Effective Snapshot Deduplication in VM Cloud

Wei Zhang

Abstract

Virtualization based cloud computing is the most successful approach of today's cloud platforms. Amazon and many others provide clusters that host hundreds of thousand virtual machines(VM) which are dynamically created by users. In such 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 size, they would cost tremendous amount of storage space without deduplication involved.

Current snapshot deduplication is mainly done through copy-on-write on fixed-size disk blocks. Such solutions are fast but unable to handle the cross VM and inner-block data duplications. 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 all use a small selection of OS and application libraries. Then 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 chunks to dedup against, and we also remove many inner-block duplication using smaller chunking granuarity and locality. Experiment shows our design can achieve high deduplication ratio and throughput without incuring significant overhead to the runtime VM environment.

1 Introduction

File storage is a critical component in cloud infrastructure, because users and applications expect a persistent place to store and share their data. But there can be a significant amount of data redundancy in the storage system because large number of users employ such systems for backups, and file data between different users may overlap especially when user base is huge. For example, in a VM-based cloud, users usualy backup their virtual machine images along with application data to a cloud file storage periodically. Therefore, improving the space efficiency is an important factor in designing a file storage service for cloud.

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[12], search and index documents in large repositories[5], scalable storage systems[8], store and transfer large directory trees efficiently and with high reliability[9].

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[15]. When the fingerprint meets a certain criteria, such as $fmodD = r$ 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[13]. 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 its deduplication throughput.

This problem has been addressed by many previ-

ous studies. Zhu[20] 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[11] 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[4] uses Broder's theorem[6] 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). We now consider the case for file synchronization service designed to service fine-grained low-locality backup workloads. Such a workload consists of newly modified files, instead of large data streams, that arrive in random order from many clients. There is generally no locality between files that arrive in a given window of time. In the absence of locality, deduplication approach designed for D2D backups perform poorly.

In addition, file storage with deduplication allow a data chunk to be shared by many files, this has not been a problem in D2D backup systems because they may use costly hardware components and RAID to reduce the risk of data loss. Unlike D2D backup systems, cloud storage prefers low-cost hardware at large scale to achieve both performance and reliability. Without software replication, data is fragile in the constant presense of disk or system failure.

Our solution, Sloud, is designed for data deduplication in cloud storage systems. Sloud exploits file similarity and only in-file data locality. Like Sparse Indexing, it splits large files into small chunks and coarse-grain segments. Sloud uses segment as the basic unit for storage allocation, thus avoiding the possibility that a set of similar large files may overload part of the system. Then a stastic based signature algorithm is used to cluster similar segments into containers which are located at fixed places by stateless routing. When new segment comes in, Sloud finds the container which is most likely to have similar data, then it store new segment to that container for chunk deduplication. By doing this Sloud trade deduplication space efficiency for performance and simplicity. In addition, Sloud provides software replication and eventual consistency to enhance file availability and integrity. Experiments shows our solution achieves a balance of space efficiency, data reliability and load balancing.

2 Approach

2.1 Similarity Detection

Data locality is important to Bringing deduplication to a distributed environment may break data locality because introduces several new problems. Former studies of centralized deduplication systems showed that spatial locality of data streams must be preserved. This is because the entire chunk hashes are too big to be loaded into memory, thus certain caching mechanism must be applied, and this require proximal chunk IDs to be stored at nearby places. Furthermore, if chunks are not stored on the disk base on spatial locality, file read requests will result in entirely random disk access, since each data chunk is so small, that would become a disaster for any host file system.

2.1.1 Granuarity

Extreme binning uses files as the basic unit for similarity detection, this simplifies the metadata management of distributed storage because it only need to manage the mapping from files to their bins. However, this solution doesn't consider the uncertainty of file sizes: it is possible that many similar big files can be assigned to one container, thus some storage nodes may be overlaoded when others are spare. For example, Amazon's S3 stores huge amount of virtual machine images which are very similar, so those files will be assigned to only a few nodes by the extreme binning. Even those files can be evenly distributed to hundreds of nodes, the overall deduplication ratio won't be good since too much dulicates are tolerated.

We suggest that data segment at the size of several megabytes is the best unit for similarity based deduplication. Sparse Indexing has already proved that content based chunking method can be also applied to break the list of chunks into large segments, such that each segment contains a few hundred of small chunks. Using such medium size segments as the basic unit for similarity detection could eliminate disadvantages mentioned above, and doing so won't add much computation cost to the deduplication process.

2.1.2 Signature Algorithm

In general, we expect to generate a signature for each segments, such that if two segments has a lot of chunks in common, there is a high probablity that their signatures are the same. Since we represent segment as a list of chunks, the information available for generating signatures are the chunk hash values, chunk sizes, and offsets.

Extreme binning suggests the min-hash signature algorithm. By Broder's theorem, the probability that the smallest elements of two sets are identical, is equal to

their Jaccard similarity coefficient. Therefore they use the smallest chunk hash value as the signature to represent the whole file.

