An In-Cloud Backup System With Deduplication for VM Snapshot

ABSTRACT

This paper proposes a VM snapshot storage architecture which adopts multiple-level selective deduplication to bring the benefits of fine-grained data reduction into cloud backup storage systems. In this work, we describe our working snapshot system implementation, and provide early performance measurements for both deduplication impact and snapshot operations.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

We've found a gold mine!

2. DESIGN OVERVIEW

In this section we briefly describe all the static things in our system, such like architecture, each componentâĂŹs data structure, data locations, relationship of data references, etc.

We discuss the characteristics and main requirements for VM snapshot backup in a cloud environment. which are different from a traditional data backup.

1. *Cost consciousness.* There are tens of thousands of VMs running on a large-scale cluster. The amount of data is so huge such that backup cost must be controlled carefully. On the other hand, the computing resources allocated for snapshot service is very limited because VM performance has higher priority. At Aliyun, it is required that while CPU and disk usage should be small or modest during backup time, the memory footprint of snapshot service should not exceed 500MB at each node.
2. *Fast backup speed.* Often a cloud has a few hours of light workload each day (e.g. midnight), which creates an small window for automatic backup. Thus it is desirable that backup for all nodes can be conducted in parallel and any centralized or cross-machine communication for deduplication should not become a bottleneck.
3. *Fault tolerance.* The addition of data deduplication should not decrease the degree of fault tolerance. It's not desirable that small scale of data failure affects the backup of many VMs.

There are multiple choices in designing a backup architecture for VM snapshots. We discuss the following design options with a consideration on their strengths and weakness.

1. *An external and dedicated backup storage system.* In this architecture setting, a separate backup storage system using the standard backup and deduplication techniques can be deployed [4, 1, 2]. This system is attached to the cloud network and every machine can periodically transfer snapshot data to the attached backup system. A key weakness of this approach is communication bottleneck between a large number of machines in a cloud to this centralized service. Another weakness is that the cost of allocating separate resource for dedicated backup can be expensive. Since most of backup data is not used eventually, CPU and memory resource in such a backup cluster may not be fully utilized.
2. *A decentralized and co-hosted backup system with full deduplication.* In this option, the backup system runs on an existing set of cluster machines. The disadvantage is that even such a backup service may only use a fraction of the existing disk storage, fingerprint-based search does require a significant amount of memory for fingerprint lookup of searching duplicates. This competes memory resource with the existing VMs.

   Even approximation [1, 2] can be used to reduce memory requirement, one key weakness the hasn't been

addressed by previous solutions is that global content sharing affects fault isolation. Because a content chunk is compared with a content signature collected from other users, this artificially creates data dependency among different VM users. In large scale cloud, node failures happen at daily basis, the loss of a shared block can affect many users whose snapshots share this data block. Without any control of such data sharing, we can only increase replication for global dataset to enhance the availability, but this incurs significantly more cost.

With these considerations in mind, we propose a decentralized backup architecture with multi-level and selective deduplication. This service is hosted in the existing set of machines and resource usage is controlled with a minimal impact to the existing applications. The deduplication process is first conducted among snapshots within each VM and then is conducted across VMs. Given the concern that searching duplicates across VMs is a global feature which can affect parallel performance and complicate failure management, we only eliminate the duplication of a small but popular data set while still maintaining a cost-effective deduplication ratio. For this purpose, we exploit the data characteristics of snapshots and collect most popular data. Data sharing across VMs is limited within this small data set such that adding replicas for it could enhance fault tolerance.

Overall, our contributions are:

1. *CDS. parallelism, localized* We designed a highly efficient global deduplication scheme using common data between VMs as the primary deduplication target, which adopts to our resource-constrained production environment very well. We developed a fast approxmation algorithm to extract the commonly used data from VM snapshot backups.

2. *Append Store.* We address the problem of managing billions of small variable-sized data objects by building an append store on top of our distributed file system (DFS). We carefully designed an efficient index data structure to optimize large sequential read/write requests, while minimize the cost of key to object location conversion.

3. *Fast Data Deletion.* Traditional deletion in deduplication systems involves reference counting, either online or offline. Both ways are costly due to the sharing of large amount of data objects. We avoid this cost of data deletion with an approxmation deletion method using bloom filters.

