dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD)

Dirk Meister Paderborn Center for Parallel Computing
dmeister@uni-paderborn.de André Brinkmann Paderborn Center for Parallel Computing
brinkman@uni-paderborn.de

***Abstract*—Data deduplication systems discover and remove redundancies between data blocks. The search for redundant data blocks is often based on hashing the content of a block and comparing the resulting hash value with already stored entries inside an index. The limited random IO performance of hard disks limits the overall throughput of such systems, if the index does not fit into main memory.**

This paper presents the architecture of the dedupv1 deduplication system that uses solid-state drives (SSDs) to improve its throughput compared to disk-based systems. dedupv1 is designed to use the sweet spots of SSD technology (random reads and sequential operations), while avoiding random writes inside the data path. This is achieved by using a hybrid deduplication design. It is an inline deduplication system as it performs chunking and fingerprinting online and only stores new data, but it is able to delay much of the processing as well as IO operations.

I. Introduction

Data deduplication systems discover redundancies between different data blocks and remove these redundancies to reduce capacity demands. Data deduplication is often used in disk-based backup systems since only a small fraction of files changes from week to week, introducing a high temporal redundancy [1], [2].

A common approach for data deduplication is based on the detection of exact copies of existing data blocks. The approach is called fingerprinting- or hash-based deduplication and works by splitting the data into non-overlapping data blocks (chunks). Most systems build the chunks using a content-defined chunking approach based on Rabin's fingerprinting method [2], [3]. For most data sets, content-defined chunking delivers a better deduplication ratios than simple static-sized chunks [4]. The system checks for each chunk, whether another already stored one has exactly the same content. If a chunk is a duplicate, the deduplication system avoids storing the content. The duplicate detection is usually not performed using a byte-by-byte comparison between the chunks and all previously stored data. Instead, a cryptographic fingerprint of the content is calculated and the fingerprint is compared with all already stored fingerprints using an index data structure, often called chunk index.

The size of the chunk index limits the usable capacity of the deduplication system. With a chunk size of 8 KB and 20 byte fingerprints, the chunk size grows per 1 TB unique data by around 2.5 GB (without considering any overheads or additional chunk meta data). A large scale deduplication system can easily exceed an economical feasible main memory capacity. Therefore, it can become necessary to store the fingerprint index on disk. In this case, the limited random IO performance of disks leads to a significant throughput drop of the system.

In this paper, we evaluate how solid-state drives (SSDs) might help to overcome the disk bottleneck. Solid-state drives promise an order of magnitude more read IOPS and faster access times than magnetic hard disk. However, most current SSDs suffer from slow random writes. We present a deduplication system architecture that is targeted at solid-state drives as it relies on the sweat spots of SSDs while avoiding random writes on the critical data path. We propose using the concept of an in-memory auxiliary index to move write operations into a background thread and a novel filter chain abstraction that makes it easy for developers and researcher to modify redundancy checks to either improve the security of the deduplication or to speed up the processing.

II. Background and Related Work

The ability to lookup chunks fingerprints in the chunk index is usually the performance bottleneck for disk-based deduplication systems. The bottleneck is caused by the limited number of IOPS (IO operations per second) possible with magnetic hard disks. Even enterprise class hard disks can hardly deliver more than 300 IOPS [5]. Because of that, solid state drives have gained traction in server environments as they promise an order of magnitude higher IOPS, high throughput, and low access times [6], [7].

Current state-of-the-art SSDs are reported to allow 3,000 to 9,000 read IOPS per second, which is equivalent to a disk array with 10 to 30 high-end disks [5]. In addition, SSDs allow a high sequential throughput (usually over 100 MB/s). The weak point of most SSDs is the limited number of random writes. Narayanan et al. report around 350 random writes per second for an enterprise SSD [5].