We propose a Size-based Similarity Detection (SSD) method, which is more storage specific because it uses the chunk size information. Intuitively if one segment is only slightly modified, then most of the chunks should remain the same, thus their sizes should mostly remain the same. Even modifications could change the chunk hash value, due to the modification resistant property of content based chunking algorithm, most of the time that chunk size is unchanged. Therefore if we want to generate a n bit signature for a segment, we simply split the range of size evenly at logarithmic scale by n , and count how many chunks fall into each range. Then if that number is above average, we set the bit for that range as 1, otherwise 0, as shown in figure.

By using the signature of segment to represent all of its chunks, Sloud clusters segments that are highly similar into one container. Each container is the collection of segments that have the same signature. When a new segment arrives, it will be assigned by its signature to the corresponding container which has chunks that are highly similar to it. Thus, highly accurate deduplcation is achieved. However, since every segment is only assigned to one container, if any of the its chunks do not exist in that container but exist elsewhere, they will be treated as new chunks, in other words, duplicates are allowed. Sloud trade such storage space efficiency for faster deduplication and fewer resources consumption such as RAM usage and disk access. In addition, we believe this is the best way to build distributed deduplication solution without losing data locality.

2.2 Architecture

We designed a distributed storage system, Sloud, to demonstrate the possibility of providing scalibility to large scale deduplication systems. Sloud cluster similar segment into containers and store those containers to the disk. The overall architecture is shown in . Sloud uses a DHT service for node discovery and storage space partitioning, all nodes in Sloud are completely equal, there's no node that has special responsibilities.

At every storage node , we keep in memory a bloom filter and a segment index for every container, the rest of part of the container, including the chunk data and chunk index, are stored onto disk. When a new segment comes in, the segment index is used for the first pass of deduplication, such that duplicate segments can be detected without looking below. The second pass is done by bloom filter, most of new chunks are immediately identified. Otherwise, Sloud look into the chunk index for the final pass of deduplication, as shown in Figure.

Assuming all node are reliable, no failure ever happens, and there is no replication involved, the deduplication process can be described as following:

1. Firstly, sender scan the files that to need be deduplicated, generate all information about chunks and segments.
2. For each segment, sender queries the DHT service to find out the node that is responsible to segment's container, then it will send a segment writting message with segment's signature and hash.
3. The receiver then lookup corresponding container's segment index to see if its a duplicate. If it is, there is no need for further deduplication, it will update the segment's reference count and acknowledge the sender deduplication is complete.
4. If the segment hash does not exist in segment index, receiver asks sender for the segment recipe, which is all the chunks' information except the actual data.
5. On receiving the segment recipe, receiver starts deduplicate them against existing chunks in that container. Firstly it goes to the bloom filter, most of the new chunks are identified in this step. For those chunks that are identified as 'exist' by the bloom filter, the receiver looks into chunk index to see if they are really there or not. At any step, if a chunk is identified as new, a request of chunk data is imdeiately sent out.
6. Incoming chunk data are written to a temporary disk place until all new chunks belonging to that segment are received. Upon this time, receiver merges new data into that container's disk file.
7. Finally the receiver update the segment index and chunk index, and acknowledge the sender this segment is sucessfully received and deduplicated.

2.3 Replication and Concistency

Sloud uses consistent hashing ring[10] to maintain a dynamic hash table for node discovery. Each node is given an unique ID and the Sha-1 checksum of their IDs are mapped onto this ring. Containers are also mapped onto the consistent hash ring by the Sha-1 checksum of their represented signatures. The benefit of using consistent hash is that adding or removing nodes from the system won't introduce too much workload of reorganization. For each container on the ring, Sloud treat the first three consequential nodes as the primary replicas of that container, and the next two nodes as the secondary replicas. Primary replicas hold the real copy of container's data, the secondary replicas are only used to cache write requests when some of the primary replicas are temporarily unavailable.

Sloud provides a variation of eventual consistency[17] in its data replication. Read requests will succeed if at

least one of the three primary replicas is available, write requests requires the majority of five primary and secondary replica nodes to be available, which means one of the primary replicas must be available for the write request. During a segment write, sender finds out the first available node in three of primary replicas, say node A, and let this node handle the rest of deduplication and replication process. Node A will first deduplicate the new segment as described in previous sections, then it tries to acknowledge other two primary replicas about this update. If succeed, then this segment is successfully written. Otherwise it will try to write this update to secondary replicas in order to make sure the new data is written to three places. Upon a successful writting to both primary and secondary replicas with three copies, node A can tell the sender the new segment is added to storage. But if the sender can not find a available primary replica node, or node A can not write to the other two replicas, then the write request will fail.

When secondary replicas receive a segment write request, it will only cache the data because they do not have that container. They will try to send cached requests to primary replicas when possible. However, if two secdonary replicas want to acknowledge a recently recovered primary replica about the lastest updates, the recovered node will receive duplicate requests. Furthermore, it is possible that multiple senders want to write identical segment simultaneously, so it is necessary to use an ID to distinguish every unique segment writting process. In Sloud this segment process ID (SPID) is composed of sender's node ID and an incremental logical clock. The SPID is used through out that segment's deduplication process and carried by all related messages. In order to avoid processing the same request twice, for every segment Sloud keeps its recent SPIDs in the segment index for a short period, therefore if the SPID of incoming segment writting request is found in segment index, receiver could know it has already received and deduplicated this segment.