4. *Fault-tolerant.* Golbal deduplication brings data sharing across different users VM snapshots, therefore a single failure on some data can break many VMs' snapshots. We avoid such all-to-all data dependency by restricting the cross VM data sharing with the CDS. By taking extra care of protecting CDS data, the overall degree of fault tolerancy is guaranteed.

2.1 Design Considerations

While this idea sounds conceptually simple, realizing it requires us to address three significant challenges:

1. *In-cloud backup* We decided to build

2. *Inline deduplication* Inline deduplication requires more resources and can suffer from latency issues as data is checked against metadata before being committed to disk or flagged as a duplicate.

3. *Data dependency*

4. *Backup speed*

2.2 Architecture Overview

Our architecture is built on the Aliyun platform which provides the largest public VM cloud in China based on Xen [**?**]. A typical VM cluster in our cloud environment consists of from hundreds to thousands of physical machines, each of which can host tens of VMs.

A GFS-like distributed file system holds the responsibility of managing physical disk storage in the cloud. All data needed for VM services, which include runtime VM disk images and snapshot backup data, reside in this distributed file system. During the VM creation, a user chooses her flavor of OS distribution and the cloud system copies the corresponding pre-configured base VM image to her VM as the OS disk, and an empty data disk is created and mounted onto her 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 virtual device driver (called TapDisk[3] at Xen). To avoid network latency and congestion, our distributed file system place the primary replica of VM's image files at physical machine of VM instance. During snapshot backup, concurrent disk write is logged to ensure a consistent snapshot version is captured.

Figure 1 shows the architecture view of our snapshot service at each node. The snapshot broker provides the functional interface for snapshot backup, access, and deletion. The inner-VM deduplication is conducted by the broker to access meta data in the snapshot data store and we discuss this in details in Section **??**. The cross-VM deduplication is conducted by the broker to access a common data set (CDS) (will discuss in Section **??**, whose block hash index is stored in a distributed memory cache. The snapshot store supports data access operations such as *get*, *put* and *delete*. Other operations include data block traverse and resource usage report. The snapshot data does not need to be co-located with VM instances, and in fact they can even live in a different cluster to improve the data reliability: when one cluster is not available, we are still able to restore its VMs from another cluster which holds its snapshot data.

Under the hood of snapshot store, it organizes and operates snapshot data in the distributed file system. We let each virtual disk has its own snapshot store, and no data is shared between any two snapshot stores, thus achieve great fault isolation. For those selected popular data that shared by many VM snapshot stores, we could easily increase its availability by having more replications.

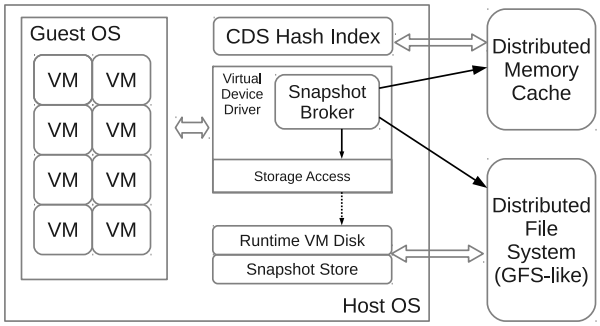

Figure 1: Snapshot backup architecture of each node.

2.3 Block Storage Device

2.4 CDS

Although locality based data reduction can remove most of the inner-VM duplications, it's not able to solve the data duplication cross VMs. Different VMs still share large amount of common data such that their snapshot backups would have a lot of data in common. Our observations on Aliyun's real VM user data reveals several major sources of cross-VM data duplication:

1. *System-related data*: These data are generated by public third-parties, they are copied/downloaded/installed into VM disks either automatically or manually. Once installed, operations on such data are mostly read only until software updates arrive. For example, data of operating systems, some widely-used software such as Apache and MySQL, and their documations all fall into this category.

2. *User-generated data*: These data are generated by individual user's behavior, either directly or indirectly. They are much less duplicated than the system-related data, but the zipf-like distribution indicates that a small amount of data in this category could represent most of data duplications.

3. *Zero-filled data*: Like previous studies [refs here], we've observed that zero-filled data exist pervasively at system wide. They are almost like the spaces and commas in text articles. Under content-based chunking, they were distilled as zero-filled blocks with the maximum allowed length.