While the price per read IO is usually better for SSDs than for enterprise-class hard disks, the capacity remains low and the price per GB high. However, in our deduplication setting the size of the chunk index – the most performance critical component of a deduplication system – is too large to be hold in main memory in a cost effective way, but can be hold on a single or a low number of SSDs.

978-1-4244-7153-9/10/ $26.00 @2010 IEEE

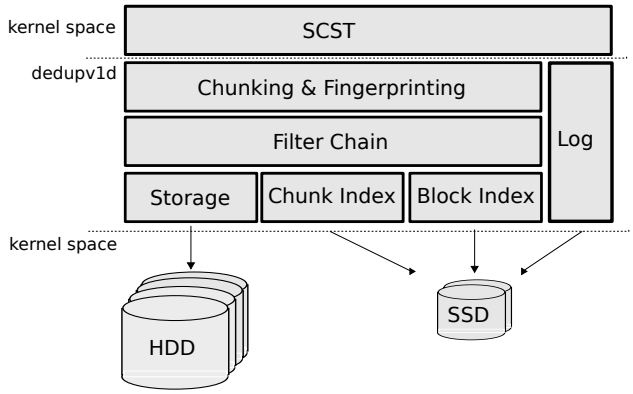


Fig. 1. Architecture of the dedupv1 deduplication system

Research on deduplication systems deals with the deduplication strategy, the resulting deduplication ratio, as well as the deduplication performance.

The most prominent deduplication strategies are fingerprinting-based and delta encoding-based. We focus on fingerprinting-based deduplication, which is based on the detection of exact replicas, comparing the fingerprint of already stored blocks with the fingerprint of a new block. Static chunking assumes that each block has exactly the same size [4], while content-defined chunking, which is based on Rabin's fingerprinting [8], is able to deliver a better deduplication ratio [2], [3], [9]. The usage of hash values can lead to hash collisions, identifying two chunks as duplicates, even if their contents differ (for a discussion, see [10], [11]).

The main bottleneck of fingerprinting approaches appears, if the fingerprint index does not fit into main memory and has to be stored on disk, inducing a strong performance drop. Using 20 Byte SHA-1 hash values already needs 2.5 GB of main memory for each TB of unique data, filling up main memory quite fast. Several heuristics for archiving systems have been introduced to overcome this drawback. Zhu et al. use bloom filters to keep a compressed index, simplifying the detection of unique data [2]. Furthermore, they introduce locality-preserving caching, where indexes are stored in containers, which are filled based on the data sequence, preserving locality in backup streams. Lillibridge et al. do not keep the complete chunk index in main memory, but divide the index into "segments" of 10 MB chunks. For each segment, they choose k champions and the lookup is only performed inside the champion index [12]. In this case, the authors trade deduplication ratio (not all duplicates can be detected) for performance. Trading deduplication ratio for performance has also been proposed for a parallel setting in [13].

III. Architecture of the dedupv1 System

We have developed dedupv1 to evaluate and compare the performance impact of solid-state technology in deduplication systems. The high-level architecture of the system is shown in Figure 1.

The dedupv1 system is based on the generic SCSI target subsystem for Linux (SCST) [14]. The dedupv1 userspace daemon communicates with the SCST subsystem via ioctl calls. Using SCST allows us to export deduplicated volumes via iSCSI. The data deduplication is therefore transparent to the user of the SCSI target.

A. Chunking and Fingerprinting

The chunking component splits the request data into smaller chunks. Each chunk has to be checked whether its data has already been stored or if the content is new. We have implemented different, configurable chunking strategies inside the chunking component. Usually, we use content-defined chunking (CDC) based on Rabin's fingerprinting method with an average chunk size of 8 KB [8] as e.g. in [2].

Each chunk is fingerprinted after the chunking process using a cryptographic hash function like SHA-1 or SHA-256. Our default is SHA-1, but our system is not limited to that choice.

