University Paderborn
Paderborn Center for Parallel Computing

Technical Report

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

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. An important advantage of the dedupv1 system is that it does not rely on temporal or spatial locality to achieve high performance. But using the filter chain abstraction the system can easily be extended to facilitate locality to improve the throughput.

1 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, the chunk size grows per 1 TB unique data by 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.

This paper is organized as follows: After describing related work in Section 2, we explain the architecture of the SSD-based deduplication system dedupv1 in Section 3. In Section 4, we present our evaluation methodology and environment, which includes a scalable approach to generate deduplication traffic, before we state the performance results. In Section 5, we give a conclusion and describe possible areas for further work.

2 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].

Nowadays SSDs are usually built on NAND flash memory. Most SSDs – including those used for the evaluation section – are connected via standard disk interfaces like SATA or SAS. Some enterprise SSDs like the Fusion-io ioDrive use PCI-Express interface [8].

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]. Birrel et al. believe that the explanation for this behavior is that usually a large logical page size is used to map sectors to flash pages, which cases significant re-reads and re-writes for small request sizes [9].

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.

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 and delta encoding. Fingerprinting 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 [10], is able to deliver a better deduplication ratio [2,3,11]. The usage of hash values can lead to hash collisions, identifying two chunks as duplicates, even if their contents differ (for a discussion, see [12, 13]).

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 [14]. 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 [15].

Delta encoding-based data deduplication tries to find near duplicates and only stores the resulting delta. It is based on standard technologies from the area of information retrieval. The shingling-process calculates a set of hash values over all fixed-sized windows and selects e.g. the biggest results as shingles or features [16]. The resemblance of two chunks is then defined as the number of joint features divided by their union [17]. Douglis and Iyengar claim that the delta encoding has become extremely efficient and it should not be the bottleneck of deduplication environments [18]. The big advantage of delta-encoding environments is that the index can be made as small as four MB for a 1 TB environment [19]. Nevertheless, the reconstruction of chunks can trigger the reconstruction of additional chunks and slowdown both reading and writing data. The problem even becomes worse, if the system gets older and the depth of deduplication tree increases [20]. This problem can be limited by clean-up processes or a restriction of the maximum tree depth.

Deduplication ratios have been analyzed in several papers [1, 21–23]. Most previous works that introduce new deduplication techniques or other improvements provide evaluations using real data sets, including web crawls, source trees or e-mail files or selected data of specific file types [3, 18, 20, 24–26]. Only some works also use real world user files for their evaluation [1, 3, 18, 26].

It is hard to compare different throughput results for deduplication systems, because there is no unified test-set. Additionally to the used traffic data, the deduplication ratio of the data, and the used hardware have also an impact on the throughput results.

Quinlan et al. reported for their Venti deduplication system with two cores, 2 GB RAM and 8 disks a throughput of 3.7 MB/s for new data, and 6.5 MB/s for redundant data [4]. This data corresponds clearly with the disk-based performance. Zhu et al. have presented various techniques to avoid expensive index lookups including a bloom filter and special caching schemes [2]. Some of these techniques assume that in every backup run the data is written in nearly the same order. They achieved a throughput of 113 MB/s for a single data stream and 218 MB/s for four data streams on a system with 4 cores, 8 GB RAM and 16 disks with a deduplication ratio of 96%.

Lillibridge et al. have presented a deduplication approach using sampling and sparse indexing. They have reported a throughput of 90 MB/s (1 stream) and 120 MB/s (4 streams) using 6 disks and 8 GB RAM [14]. While based on similar assumptions as Zhu et al., they have detected less redundancy, but their throughput is not decreasing if the locality is weaker.

Lui et al. achieved a throughput of 90 MB/s with one storage node (8 cores, 16 GB RAM, 6 disks) and an archival server (4 cores, 8 GB RAM) [24]. Since it is not clear

how the disk bottleneck has been handled, we assume that chunk lookups are directly answered from cache.

The throughput achieved by Zhu et al. and the deduplication quality of Lillibridge et al. are based on techniques that highly depend on the specific locality assumptions, which are only valid in backup scenarios. The approaches presented in this paper do not depend on data locality, making deduplication more attractive for scenarios where no such locality exists.

3 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 3.1.

The dedupv1 system is based on the generic SCSI target subsystem for Linux (SCST) [27]. 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 similar SCSI integration for user-space devices is provided via the tgt Storage Target Framework that could easily be used in our environment [28].