Without eliminating the data duplication between VM snapshot backups, storage space is severely wasted when the number of VMs increases. To address this problem, we developed a technique called *Common Data Set (CDS)* to eliminate data duplication for all three situations above. CDS is a public data library that shared by all VM snapshot backups in the cluster, which consists of the most popular data blocks in VM snapshot backups. It is generated, managed and accessed in an uniform manner by all VMs such that [write some fancy system characteristic stuff here].

2.5 CDS Size V.S. Coverage

As a shared data library, we expect CDS to collect almost all the system related data, the most popular user-generated data and the zero-filled data block. With CDS as a public data reference, each VM's snapshot backup has no need to store its own copies for data that can be found in CDS, instead they can just store a reference.

The more data we put into CDS, the closer we come to complete deduplication. But in reality we are facing a list of restrictions such like [blabla]. We borrowed the idea of web caching[refs here] to analysis the size of CDS vesus its effectiveness on data reduction.

We let M be the number of nodes in a cluster, N be the number of VMs hosted per node, and let D denotes the average size of one VM's snapshot backups (after level 1 and 2 reduction, which is near 40 GB in our production environment). And let C_0, C_s and C_u denote the size of zero-filled block, system-related data and user-generated data in CDS, then the total size of CDS is represented by $C = C_0 + C_s + C_u$. We let S_0, S_s and S_u denote the corresponding data coverage ratio (with respect to D) by C_0, C_s and C_u, so the total space saving ratio is $S = S_0 + S_s + S_u$.

Zero-filled block at maximum length is the first item we need to put into CDS. This is the one and only special data block, and because CDS only stores unique data blocks, C_0 is almost equal to 0. In practice we found zero-filled blocks account for near 20% of total data, so S_0 is 20%.

The rarely modified system-related data are the second to be added to CDS. We have thousands of VMs in each cluster, but all these VMs fall into only a few OS types, using a limited selection of software, therefore data duplication in this category is highly noticable in a global block counting. In particular, most of such data already exist in the VM base images. We analyzed VMs running various services from 7 mainstream OS types in our cluster, and found close to 50GB of such data in total. Because system-related data don't change frequently, it's safe to expect that for 20 OS variations and software updates in 2 years, C_s will not grow to exceed 200 GB. For each VM, from 2 to 20 GB of data can be reduced in this way, depends on the OS and service type, so on average we estimate S_s would be no less than 20%.

The rest of data are user-generated, the total size of such data can be written as $D_u = D - S_0 - S_s$, which is 60% of D. It follows a zipf-like distribution with α between 0.65 to 0.7. Let T_u be the total size of unique data blocks in D_u, in practice we notice that complete deduplication for such data always result in a 2:1 reduction ratio, , so $T_u = D_u/2$. Since we collect the most popular user-generated data into CDS, the coverage of CDS on user-generated data is $(C_u/T_u)^{1-\alpha}$.

The following table lists CDS coverage on user-generated data under different α and C_u/T_u.

[a table of coverage with alpha from 0.65-0.70, ratio from 0.002 0.005 0.01 0.02 0.05]

It's obviously that at least 30% of user-generated data can

be reduced in this way, using about 0.02 of T_u, or 0.01 of D_u. The size of user-generated data in CDS would be $0.006D$, with coverage $S_u = 0.18D$.

Thus the estimation of CDS coverage is $S = S_0 + S_s + S_u = 0.58$, with the size of CDS to be no more than $(200 + 0.006 * D * N)$ GB. In a VM cluster of 100 machines, with 25 VMs on each physical machine, the size of CDS is 800 GB in total, or 8 GB per machine. The total size of CDS index would be 8 GB, which will cost 80 MB memory on each machine.

2.6 Append Store
3. SNAPSHOT OPERATIONS
4. APPEND STORE
4.1 Introduction

Append Store (AS) is our underlining storage engine for storing snapshot data after deduplication. AS is built on top of our highly scalable distributed file system (DFS), which is very similar to Google's file system in a sense that it is optimized for large files and sequential read/append operations.

4.2 Design considerations

While scalability has been handled by the DFS, there are still several challenges in storing billions of var-sized small data chunks:

Locality perseveration Chunk data must be placed next to each other in the order of their writing sequence, because reading a snapshot incurs large sequential reads of chunks under their logical sequence in the snapshot.