B. Filter Chain

The filter chain component decides if the content of a chunk is a duplicate or if the chunk content has not been stored before. The filter chain can execute a series of filters. After each filter step, the result of the filter determines which filter steps are executed afterwards. Each filter step returns with one of the following results:

EXISTING:
: The current chunk is an exact duplicate, e.g. a filter that has performed a byte-wise comparison with an already stored chunk returns the result. The execution of the filter chain is stopped if a filter step returns this result.

STRONG-MAYBE:
: There is a very high probability that the current chunk is a duplicate. This is a typical result after a fingerprint comparison. Other filter that cannot provide any better result than STRONG-MAYBE are not able provide a better information than that it is very likely that the chunk is a duplicate. Therefore after this result, it only makes sense to execute filters that can return EXISTING. STRONG-MAYBE filters are skipped.

WEAK-MAYBE:
: The filter cannot make any statement about the duplication state of the chunk. All filter steps later in the chain are executed.

NON-EXISTING:
: The filter rules out the possibility that the chunk is already known, e.g. after a chunk index lookup returns a negative result. The execution of the filter chain is canceled if a filter returns this result.

If the chain classifies a chunk as new, the system runs a second time through the filter chain so that filters can update their internal state.

This flexible duplicate detection enables the development and evaluation of new approaches and requires minimal implementation efforts. The currently implemented filters are:

Chunk Index Filter:
: The chunk index filter (CIF) is the basic deduplication filter. It checks for each chunk whether the

fingerprint of the chunk is already stored in the chunk index. The filter returns STRONG-MAYBE, if a chunk fingerprint is found in the chunk index. Otherwise, the chunk is unknown and the filter returns NOT-EXISTING. Afterwards, during the update-run, the chunk index filter stores the new fingerprint inside the index structures.

This filter performs an index lookup for each check, which often hits the SSD or the disk storing the chunk index. If possible, other filters should be executed before the chunk index filter so that this filter is only executed if no other filter returns a positive answer.

Block Index Filter:

The block index filter (BIF) checks the current chunk against the block mapping of the currently written block that is already present in main memory. If the same chunk is written to the same block as before, the block index filter is able to avoid the chunk index lookup.

In a backup scenario, we are able to clone the blocks of the previous backup run using a fast server-side copy approach to the volume that will hold the new backup data. When the current backup data is written to the clone volume and if the data stays at the same block, the block index filter is able to avoid some chunk index checks.

Byte Compare Filter:

The byte compare filter (BCF) performs an exact byte-wise comparison of the current chunk and an already stored chunk with the same fingerprint. While this introduces additional load on the storage systems, it also eliminates the possibility of unnoticed hash collisions.

Bloom Filter:

We have implemented this and the container cache filter described next to test the flexibility of the filter chain concept and to show how easy deduplication optimizations can be implemented using this programming abstraction. Both optimizations have been presented by Zhu et al. [2].

A bloom filter is a compact data structure to represent sets. However, a membership test on a bloom filters has a certain probability of an false positive [15], [16]. In the context of data deduplication, bloom filter can be applied as follows: The fingerprint of each known chunk is inserted into the bloom filter. For each chunk, the bloom filter is checked for the fingerprint. If the membership test is negative, we are sure that the chunk is unknown and NON-EXISTING is returned. If the membership test is positive, the filter returns WEAK-MAYBE, as there is the possibility of false positives. The bloom filter helps to accelerate the writing of unknown chunks, e.g. in a first backup generation because expensive chunk index lookups are avoided.

Container Cache Filter:

The container cache filter is also an implementation of concepts presented by Zhu et al. [2]. It compares a fingerprint with all entries of a LRU read cache. The read cache uses containers, where each container includes a set of fingerprints. If the check is successful, a STRONG-MAYBE result is returned and other filters, especially the chunk index filter, are not executed.

If the check is negative, a WEAK-MAYBE result is returned. After the filter chain is finished and the result has been a STRONG-MAYBE (e.g. based on a chunk index lookup), a last artificial filter claiming that it allows an EXISTING result is responsible for loading the fingerprint data of the container of the chunk into the cache. The result of this post process filter is also a STRONG-MAYBE.