3.1 Chunking and Fingerprinting

The chunking component splits the request data into smaller chunks. These chunks usually have a size between 4 KB and 32 KB and build the deduplication unit. Each chunk has to be checked whether its data has already been stored or if the content is new. A limitation is that redundancies cannot be detected if only parts of the chunk have been previously stored. The choice of the chunk size is therefore a tradeoff between possible deduplication ratio and metadata size, as smaller chunk sizes lead to more fingerprints in the index structures.

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 [10]. The CDC strategy, also called variable-sized chunking, calculates a hash value for all substrings of a fixed size (usually 48 bytes) of the file. The chunk ends if it holds for the hash value f of the substring that

$$f \bmod n = c \text{ for a constant } 0 < c \leq n.$$

The chunks generated by this method have a variable size with an expected size n. CDC is used by nearly all deduplication systems [2, 3, 29, 30]. Minimal and maximal chunk sizes are enforced to avoid too small and too large chunks [31].

Alternatively, static sized-chunks can be used. The calculation of static-sized chunks is significantly faster, but usually results in a reduced deduplication rate [1].

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.

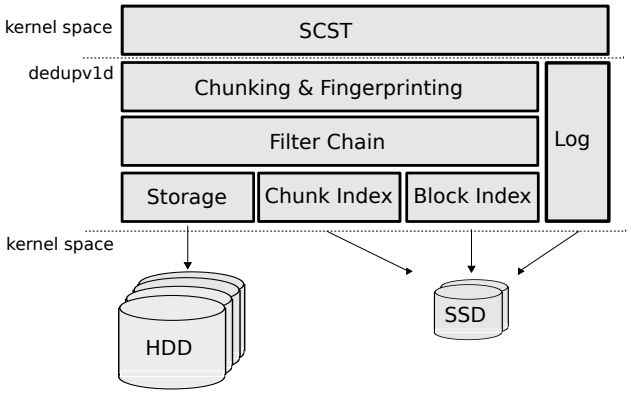


Figure 3.1: Architecture of the dedupv1 deduplication system

3.2 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. The design is similar to the "Chain of Responsibility" pattern [32].

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.

The block index filter can always be activated as the operation is very fast and does not perform any IO operations.

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 [33, 34].

The probability of a false positive can be bounded after the insertion of n objects, using k hash functions and m bits of main memory by

$$(1 - (1 - \frac{1}{m})^{kn})^k.$$

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.

Zhu et al. assume that data is often written in the same order week after week in a backup scenario. Therefore chunks that are processed within a short time-span at their first occurrence are likely to be processed within a short time-span in later backup runs. These chunks are likely to be stored in the same container. If the first chunk is checked, a chunk index lookup reveals the container id and the post process filter loads the corresponding fingerprint data for the container. Later requests to other chunks in that container already find the cached fingerprint data and can avoid expensive chunk index lookups.

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.

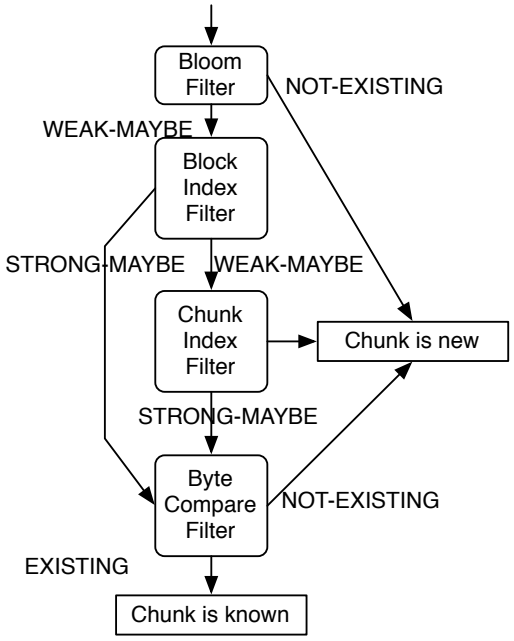


Figure 3.2: Illustration of the filter chain control flow (example)

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.

If a filter returns NON-EXISTING, the chunk is unknown. In this case, the system stores the data in a new container and inserts the chunk into the auxiliary index of the chunk index. If we avoid using a bloom filter, the chunk index filter would perform an index lookup to return the same result. In both cases, we do not perform any index update operations in this critical path of the execution, which would lead to a random write on the persistent chunk index.

