DBLK: Deduplication for Primary Block Storage

Yoshihiro Tsuchiya, Takashi Watanabe
Fujitsu Limited
Kanagawa, Japan
{tsuchiya.yoshi,wtnb}@jp.fujitsu.com

***Abstract*—The deduplication block-device (DBLK) is a deduplication and compression system with a block device interface. It is used as a primary storage and block-wise deduplication is done inline. Since deduplication for primary storage requires low latency and detecting block-wise deduplication creates a large amount of metadata, it is necessary to efficiently use the memory of the system. We solved this problem by developing a multilayer Bloom filter (MBF) to reduce the size of the data structuren in the memory for indexing duplicate data.**

Keywords-component; storage system, deduplication

I. INTRODUCTION

Data deduplication is a technique for reducing the size of data in storage systems by removing redundant data, which reduces storage-system costs. Deduplication systems split data into small chunks and compare them to the existing data sets in the system. If incoming data matches the existing data, they are duplicates and the system does not record the data, but instead, records the number of reference of the data.

In a *backup* system, there are duplicated data because two copies of full-backups are almost identical. In a file server, i.e. a *primary* storage system, email and documents are sources of duplication. Operating system images of virtualization systems, such as VMware and Xen, also create redundant data. Both backups and VMs are attractive for applying deduplication technology because of their high rate of redundancy.

Strategies for finding duplicates in backup and primary storage systems are different. A backup system obtains a large backup stream and uses content-defined chunking by Rabin fingerprinting [19] against it. Since daily backup streams are similar to each other, the access pattern is predictable. Zhu et al. [22] showed how to optimally access their metadata. On the other hand, the access size for primary storage systems may be as small as a file system block size and access patterns may be random. To obtain low latency for random access, systems must cache metadata in the memory, but the size of metadata is very large. For example, an 8-TB storage system with a 4-KB deduplication unit creates 2 billion metadata sets (2 G = 8 TB/4 KB), and when the size of single metadata set is around 40 bytes, the total amount of metadata for 8 TB will be 80 GB (= 2 G x 40 bytes), which will not fit in the memory of the server.

Inline deduplication systems deduplicate data before storing them, while *post-process deduplication* systems first store data in their storages and deduplicate later. With post-process deduplication, a storage system can hide deduplication overheads; however, this requires extra storage. Our challenge was to minimize the deduplication overhead to minimize its latency by using memory to store metadata in inline deduplication.

Our goal was to develop an efficient data structure, the *multiple layer Bloom filter (MBF)*, to efficiently store metadata in memory.

This work is supported by the Green IT project of the New Energy and Industrial Technology Development Organization (NEDO).

II. RELATED WORK

A. Duplication detection by hash

Manber [13] developed a technique for finding similar documents in a filesystem using rolling checksum. Rabin fingerprinting [19] is a type of rolling checksum. Broder [3] developed shingling, which detects the similarity between sets of hash values to find near duplicate documents on the Internet.

B. Deduplication unit

Venti [18] and Foundation [20] are deduplication systems that detect redundancy with a fixed block size. Venti is an archival system. Foundation is a system for preserving snapshots of virtual machines (VMs) and uses a Bloom filter to detect duplicates.

NetApp's deduplication function [1] for file servers is integrated with WAFL and FlexVol [8]. It uses hashes for file-system blocks and finds duplicates. Hash collisons are resolved by byte-for-byte comparison. It is a post-process deduplication, which processes in the background.

The LBFS [16], Data Domain [22], HYDRAstor [7], REBL [11] and TAPER [10] find duplicates using content-defined chunks.

The Ocarina ECOsystem [17] uses a content-aware method, which works with several types of file types such as zip and pdf. It extracts sub-file objects and removes redundancy.

Meyer and Bolosky [15] conducted a large-scale study of file system contents on desktop machines, to evaluate the difference between whole-file and block-based deduplication.

C. Performance of deduplication and resource consumption

The Data Domain [22] uses the spatial locality of data in a backup stream to improve performance of searching for hash information.

Sparse indexing [12] is a technique for reducing the size of a data index kept in RAM by sampling chunks' hashes.

These heuristics work fine with large data sets with locality; however, unlike in backup storage systems, we need to assume that access patterns have small or no locality in primary storage systems.