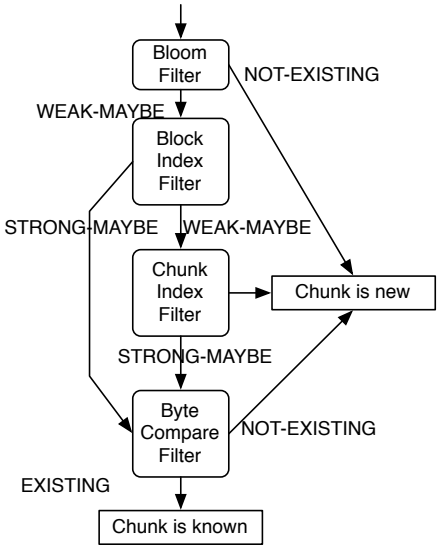


Fig. 2. Illustration of the filter chain control flow (example)

To illustrate the filter chain concept, let us consider an example configuration for the filter chain that consists of a bloom filter, a block index filter, chunk index filter, and a byte compare filter. An already known chunk will be detected by the bloom filter. However, since the bloom filter might return a false positive, this only leads to a WEAK-MAYBE. Therefore, the block index filter is executed. If the previous block mapping of the current block also contains a chunk with the same fingerprint, the filter returns a STRONG-MAYBE and the chunk index filter and any other filter that can at best return a STRONG-MAYBE result are skipped because it would not provide any new information and we can be reasonably sure that the chunk is known. If a chunk with the same fingerprint is not been used in the block before, WEAK-MAYBE is returned.

If that is the case, the chunk index filter performs an index lookup, finds the chunk index entry for the given fingerprint and returns STRONG-MAYBE together with the container id of the container that stores the chunk (see the next subsection for a description of containers). If the byte compare filter is executed and it reads the container data of the chunk and performs a byte-wise comparison, which probably leads to an EXISTING result.

Figure 2 illustrate the possible control flow with the example configuration.

C. Storage

The chunk data is stored using a subsystem called *chunk storage*. The chunk storage collects chunk data until a container of a specific size (often 4 MB) is filled up and then writes the complete container to disk. The chunk storage is therefore similar to the chunk container of Zhu et al. [2] and Lillibridge et al. [12].

If a currently open container becomes full, the container is handed over to a background thread that writes the data to the attached storage devices. The background thread notifies the system about the committed container using the log. Other components, e.g. the chunk index, can now assume that the data is stored persistently. The chunks of containers that are not yet committed to disk have to be stored in the auxiliary index and must not be stored persistently until the chunk index receives a notification from the container store.

D. Chunk Index

A major component of the system is the *chunk index* that stores all known chunk fingerprints and other chunk meta data. The lookup key of the index is a (20 byte for SHA-1, 32 byte for SHA-256) fingerprint. In addition, each chunk entry contains the storage address of the chunk in the chunk store and a usage counter used by the garbage collection.

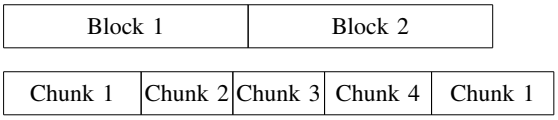
The chunk index uses two index structures, the persistent index that uses a paged disk-based hash table and an in-memory auxiliary index.

The auxiliary chunk index stores chunk entries for all chunks whose containers are not yet written to disk (non-committed chunks) as such chunk entries are only allowed to be stored persistently after the chunk data is committed. In addition, the in-memory auxiliary index is used to take index writes out of the critical path. It also stores chunk entries that are ready to be committed, but are not yet written to disk. If the auxiliary index grows beyond a certain limit or if the system is idle, a background thread moves chunk metadata from the auxiliary index to the persistent index. In case of a system crash, the chunk index is recovered by importing the recently written chunks from the chunk store.