Periodically, Sloud synchronize each container among its replicas. Synchronization requires all primary and secondary replicas to be available. During this process, primary replicas compare each others' recent SPIDs along with secondary replica's cache to determine which write request it has missed. Upon a successful synchronization, all cached segment updates are ensured to be delivered to three primary replicas, thus the secondary replicas' cachecan be cleaned up. In addition, recent SPIDs in segment index is also stale, they will be cleaned up so that Sloud do not need to keep too many recent SPIDs in segment index. If any node is not available during the synchronization, this process will fail. However, the primary replicas still try to synchronize as much data as possible, except that the cache and SPIDs will not be cleaned up.

3 Related Works

Several approaches have been previously proposed to enable efficient deduplication in D2D backup.

DDFS[20] exploits chunk locality to achieve high-throughput perfect deduplication. It preserves locality by a Stream-Informed Segment Layout and exploits locality with Locality Preserved Cache. An in-memory Bloom Filter is also used to accelerate non-duplicate chunk identification.

Sparse Indexing[11] is an approximate deduplication technique designed for D2D backup. It divides data stream into variable-sized multiple kilobytes chunks, and construct multiple megabytes segments using the same chunking technique, which are then sampled and mapped to a compact in-memory sparse index. Incoming segments are only deduplicated against several existing similar segments selected according to the sparse index.

Both DDFS and Sparse Indexing are designed for D2D backup workloads, and do not address the scalability issue in a distributed environment. A few scalable deduplication approaches have been proposed recently.

Extreme Binning[4] is a scalable parallel deduplication approach that targets at non-traditional backup workloads that consist of low-locality individual files. It groups highly similar files into bins, and eliminates duplicate chunks inside each bin. Duplicate chunks are allowed to exist among different bins, resulting in approximate deduplication. By keeping only the primary index in memory, Extreme Binning can reduce the RAM requirement while maintaining a reasonably high throughput. However, their per-file based similarity detection is going to group all similar files into one node, which will break load balancing if some files are huge and similar(e.g., virtual machine images).

MAD2[18] is a scalable high-throughput exact duplication approach. MAD2 utilizes on-disk Hash Bucket Matrix to preserve fingerprint locality and integrates in-memory Dual Cache to capture and exploit locality. In addition, MAD2 employs Bloom Filter Array to efficiently identify unique incoming fingerprints and indicate where a duplicate may reside. By employing a DHT-based Load-Balance technique to distribute file recipes and chunk contents among multiple storage nodes in their backup sequences, MAD2 further enhances performance with a well balanced load. However, the data locality does not exist for cloud storage because only changed chunks are expected to be uploaded, and the heavy usage of memory and CPU indicates such exact deduplication backup systems need storage-exclusive servers with hardware replication support, which is not a general case for the cloud..

HYDRAstor[8], a scalable secondary storage solution, constructs its backend using a grid of storage nodes built around a distributed hash table. The backend maintains large-scale variable-sized, content-addressed, immutable, and highly-resilient data blocks that are logically organized in a directed acyclic graph. Duplicate chunks are eliminated according to their hashes. HYDRAstor adopts an average chunk size of 64KB, among other constraints, to keep all the metadata in memory and avoid the duplicate-lookup disk bottleneck. This degrades the space efficiency of deduplication, and still requires huge amount of memory.

We believe a deduplication backend of cloud storage must have scalability and high availability built in mind, being able to run on low-cost non-proprietary machines. Unlike sloud, all above systems lack one or a few such properties. All exact deduplication approaches are too costly, this is why we choose similarity based approach to trade deduplication accuracy for speed.

Eariler deduplication systems mainly focus on improving storage space efficiency by eliminating duplicates at the file level, fixed-size block level, or variable-sized chunk level. EMC's Centera[3] identify and eliminate duplicate data by comparing the hash of the whole file or fixed content. Venti[14], a block-level archival storage, removes redundant fixed-size data blocks by comparing their secure hashes. Pastiche[7] utilizes chunk-level duplicate detection to construct a resource-saving peer-to-peer backup network. Deep Store[19], a large scale archival storage system, uses both variable-sized chunk-level deduplication and delta compression to save storage. Jumbo Store[9] organizes variable-sized chunks into Hash-Based Directed Acyclic Graphs to save both storage and bandwidth while performing incremental upload and versioning for a utility rendering service.

Duplicate detection technique has also been used in bandwidth-saving synchronization protocols[16] and low-bandwidth network file systems[12].

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] Emc centera. `http://www.emc.com/products/family/emc-centera-family.htm`.

[4] 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.

[5] 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.

[6] 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.

[7] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[8] 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.

[9] 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.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[11] 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.

[12] 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.

[13] 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.

[14] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[15] 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`.

[16] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, February 1999.

[17] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[18] J. Wei, H. Jiang, K. Zhou, and D. Feng. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –14, May 2010.

[19] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 804–8015, Washington, DC, USA, 2005. IEEE Computer Society.

[20] 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.