D. *Better performance with SSDs*

It is also efficient to use fast devices for frequently used data. Chunkstash [6] and dedupv1 [14] use solid-state drives (SSDs) for metadata. Solid-state drives and metadata are a good combination because I/Os for metadata are always small and random, which SSDs are good at. However, we did not take this approach because we wanted to avoid hardware limitations, thus we tried to effeciently use memory.

E. *Bloom filter and its applications*

A Bloom filter [2] is a space-efficient data structure for managing the membership of a set. In deduplication technology, a Bloom filter is often used to check whether incoming data are already members of existing data.

Broder [4] describes a Bloom filter and its applications. Like our multilayer Bloom filter (MBF), there are examples that use multiple filters. For example, the attenuated Bloom filter [21] uses Bloom filters to find the shortest path to the requested data in a P2P network. The Bloomier filter [5] also has multiple Bloom filters and multiple layers, similar to our MBF. The Bloomier filter implements an associative array.

III. DBLK DESIGN

The deduplication block-device (DBLK) is a block-level storage system working as a backing store of an iSCSI target (*Figure 1*). *Figure 2* shows the major data structures in the DBLK.

The DBLK splits data into a fixed size (4 KB), computes a collision-free hash value (we are currently using *SHA-1*) of the data, deduplicates and compresses the data, and writes them sequentially as a *data log* on disk drives. Metadata are also stored in the data log. Logs are kept in *chunks*, which is the fixed size unit in disks.

The DBLK maintains the mapping between hash values to *physical block addresses (PBAs)*. It appends this *hash information*, including a hash value, the PBA of the data, and the number of references to the data to the *hash log*. Hash information is written sequentially in the hash log. If a write is a duplicate, the DBLK only updates the reference count in the hash log.

The DBLK maintains the mapping between a *logical block number (LBA)* and a SHA-1 hash value in a *block map* for each volume. It uses a Bloom filter to check if incoming data are duplicates of existing data.

Since the size of an entire hash log, which contains the mapping of hash values and PBAs, will be bigger than the physical memory of a server, the DBLK requires a *hash index* to find locations of hash information in the hash log from hash values.

Since data and metadata are over-written, *garbage collection* is necessary to discard unnecessary data in the log. There are two types of garbage collections. One is garbage collection for the data log. To find unnecessary data logs in a chunk, the DBLK detects a chunk that contains more garbage than the configured limit and checks the hash log whether the data in the chunk are referenced. If the data log is not referenced, it will be discarded in the garbage collection. The second type is that for the hash log. We call this *compaction* of the hash log. To find unnecessary hash log elements, the DBLK chooses a hash-log block and checks the number of references in the hash information. If the number of references is zero, an element in the hash log is removed during a compaction.

The DBLK periodically creates checkpoints in the data log. After it determines that all the metadata are written to the data log, it creates a checkpoint in a chunk. If a system failure occurs, the DBLK checks each chunk and finds an incomplete chunk in which there are data logs after the last checkpoint. The DBLK replays the data logs and makes sure that the metadata and data are consistent then restarts the service.

If the DBLK maps SHA-1 values to the hash-log blocks using a hash function, such as **mod of #hash-log-blocks**, we do not need to have a hash index. However, in a sequence of writes, every 4-KB data set will have one SHA-1 value, and hash-log I/Os will be distributed in hash logs because SHA-1 values have no locality. In this case, the system ends up with one hash-log I/O for each write I/O. In our design, we chose to append hash information to one hash-log block to minimize hash-log I/Os for better performance, and we need a hash index to search hash information.

The size of the hash index was the problem. *Figure 3* shows the size of metadata of our DBLK implementation with 8-TB physical capacities. The size of the hash log was 80 GB. It would not fit in the memory, thus we needed the hash index in the memory. If a hash index, which maps a 160-bit SHA-1 value to the address of the hash log, is implemented using a regular hash table or B-tree, its size will be more than 56 GB (=2 billion x (160-bit+PBA)), which will not fit in the memory of the DBLK server either. What is worse, if a hash index is stored in a hard drive, a hash index will be accessed randomly and the cache will not work well since SHA-1 values do not have locality. We solved this metadata problem by using our MBF.