The design of the chunk index is influenced by the LSM tree data structure that also maintains a persistent and an in-memory index [17], [18]. However, the goals are different. The goal of an LSM tree is to minimize the overall IO costs, e.g. by optimizing the merging of the in-memory index and the persistent index using a special on-disk format. This is important in an OTLP setting where no idle times can be assumed. Our goal is to delay the IO such that the update operations can be done outside the critical path or for highly redundant backups even after the backup itself.

E. Block Index

The *block index* stores the metadata that is necessary to map a block of the iSCSI device to the chunks of varying length



(a) Illustration of a data stream split up into blocks of a fixed length and chunks of variable length

Block	Chunk	Offset	Size	Container
Block 1	Chunk 1	0	9	–
	Chunk 2	0	6	–
	Chunk 3	0	1	–
block 2	Chunk 3	1	5	–
	Chunk 4	0	7	–
	Chunk 1	0	4	–

(b) Visualization of the block mapping that results from the data stream shown above

Fig. 3. Example of a block mapping

that from the most recent version of the block data. We call such a mapping the "block mapping". The size of a block can be set independently from the device block size of the iSCSI device (often 4 KB). Usually a much higher block size is chosen (64 KB to 1 MB) so that a block mapping contains multiple full chunks.

The purpose is very similar to the data block pointers of a file in a file system. In a file system the data block pointers denote which data blocks contain the logical data of a file. A block mapping denotes which chunks represent the logical data of a block. In contrast to a file system where usually all data blocks have the same length, the chunks have different lengths and often there is no alignment between chunks and blocks. So a block mapping consists of an ordered list of chunk fingerprints and an offset / size pair denoting the data range within the chunk that is used by the block. Additionally, we store the container id in the block mapping item, which is the foundation of the block index filter and a high read performance. Figure 3 illustrates how an example data stream is split into static-sized blocks and variable sized chunks (a) and how the block mapping for such a data stream looks like (b).

As the chunk index, the block index consists of a persistent and an in-memory index. The in-memory index stores all block mappings that are updated, but are not yet allowed to be committed since referenced chunk data are not committed to disk. We also hold fully committed block mappings in the auxiliary index to avoid expensive write operations in the critical path. It should be noted that consistency is still guaranteed because all operations are written in the operations log.

F. Log

The log is a shared operations log that is used for two purposes: To recover from system crashes and to delay write operations.

If the dedupv1 system crashes, a replay of the operations log ensures a consistent state, meaning especially, but not limited to this, that no block references a chunk that is not stored in the chunk index and that no chunk index entry references container storage data that has not been written to disk. The log

also helps to delay may write operations so that the amount of IO operations in the critical path is minimized because the log assures that the delayed operations can be recovered either in case of an crash and because the system can process logged operations during a background log replay, e.g. the garbage collection must not update its state inline.

IV. Evaluation

In this section, we first describe the methodology and the environment used to evaluate the SSD-based deduplication architecture proposed in the previous section. Afterwards, we present performance benchmarks with various index configurations.

We distinguish the first backup generation and further backup generations. The storage system has not stored any data before the first backup generation and the first backup run cannot utilize any temporal redundancy. In the second (and later) generations, the deduplication system can use chunk fingerprints already stored in the index. For the first backup generation, we used a 128 GB subset of files stored on a file server used at our institute. The file system contains scientific scratch data as well as workgroup data in the first data generation. The second generation data is randomly generated based on the first generation data and trace informations from a recent study [1]. The traffic data files contain on average 32.5% redundancy within a single backup run (internal redundancy) and 97.6% redundancy, if previous backup runs are also utilized (temporal redundancy). A full description of the trace generation process can be found in the extended technical report version of this paper [19]. Since we only benchmark the first and the second generations, we are not able to observe long-term effects.