Flexible referencing To avoid overwhelming metadata operation capbility of DFS's master, we have to group small chunks into large data files in DFS. However, snapshot deletion brings chunk data deletion, and reclaiming disk space would require many chunk data being moved. If such chunk movement incurs updates of data referencing in snapshot and segment recipes, then the cost of deleting a snapshot would be extremely high.

Efficient chunk lookup The translation from data reference to data location must be very efficient in two ways: first, a sorted index is necessary to fast locate the data location in $O(log(n)$ time; second, the size of index must be optimized to minimized its memory footprint and the cost of fetching index.

4.3 Architecture

Append Store supplies three interfaces: *get(ref)* accepts a data reference and retrives data, *put(data)* accepts data and returns a reference to be stored in metadata recipes, *delete(ref)* deletes the data pointed by the reference. Under the hood, small var-sized data are grouped and stored into larger data containers. Each VM has its snapshot data stored in its own Append Store, specified by the VM ID. We split every Append Store into multiple data containers so that reclaiming the disk space would not result in rewriting all the data at the same time.

As shown in Fig.2, every data container is represented as three data files in DFS: the data file holds all the actual data,

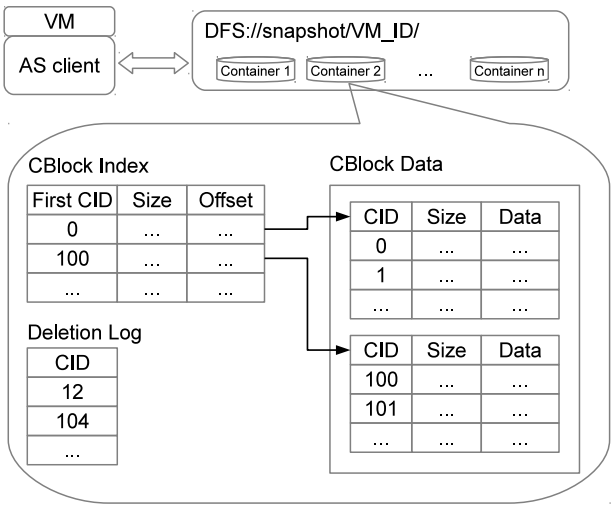

Figure 2: Architecture of Append Store

the index file is responsible for translating data reference into data locations, and a deletion log file remembers all the deletion requests to the container.

A data reference is composed of two parts: a container ID (2 bytes) and CID (6 bytes). Append Store assign every piece of data a CID for its internal data referencing. When new data is appended, its CID is the current largest CID in that container plus one. As a result, all the data locations are naturally indexed by this self-incremental CID, no extra sorting is needed.

Append Store groups multiple chunk data (i.e., 100) into larger units, called *CBlock*. CBlock is the basic unit for append store's internal read/write/compression. There is one index entry in the container index corresponding to every CBlock. It keeps the first chunk's CID in that CBlock, and the CBlock data's size and location.

Using CBlock brings us several advantages: First, the write workload to DFS master is greatly reduced; second, grouping small chunks gives better compression. Third, reading a CBlock (200 - 600 KB) typically cost the same amount of disk seek as reading a 4KB chunk. Finally, this greatly reduces the size of index. Let m be the number of chunks in each CBlock, then the overall index size is reduced to $1/m$. In our implementation, using $m = 100$ reduces the index for a 1GB container from 10 MB to 100 KB.

4.4 Operations

In order to read a chunk data by reference, Append Store client first loads the container index file specified by the container ID, then search the CBlock index to find the entry that covers the chunk by CID. After that, it reads the whole CBlock data from DFS, decompress it, seek the exact chunk data specified by CID. Finally, the client updates itws internal chunk data cache with the newly loaded contents to anticipate future sequential reads.