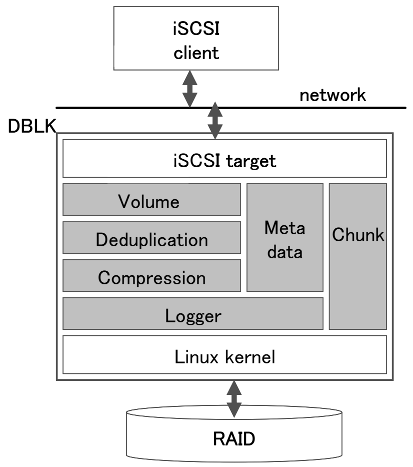

Figure 1: DBLK architecture

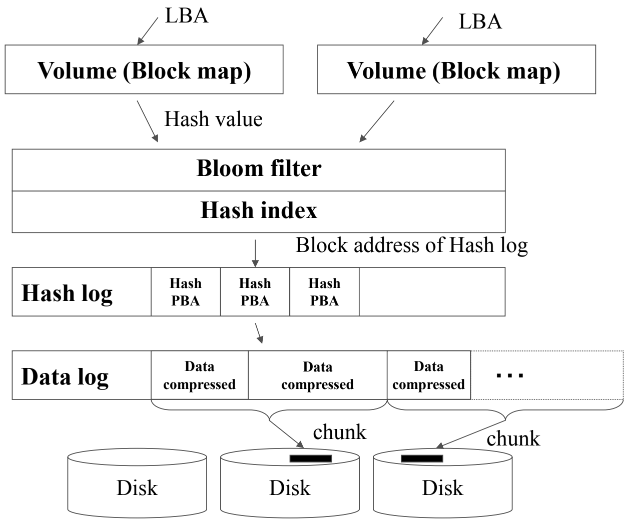

Figure 2: DBLK data structure

Physical capacity of Storage:	8 TB
Dedup&Compress block:	4 KB
Number of blocks:	8 TB/4 KB = 2 G
SHA-1:	160 bits
Hash log:	40 byte x 2 G = 80 GB

Figure 3: DBLK metadata size

IV. MULTILAYER BLOOM FILTER

A. *Bloom filter*

Before we explain our MBF, let us take a look at a conventional Bloom filter. A Bloom filter [2] is a space-efficient data structure used to determine if an element is already a member of the current *set*. It is possible to have *false positives*; however, it is not possible to have *false negatives*. This means that if the Bloom filter determines that an entry is new, it is 100% correct; however, if it detects that the entry is an existing member of the set, it could be wrong. The probability of false positives, *false positive rate (FPR)*, is controlled by the number of elments (n), bits (m), and positions (k) in the filter, as expressed in the following formula.

$$FPR = \{1-(1-1/m)^{kn}\}^k \cong (1-e^{-kn/m})^k \quad (1)$$

The k and m can be chosen for a given n, and the FPR can be decreased as desired.

The initial state of a Bloom filter is a cleared m-bit array. The k is used for an element to indicate k in m-bits. To add an element, compute the k and set the bits to 1 in the Bloom filter at those positions. To query an element, calculate k of the data and check k in the filter. If any of the positions return 0, the element is new to the filter. Otherwise, i.e., if all the positions are 1, the element might be an existing member in the Bloom filter. An element cannot be removed from a Bloom filter because several elements can share the same bits.

The DBLK uses a Bloom filter in the following way. For writes, the Bloom filter checks incoming data. If the filter is negative, the data is new, thus the system writes it. If the filter is positive, the system searches the hash log using the hash index and increments the reference count. If false positive, the system fails to find the hash log and writes the data. For reads, the system does not use a Bloom filter.

To solve the problem of the hash-index size, we extend the Bloom filter from detecting duplicates to finding the hash-log block. Under the main Bloom filter, we add two more Bloom filters, each half the size of the first. One corresponds to the left half of the hash log and the other filter corresponds to the other half. If the first Bloom filter is positive, then we check the two child filters. The area of the hash log can be narrowed to a single block by repeating this. When the DBLK reads the block of the hash log, it is highly expected, in the sense of the FPR, that the block contains the information.

We call this technique the MBF (*Figure 4)*. The lowest layer of the Bloom filter is associated with the hash-log blocks. If the Bloom filter is positive at the lowest layer, The DBLK reads the corresponding hash-log block and finds hash information in the block.

