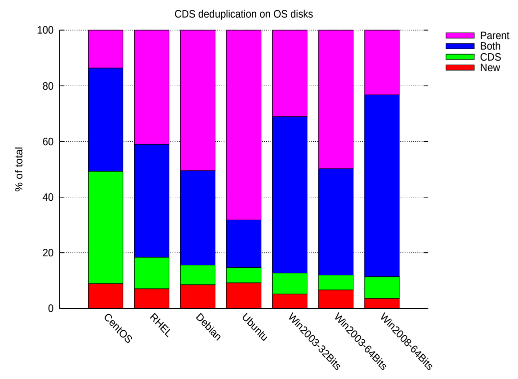

Figure 4: CDS deduplication effect on OS disks

eliminating the duplication of such a small data set. In this section, we evaluate the effectiveness of CDS using real user VM disks from our production VM cluster.

4.1 Experiment Setup

The data set we use is the same as described in previous section. We choose extreme binning and perfect deduplication to compare against. In all experiments, our deduplication enforces 2MB fix-sized segment boundary, and uses TTTD algorithm to divide segment into 4KB variable-sized blocks. For perfect deduplication and extreme binning, the whole snapshots are splitted using TTTD with 4KB average size. The original extreme binning paper uses whole file as the input unit, but that is way too big for our system. Thus we split image snapshot files into variable-sized segments base on the block hash list, using TTTD with average size of 2MB.

4.2 OS Disk

We extract the CDS of OS disks by counting the blocks that will appear in a VM's block store if no CDS is involved in the deduplication process. The threshold is set to less than 1.5%, which is quite sufficient to include the OS related data since their duplication is much heavier than others. Then we use this CDS to run the deduplication process again. Finally we extracted about 80GB of CDS data from 350 OS disk snapshots, the corresponding CDS meta occupies 800MB in CDS cache.

For each block, we tag it with one of the following:

- New: this block cannot be deduplicated and thus write to block store.
- CDS: this block is deduplicated by CDS.
- Parent: this block is not found in CDS, but is found in parent snapshot's segment recipe.
- Both: this block is both found in CDS and parent snapshot.

As we can see from 4, locality dominates. This is because the interval between two snapshots is quite short

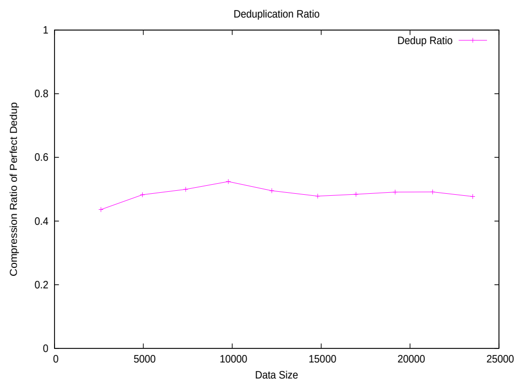

Figure 5: Perfect deduplication on data disks

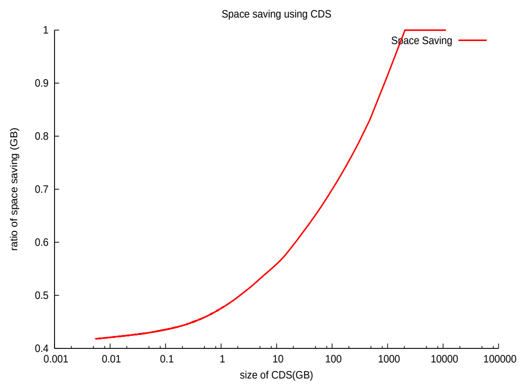

Figure 6: Size of CDS vesus space saving

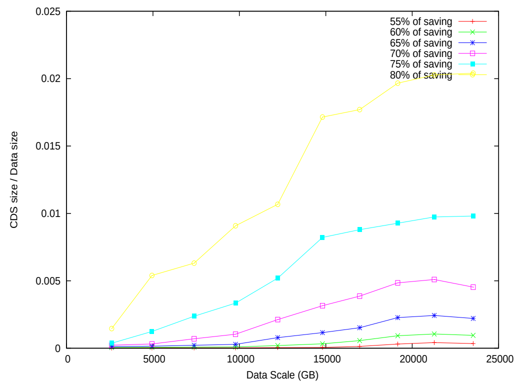

Figure 7: CDS deduplication effect on data disks

due to our daily snapshot strategy. However, locality still doesn't work well on some of the OSes. But CDS, on the contrary, finds a lot of duplicates that locality can't find, especially in a VM's first snapshot.