Symbol	Description	Measurement
N_{seg}	Number of segments in the snapshot	
N_{chunk}	Number of chunks in one segment	
P_{dirty}	Probablity of a segment is dirty	
P_{new}	Percentage of new data	
P_{change}	Probablity that a chunk is not found in parent snapshot	
P_{in_CDS}	Percentage of chunks in CDS	
T_{scan}	Time to scan segment data and caculate content hash	74 ms
T_{write_AS}	Time of writing a chunk into append store	0.24 ms
T_{read_AS}	Time of reading a chunk from append store	
T_{read_chunk}	Time of reading a chunk	
T_{check_CDS}	Time of checking CDS	
T_{read_CDS}	Time of reading a chunk data from CDS data store	

Table 1: List of performance factors

Write requests to append store are accumlated. When the number reaches m, the AS client forms a CBlock by assigning every chunk a CID, compress the CBlock data, and append it to the CBlock data file. Then a new CBlock index entry is appended to CBlock index.

Append store adopts lazy delete strategy. The deletion requests are appended into every container's deletion log file with the CID of data to be deleted. CIDs in deletion log are guaranteed to be referenced by nobody and can be safe deleted in future. Periodically, snapshot management system asks append store to compact containers in order to reclaim disk space. The actual compaction will only take place when the number of deleted items reached $d\%$ of container's capacity. During compaction, append store creates a new container (with the same container ID) to replace the existing one. This is done by sequentially scan the old container, copying all the chunks that are not found in deletion log to the new container, creating new CBlocks and indices. However, every chunk's CID is plainly copied rather than re-generated. This does not affect the sorted order of CIDs in new container, but just leaving holes in CID values. As the result, all data references stored in upper level recipes are unaffected, and the data reading process is as efficient as before.

4.5 Analysis

Snapshot operation performance analysis:

The overall time of writing a snapshot is:

$$T_{write} = P_{dirty} * N_{seg} * [max(T_{scan}, T_{read\_AS}) + P_{change} * N_{chunk} * T_{check\_CDS} + P_{new} * N_{chunk} * T_{write\_AS}] \quad (1)$$

The overall time to read a snapshot is:

$$T_{read} = N_{seg} * T_{read\_AS} + N_{chunks} * [P_{in\_CDS} * T_{read_CDS} + (1 - P_{in\_CDS}) * T_{read\_chu} \quad (2)$$

Average time of reading a chunk:

$$T_{read\_chunk} = P_{in\_CDS} * T_{read\_CDS} + (1 - P_{in\_CDS}) * T_{read\_AS} \quad (3)$$

Time of reading append store:

$$T_{read\_AS} = 0 * P_{in\_cache} + (1 - P_{in\_cache}) * T_{load\_CBlock} \quad (4)$$

5. DELETION

In a mature VM cluster, snapshot deletions are as frequent as snapshot creations. Our system adopts lazy delete strategy so that all snapshot deletions are scheduled in the backup time window at midnight. Therefore, snapshot deletions must be fast enough to fit in time window and efficient enough to satisfy our resource constraints. However, there is no simple solution can achieve these goals with high reliability. In Aliyun's cloud, we use a *fuzzy deletion* method to trade deletion accuracy for speed and resource usage. This method tolerates a tiny percentage of storage leakage to make the deletion operation faster and efficient by an order of magnitude, yet we still adopt an slow but accurate deletion method to fix such leakage in the long-term. Our hybrid deletion strategy, using fuzzy deletion regularly and accurate deletion periodically, accomplishs our speed, resource usage and relibility goals very well.

5.1 Challenges

Contrary to a traditional backup system, a dedupe system shares data among files by default. Reference management is necessary to keep track of chunk usage and reclaim freed space. In addition to scalability and speed, reliability is another challenge for reference management. If a chunk gets freed while it is still referenced by snapshots, data loss occurs and files cannot be restored. On the other hand, if a chunk is referenced when it is actually no longer in use, it causes storage leakage.

Reference counting, while being simple, suffers from low reliability especially in our distributed environment, because it is vulnerable to lost or repeated updates: when errors occur some chunks maybe updated and some may not. Complicated transaction rollback logic is required to make reference counts consistent. Moreover, there is almost no way to verify if the reference count is correct or not in a large dynamic

system. In real deployments, where data integrity and recoverability directly affect product reputation, simple reference counting is unsatisfactory.