B. *Details of MBF*

To add an entry to the MBF, the DBLK first stores hash information in the hash log, computes k for the element, and sets the bit to 1 in the lowest Bloom filter that belongs to the hash log block. Then going up to the upper layer, set bits to 1 for the Bloom filters above.

The k of an element for the Bloom filters are computed once. The set of their values *{v1,v2,…,vk}* can be used in each layer by just calculating the moduli by filter size, from top to bottom, rather than computing positions for each layer. The set of position values in each layer are computed in the following way by using the *C*-language style modulo sign (*%*):

First layer: *{v1%m, v2%m,…, vk%m},*
Second layer: *{v1%(m/2), v2%(m/2),…,vk%(m/2) },*
.....
The i-th layer: {v1%($m/2^i$), v2%($m/2^i$),…,vk%($m/2^i$) }.

The false positive rate of the i-th layer does not depend on the number of layers i, because the FPR of the i-th layer is calculated using the following formula. The number of bits in each Bloom filter in the i-th layer is $m/2^i$, and the number of elements is $n/2^i$.

$$FPR(i) \cong (1-e^{-k(n/2^i)/(m/2^i)}) = (1-e^{-kn/m})^k \quad (2)$$

To query an entry in our MBF, the system computes k for the element and checks the MBF from top to bottom, as shown in *Figure 4*, and after the MBF returns the block number of the hash log, it reads the block and obtains the hash information.

Figure 4 shows an example of a binary MBF. Our MBF could be an *N*-ary tree of the Bloom filters rather than the binary trees explained above, and it could be implemented with fewer layers (i.e., less memory). If the number of hash-log blocks is *b*, with *N*-ary Bloom filters and the number of layers (i.e., the height of the MBF) is *h*, then $N^h = b$; therefore,

$$h = \log(b) / \log(N) . \tag{3}$$

Since the total amount of memory used by the MBF is $h \times m$ bits, it is possible to choose the right *N* to adjust the MBF to fit in the memory of the system which the DBLK runs.

For an 80-GB hash log, for example, there are 20 million 4 -KB blocks. If *N = 64*, *h = 8*, and if *N = 256*, *h = 3*. If a layer of the Bloom filter is 5.5 GB, the size of the MBF is 44 GB for *h = 8* and 16.5 GB for *h = 3*.

C. MBF-BT: an MBF optimization

In the *i*-th layer of a binary MBF, there are 2^i Bloom filters. When the DBLK queries an element, it must check *k* in the 2^i filters. This requires $k \times 2^i$ checks.

The *MBF-bitwise transposition (MBF-BT)* optimizes the MBF in the following way. *Figure 5* shows an example with *i = 8*. There are 64 filters in this layer. If there are *R* bits in each of the 64 filters, the entire layer can be mapped upon R-64 bit integers due to transposition. In this layout, the number of checks is reduced from 64 to 1 in the following way.

Suppose we have *R* integers $\{x_1, \ldots, x_R\}$ and we have to check *{p1,...,pk}*, the check is done in the following way:

```
Y=0xffffffffffffffff //initialize
for p in {p1, p2, …, pk}
    Y &= $x_p$   // bitwise AND
```

If the result of Y is zero, the filter layers are all negative, and if Y is not zero, one of the filters is positive, and we just need to find out which bit is 1 in Y. If the *j*-th bit is in Y, this means that the *j*-th Bloom filter is positive. Then the DBLK reads the *j*-th block of the hash log. In this example, the increase in speed is roughly 64x.

The MBF-BT lowers the cost of looking up multiple Bloom filters by using an integer-wise check rather than a bit-wise check; it is possible to reduce the height by increasing *N*. In our implementation, we chose *N = 1664* and the height of the MBF as two, including the top Bloom filter. The top Bloom filter is a collection of 1664 sub-Bloom filters. Therefore, the MBF manages 1664^2 hash-log blocks. The size of the MBF is 11 GB. Compared to the hash index with a hash table implementation we have discussed previously, which requires 48 GB plus 5.5 GB (i.e. 53.5 GB) for the Bloom filter, 11 GB is clearly an improvement. *Figure 6* shows examples of the different MBF configurations, the left one is the same as that shown in *Figure 4*, the middle one is with fewer layers, and the right one is the implemented MBF with two layers.