Figure 3.2 illustrate the possible control flow with the example configuration

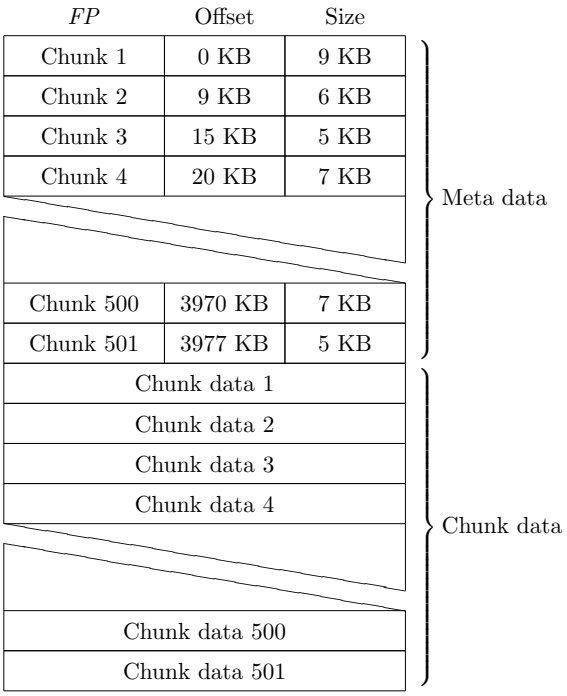


Figure 3.3: Structure of a container

3.3 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.

A container includes a metadata section and a data section. The metadata section stores the fingerprints and the position and size of the chunk data in the data section. The data section contains the chunk data and further metadata, e.g. the compression type if one is used. The structure of a container is illustrated in Figure 3.3.

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.

The dedupv1 system supports compressing the container data using zlib and bz2 compression to further reduce the needed storage capacity.

The chunk storage is similar to the chunk container of Zhu et al. [2] and Lillibridge et al. [14]. Even the Venti system used a similar data structure called “arena” [4].

3.4 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 and an in-memory auxiliary index. The persistent index is stored in a paged disk-based hash table. Since the chunk index keys are cryptographic fingerprints, we cannot assume any spatial locality so that an ordered data structure like a B-Tree variant would not provide its benefits.

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 [35,36]. 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. Figure 3.4 visualizes what chunk data is stored in the persistent and the auxiliary chunk index and when entries are moved between them.

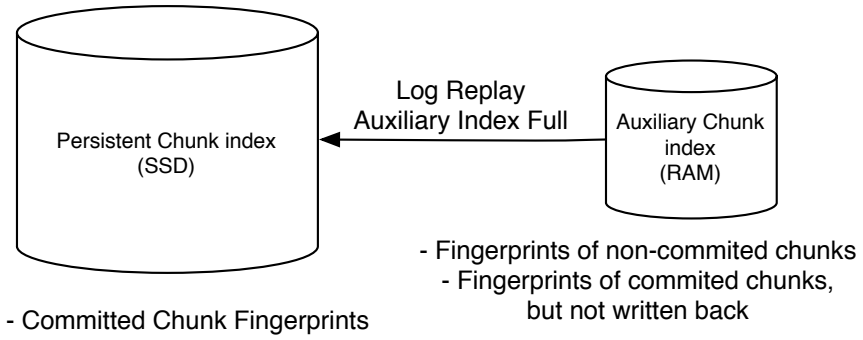
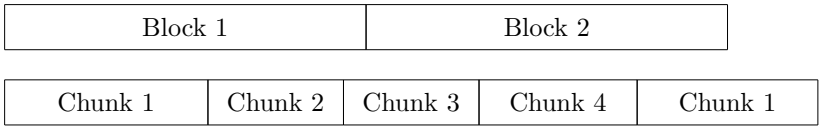


Figure 3.4: Illustration of the persistent chunk index and the auxiliary chunk index



(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

Figure 3.5: Example of a block mapping

3.5 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 that from the most recent version of the block data. We call such a mapping the "block mapping".

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 3.5 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). Figure 3.6 shows how the chunk index and the block index data is stored on disk.

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. This reduces the meta data overhead and also decreases the number of block index reads. Write and reads requests smaller than the block size are possible, but multiple accesses to the same block are serialized using a reader-writer locking scheme.