Mark-and-sweep is generally a better solution. During the mark phase, all snapshot recipes are traversed so as to mark the used chunks. In the sweep phase all chunks are swept and unmarked chunks are reclaimed. This approach is very resilient to errors: at any time the process can simply be restarted with no negative side effects. Scalability, however, is an issue. *needs to explain its resource usage*

5.2 Accurate Deletion

Our accurate deletion uses mark-and-sweep to find all the unused chunks in a VM's append store. The following steps would take place during an accurate deletion:

1. Scan all the containers. For each container, extract full list of existing data references and allocate a bitmap. (Notice this list is self sorted.)
2. Scan all the snapshot recipes and segment recipes. For every data reference pointing to append store, find it in the above lists and mark the corresponding position in bitmaps.
3. Scan the bitmaps, for every bit that are not marked, tell append store the corresponding data reference can be deleted.

Although accurate deletion guarantees reliability and correctness, its speed and resource usage are big problems to our VM cloud. Let T_{scan_AS} be the time to scan the VM's append store, $T_{scan\_recipes}$ be the time to scan one snapshot recipe and its segment recipes, T_{mark} be the time of finding a data reference in the list and mark bitmap, N_{SS} be the number of snapshots and N_c be the number of chunks in one snapshot. The overall time of running an accurate deletion for one VM is:

$$T = T_{scan\_AS} + N_{SS} * T_{scan\_recipes} + N_{SS} * N_c * T_{mark} \quad (5)$$

For a virtual disk that has 10 snapshots, 50 GB of data in the append store, T_{scan_AS} would have cost half an hour already. In addition, scanning recipes of 10 snapshots cost another 5 minutes. Furthermore, maintaining the full lists of data references and bitmaps costs about 100 MB of memory. Having tens of VMs co-lived in one physical machine and deleting snapshot on a daily basis, it's infeasible to run accurate deletion as frequent as snapshot deletions.

5.3 Fuzzy Deletion

We design fuzzy deletion to fast identify unused data in the append store. Instead of scanning the whold append store, it uses a bloom filter to check if a data are still referenced by living snapshots. However, there is a small false-positive probility which would indetify unused data as in use, i.e., storage leakage. The following steps would take place during an fuzzy deletion:

1. **Creating bloom filter** Scan all the living snapshot recipess and their segment recipes, for every reference pointing to append store, add it to the bloom filter.
2. **Check existance** For every data reference in the deleted snapshot recipe and its segment recipes, check the existance of that data reference in bloom filter. If not found, it is safe to delete that piece of data from append store because no living snapshots has referenced it.

The overall time of running a fuzzy deletion for one snapshot deletion would be scanning all the living snapshots and deleted snapshots, since operations on the in-memory bloom filter can be done in parallel and is much faster than loading recipes from DFS:

$$T = (N_{SS} + 1) * T_{scan\_recipes} \quad (6)$$

Using the example and analysis in previous section, this fuzzy deletion can be done in 5 minutes. Memory usage of the bloom filter depends on its false-positive probility P_{bl}, when set P_{bl} to 0.01, the memory footprint of fuzzy deletion is about 15 MB.

The design goal of fuzzy deletion is to reduce the frequency of running accurate deletion. For one VM's snapshots, let $D_{leakage}$ be the amount of storage leakage, D_{del} be the average amount of data to be deleted in one snapshot deletion, then we have:

$$D_{leakage} = N_{fuzzy} * P_{bl} * D_{del} \quad (7)$$

where N_{fuzzy} is the number of runs of fuzzy deletion. An accurate deletion is triggered to fix the storage leakage when $D_{leakage}/D_{del}$ is accmulated to exceed certain threshold T:

$$D_{leakage}/D_{del} = N_{fuzzy} * P_{bl} > T \Rightarrow N_{fuzzy} > T/P_{bl} \quad (8)$$

Therefore, when $P_{bl} = 0.01$ and $T = 1$, there would be a run of accurate deletion for every $T/P_{bl} = 100$ runs of fuzzy deletion. On a machine that hosts 20 VMs and each VM deletes snapshot daily, there would be less than one accurate deletion scheduled per day.

5.4 Discussion

When a snapshot is scheduled for deletion, we always choose fuzzy deletion.

6. FAULT TOLERANT
7. EXPERIMENTS
8. CONCLUSION
9. REFERENCES

[1] D. Bhagwat, K. Eshghi, D. D. E. 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, 2009.

[2] 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.

[3] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. page 22, Apr. 2005.

[4] 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.