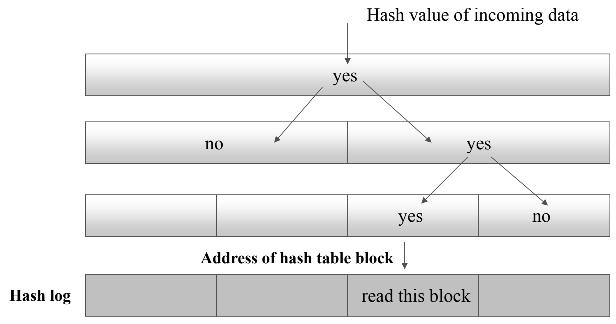

Figure 4: Binary multilayer Bloom filter (MBF)

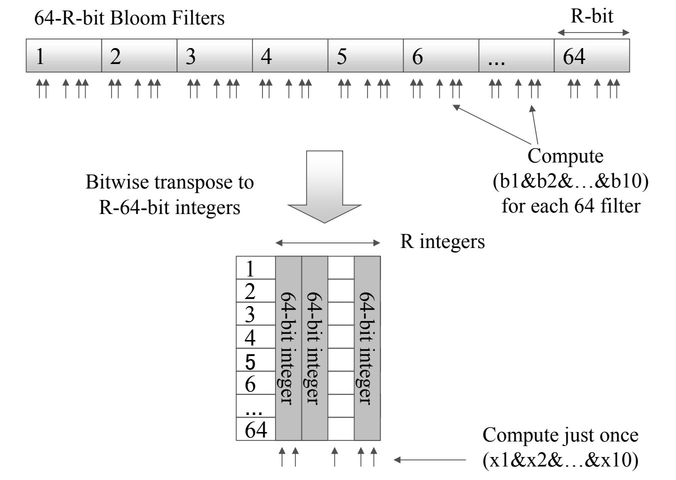
64-R-bit Bloom Filters
R-bit
1
2
3
4
5
6
...
64
Compute (b1&b2&…&b10) for each 64 filter
Bitwise transpose to R-64-bit integers
R integers
64-bit integer
64-bit integer
64-bit integer
Compute just once (x1&x2&…&x10)

Figure 5: MBF-BT: Physical Layout of MBF. MBF layer, which contains 64 Bloom filters, is mapped on several 64-bit integers. Five short arrows indicate where to check in bitmap

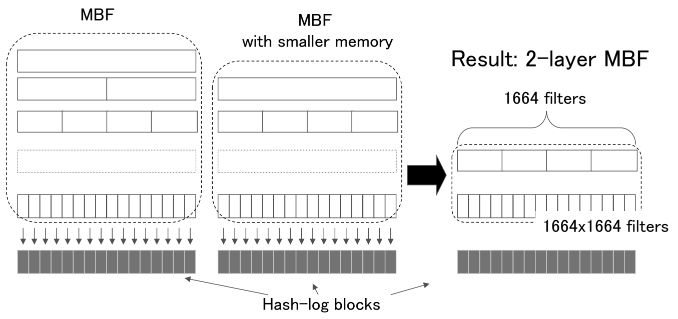

Figure 6: MBF implementation. Left MBF is binary implementation and middle MBF is 4-ary implementation. Right MBF is implemented in DBLK. It has 1664 filters at first layer and 1664 x 1664 filters at second layer. Each filter at bottom corresponds to hash –log block.

D. Refreshing and regenerating MBF

Since a Bloom filter does not allow removal of an element, the DBLK must periodically regenerate the filters because garbage bits of removed items remain in the MBF. This is done by reading the hash log and creating new filters. Refreshing is associated with compaction. The DBLK removes garbage in a hash-log block and refreshes the corresponding filter in the MBF.

Since the information in the MBF is generated from the hash-log information, storing the MBF on a disk drive is not necessary; however, it is possible to shorten initialization by reading the MBF from the disk rather than regenerating it.

E. Implementation

We used Linux iSCSI target framework software [9], and Zlib version 1.2.3 was used for compression in the DBLK. The DBLK uses SHA-1 (160 bits) for its hash value.