Combining all the VMs, we see the overall 7.4TB of data is reduced to 512GB. Extreme bining reduces this data set to 542GB, which is slightly worse. As a reference, perfect deduplication achieves 364GB in this experiment.

Overall, none of locality or CDS can solely work well, but by combining them together we get fairly good and stable deduplication ratio to all kind of OSes. If compare to all incremental backup solutions, CDS can save addition 50%+ of disk space because it greatly reduces the cross-VM duplicates.

4.3 Data Disk

Figure 5 shows the compression ratio of perfect deduplication at different data scales. Basically perfect deduplication would help us save 50% of space on user data, regardless of scale. If we put all these unique data into CDS, we could achieve perfect deduplication, which is not affordable. So we need to see how much space saving of perfect deduplication can be achieved through a limit size CDS.

We rank unique data blocks by their duplication count, and choose the hottest blocks as CDS. We define *space saving ratio* as the space saving of CDS divide by perfect deduplication saving. Figure 6 shows the relationship between CDS size and space saving. Its clear a very small amount of CDS data provides more than 50% saving. But this effect decreases when more data are added to CDS. The lower bound of CDS space saving ratio is 50%, which is very easy to accomplish.

The upper bound of CDS size is restricted by system memory resource. Figure 7 shows how CDS space saving is affected by the system scale. In this experiment we first set out a goal of space saving ratio, then we watch how much data we need to put into CDS to achieve this goal. From the graph we can see a 75% saving goal lead to a stable ratio between CDS size and data size, which requires 0.01% of data to be put in CDS.

Base on above data we can estimate the size of data CDS and its effect. Currently we prepared 500MB memory per machine to store CDS meta, then it can represent 50GB of data. If we assume each VM has 30GB of user data at runtime, and we host 25 VMs per machine, maintain 10 snapshots per VM, each brings 10% additional modified data. Thus the user data in snapshot system is 1.5TB per machine. So the upper bound of $CDSsize/Datasize = 0.033$, which is sufficient for the 75% saving goal.

Unlike the CDS of OS disks which is mainly composed of OS related data thus highly predictable, data disks is unpredictable because we cannot control what user can put in there. But we still suspect that highly duplicated data in existing data are very likely to be duplicated again. So we randomly pick 50 out of 1322 data disks as the new data, and use the rest as existing data to extract CDS. Using 1.5% as CDS threshold, we see the total 1198GB of new data is reduced by 755.8GB, while perfect deduplication can reduce 1017.4GB. So 74.3% of duplicate blocks are eliminated by pre-trained CDS, which is quite satisfiable.

5 Conclusion

In this paper we present a new parallel deduplication technique for VM cloud. Our technique utilizes the special data characteristic in VM cloud backup system to accommodate very limited resources for data deduplication. By storing the commonly used OS related data, and the hottest user generated data, we greatly reduce the cross-VM data duplication in VM snapshot backups. Experiments show our solution can eliminate the majority of data duplication with a tiny fraction of block hash index store in memory. It does not only saves valuable system resouces in the VM cloud, but also makes deduplication much faster.

References

[1] Dedupe-centric storage. Technical report, Data Domain Corp.

[2] Emc avamar. `http://www.emc.com/products/detail/hardware/avamar-data-store.htm`.

[3] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1 –9, 21-23 2009.

[4] D. Bhagwat, K. Eshghi, and P. Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD '07)*, pages 105–112, August 2007.

[5] A. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, page 21, Washington, DC, USA, 1997. IEEE Computer Society.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[7] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes. Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 123–138, Berkeley, CA, USA, 2007. USENIX Association.

[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*, pages 111–123, 2009.

[9] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001.

[10] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[11] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981. `http://www.xmailserver.org/rabin.pdf`.

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