The evaluation hardware consists of a server with an 8-core Intel Core i7 CPU, 16 GB main memory, a fibre channel interconnect to a SAN with 11 disks a 1 TB configured as RAID-5 with one spare disk, a 10 GB network interconnect, and four 2nd generation Intel X25-M SSDs with 160 GB capacity each. We limited the available main memory capacity to 8 GB to be more comparable with previous reported results about the throughput of deduplication systems.

Up to four worker nodes are concurrently writing backup data to the deduplication system using a 1 GB network.

The configuration is based on Content-defined Chunking (CDC) with an expected chunk size of 8 KB, a container size of 4 MB using no compression, and a chunk index initiated with a size 32 GB and 2 KB pages. The size is chosen large enough to hold over 750 million chunks, which is equivalent over 5 TB of raw data when we consider a maximum fill ratio of 70% and 32 byte data per chunk. The chunk index, the block index, and the operations log are spread to all SSDs. The chunk data is always stored on attached SAN. We allow the auxiliary (in-memory) chunk index to contain all chunks of a run. The only filter we use in this setting is the chunk index filter. We performed five measurements for each configuration and calculated the confidence intervals using 0.95 confidence level. All values are reported as averages in its steady state.

We evaluated the system by varying the storage system of the index. Besides the base configuration with four X-25M SSDs, we also evaluated the system using only one and two SSDs of the same kind. Additionally, we also evaluated a configuration with a completely main-memory based chunk index and a SAN-based block index.

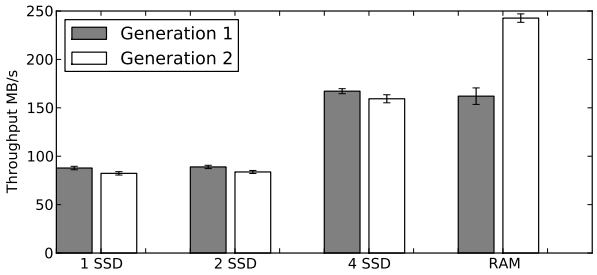


Fig. 4. Throughput using different index storage systems

Figure 4 shows the average throughput using the floating traffic and the block traffic for different index storage systems.

The base configuration with 4 SSDs achieves 167.2 MB/s (+/− 3.4 MB/s) in the first data generation and slightly less with 160.3 MB/s (+/− 4.3 MB/s) in the second generation. The four SSDs provide 2,848 (+/− 64) read IOPS per SSD during the deduplication. Additionally to the raw SSD speed, the throughput is increased by caching effects due to the OS page cache and the auxiliary chunk index that allows chunk index checks for around 30% of the chunks (internal redundancy). With 2 SSDs, the throughput is reduced to 88.4 MB/s (+/− 1.2 MB/s) and 83.3 (+/− 1.5 MB/s) for the two inspected backup generations.

Interestingly, the throughput with only a single SSD is not significantly lower than using two SSDs. The first generation is written with an average throughput of 87.8 MB/s (+/− 1.8 MB/s) and the second generation is written with an average throughput of 82.3 MB/s (+/− 1.7 MB/s). This is caused by a much higher number of performed read IOPS in that configuration.

The reason for this quite unexpected behavior is that instead of around 3,000 IOPS per SSD executed by the 2- and 4-SSD system, the single-SSD system executed 5,375 (+/− 54) reads per second. In additional raw IO measurements we noticed that a higher IO queue length – that is the number of concurrent requests that are issued to the disk – leads to a much higher number of performed requests per second for the Intel X25-M SSDs (around $9-12$ on average). Since all requests are split to multiple SSDs in the other configuration, the IO queue length is smaller (around $5-6$ on average) here.

If the complete chunk index fits in memory and the block index is stored on disk, the system achieves a throughput of 162 MB/s (+/− 8.5 MB/s) for the first generation, respectively 242.7 MB/s (+/− 4.4 MB/s) for the second generation. Surprisingly, this is not much faster than the SSD-based system. In that configuration, the block index builds the bottleneck.