The chunk size, the unit of data stored on a disk, and the unit of garbage collection is 1 GB. The block size for deduplication and compression is 4 KB.

Instead of using a 64-bit integer for the MBF-BT, we used the MMX/SSE instruction set, which allows access to 128-bit registers. This means we obtain 128x performance.

The MBF is configured to manage our 8-TB RAID (*Figure 3*). As previously mentioned, there are only two layers of the MBF. The total amount of memory for the hash index and the Bloom filter is 11 GB (5.5 GB + 5.5 GB), and the number of sub-filters is $N = 1664$. The average query time for the MBF is less than 20 us.

The number of bits for each element in the Bloom filter is set to 23 (m/n by the notation in the previous section), and the k for a Bloom filter is 16. The estimated false positive rate is about 1.59×10^{-5}.

Our experiments showed that the DBLK latency is comparable or even less than its base RAID system for random writes, mostly because of the data log. It achieves more than 1500 *IOs per seconds (IOPS)* with random write and 700 IOPS for random read for 4-KB I/O sizes.

V. CONCLUSION

We developed a technique for reducing the latency of a deduplication system by storing metadata in memory using our MBF. Our MBF can be setup with a small amount of memory, compared to a regular hash table or a B+tree. In addition, the size of our MBF is adjustable to the size of the RAM by changing the number of layers, and the MBF-BT optimizes its access time based on its bitwise transposition.

We implemented the DBLK, a deduplication system for primary storage with a block device interface, using our MBF.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We also thank our managers, Katsuhiko Nishikawa, Yasuo Noguchi, Riichiro Take and Koichi Kumon for their suggestions and support.

REFERENCES

[1] C. Alvarez, "NetApp deduplication for FAS and V-Series deployment and implementation guide," Technical ReportTR-3505, January 2010.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM 13, 7 (1970), pp. 422-426, 1970.

[3] A. Z. Broder, "On the resemblance and containment of documents," Proceedings of the Compression and Complexity of Sequences 1997, pp. 21-29, 1997.

[4] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," Internet Mathematics, 2002.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," The fifteenth ACM-SIAM Symposium on Discrete Algorithms (SODA), Jan 2004.

[6] B. Debnath, S. Sengupta and J. Li, "ChunkStash: speeding up inline storage deduplication using flash memory," The 2010 USENIX Annual Technical Conference, June 2010.

[7] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: a scalable secondary storage," The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[8] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas, "FlexVol: flexible, efficient file volume virtualization in WAFL," The 2008 USENIX Annual Technical Conference, June 2008.

[9] T. Fujita, "Linux SCSI target famework", http://stgt.sourceforge.net/

[10] N. Jain, M. Dahlin, and R. Tewari, "TAPER: tiered approach for eliminating redundancy in replica synchronization," The 4th USENIX Conference on File and Storage Technologies (FAST '05), December 2005.

[11] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey, "Redundancy elimination within large collections of files," The 2004 Usenix Annual Technical Conference, June-July 2004.

[12] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise and P. Camble, "Sparse indexing: large scale, inline deduplication using sampling and locality," The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[13] U. Manber, "Finding similar files in a large file system," The Winter 1994 USENIX Technical Conference, January 1994.

[14] D. Meister and A. Brinkmann, "dedupv1: improving deduplication throughput using solid state drives (SSD)," IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), May 2010.

[15] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," The 9th USENIX conference on File and Storage Technologies (Fast '11), February 2011.

[16] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," The 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Alberta, Canada, October 2001.

[17] Ocarina-networks, "The Ocarina ECOsystem: content-aware dedupe and compression," June 2009.

[18] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," The First USENIX conference on File and Storage Technologies (Fast '02), January 2002.

[19] M. O. Rabin, "Fingerprinting by random polynomials," Harvard University Technical report TR-15-81, 1981.

[20] S. Rhea, R. Cox and A. Pesterev, "Fast, inexpensive content-addressed storage in Foundation," The 2008 USENIX Annual Technical Conference, June 2008.

[21] S. C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," IEEE Computer and Communications Societies (INFOCOM '02), June 2002.

[22] B. Zhu, K.Li, and H. Patterson, "Avoiding the disk bottleneck in the Data Domain deduplication file system," The 6th USENIX Conference on File and Storage Technologies (FAST '08), February 2008.