The bottlenecks of all four setups are visible in the profiling data, which is shown in Figure 5. The figure shows the shares of different system components on the overall wall clock time on the data path. In all SSD-based configurations, the chunk index is the major bottleneck. In the RAM-based system, other

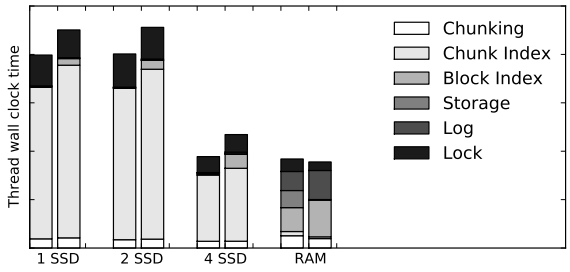


Fig. 5. Average ratios of system components on the overall runtime in the data path. The left bars denote the first generation runs, the right bars denote the second generation runs

bottlenecks become dominant: The disk-based block index and the storage component.

Zhu et al. presented various techniques to avoid expensive index lookups including a bloom filter and special caching schemes [2]. They assume that in every backup run the data is written in nearly the same order. This request order locality is only given in backup scenarios. They achieved a throughput of 113 MB/s for a single data stream and 218 MB/s for 4 data streams on a system with 4 cores, 8 GB RAM and 16 disks with a deduplication ratio of 96%. The throughput would degenerate in low-locality settings. Lillibridge et al. presented a deduplication approach using sampling and sparse indexing. They reported a throughput of 90 MB/s (1 stream) and 120 MB/s (4 streams) using 6 disks and 8 GB RAM [12] based on similar assumptions as Zhu et al.. We achieve more than 160 MB/s without depending on locality.

This comparison shows that it is possible to build a deduplication system using solid-state drives that are able to provide a performance that is on-par with state-of-the-art deduplication systems.

V. Conclusion

The evaluation shows that current SSD technology can build the basis for high-throughput fingerprint-based data deduplication. Without depending on locality, the system achieves over 160 MB/s in all backup generations with a single node system. The system is build around the specific characteristics of SSDs such as using additional in-memory index structures that are inspired by LSM trees to avoid random writes. The system can easily be extended by a flexible and powerful filter chain approach.

Our future research focus may lie on the long-term behavior of deduplication systems considering aging effects as well as further scaling aspects.

References

[1] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *Proceedings of 2nd The Israeli Experimental Systems Conference (SYSTOR'09)*, May 2009.

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

[3] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, 2004.

[4] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, 2002.

[5] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of 4th ACM European conference on computer systems (EuroSys '09)*, April 2009.

[6] "Solid state 101 - an introduction to solid state storage," White Paper, Storage Networking Industry Association, January 2009.

[7] D. Myers, "On the use of nand flash memory in high-performance relational databases," Master's thesis, MIT, February 2008.

[8] M. O. Rabin, "Fingerprinting by random polynomials," TR-15-81, Center for Research in Computing Technology, Tech. Rep., 1981.

[9] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference*, San Fransisco, CA, USA, 1994, pp. 1–10.

[10] V. Henson, "An analysis of compare-by-hash," in *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2003, p. 3.

[11] J. Black, "Compare-by-hash: a reasoned analysis," in *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, 2006, p. 7.

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

[13] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, Sep. 2009.

[14] V. Bolkhovitin, "Generic scsi target middle level for linux," http://scst.sourceforge.net/, 2003.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[16] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, June 2000.

[17] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[18] J. Stender, B. Kolbeck, M. Hgqvist, and F. Hupfeld, "Babudb: Fast and efficient file system metadata storage using lsm-trees," in *Proceedings of the 6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, May 2010.

[19] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives (technical report version)," University of Paderborn, Tech. Rep., 2010.