As the chunk index, the block index consists of a persistent and an in-memory index. As persistent data structure, we use the Tokyo Cabinet implementation of a B+ tree [37]. In a backup scenario, we assume largely sequential access so that each B+ tree read operation fetches multiple consecutive block mappings into the memory, which are likely

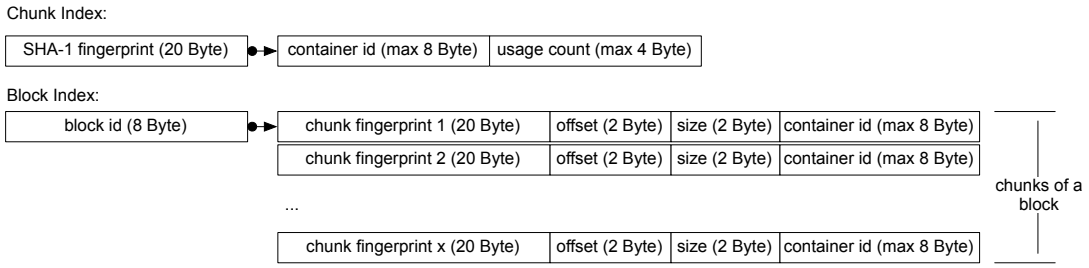


Figure 3.6: Chunk index and block index data structures (using SHA-1 for clarity)

to be used by later requests. 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.

3.6 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.

The log triggers certain events in the system and a) writes the events to an operations log and b) notifies the other system components directly of the event. The most important event types that are logged are:

Container Commit:
: A container commit event denotes that the system guarantees that the chunk data stored inside a particular container is persistently written to disk. Any chunk and block information that relies on chunk data stored in a container that is not yet committed is now allowed to be stored permanently. After the other components have received a container commit event, this data is free to be stored on disk, too. Observers of this event are the block index and the chunk index.

Block Mapping Written:
: After a block mapping has been updated during a write request, the modified

version and the original version of the block mapping are logged. The block index uses the event to restore the up-to-data version of the block index if the system has crashed. The garbage collection uses the event to update the chunk usage count information.

The entries of the event log are eventually sent to the component a second time when they are replayed. This happens on the one hand if the system crashes and the components recover their state using the operations log (crash replay) or if the system is idle (background replay). If an event is replayed at least once, it is deleted from the log.

3.7 Garbage Collection

While a complete description of the garbage collection process is not the focus of this paper, we only give a short description.

A background thread observes, using the operations log, which blocks are written and calculates the difference between the previous block mapping and the current block mapping. The difference denotes which chunks were used in the previous block and which are no longer in the current block or which chunks are now referenced more often than before. With this information, the background thread updates the chunk usage count based on this data.

If a chunk has a usage count of zero, the chunk is marked as a "garbage collecting candidate". We cannot be sure that the chunk is really not used at this point, because the system is still processing the log and the chunk might be referenced afterwards and because storage requests might be processed concurrently to the garbage collection. To overcome this, chunks without references are rechecked when the system is completely idle and the operations are fully replayed. If a garbage collecting candidate has still a usage count of zero, the chunk is flagged as deleted. Nevertheless, the data is not overwritten immediately.

The chunk store has a separated garbage collection strategy that registers the deletion and eventually merges multiple containers with deleted entries into a new container. The selection strategy for merging containers is still on-going research. A greedy strategy is to merge random containers that are filled less than 50%. If the containers have had the ids 1000 and 1010 before being merged, the new container has both ids assigned so that requests to container 1010 still have access to the container with the data of 1010, but at that time merged with the data of 1000. This merging concept has the advantage that the chunk index and the block index never need to be updated. A storage address that is valid at one point will be valid forever.

4 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, number of clients, and other configuration parameters that are of interest.

4.1 Evaluation Methodology and Environment

Often the evaluation of existing deduplication storage systems is based on measurements of operational systems [2, 4, 14]. It is hard to compare the results since these measurements are not repeatable. Only Zhu et al. also used a synthetic benchmark to evaluate their caching techniques [2]. However, even that benchmark is not available in public. As long as no agreed deduplication benchmark exists, the best we can do is to setup our own benchmarking environment.

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.

Based on this real data, we generated the changes that are introduced within a week for the second generation traffic data. Based on traces used in a recent study [1], we have calculated the run-length distribution of unique data (U), internal redundant data (IR) and temporal redundant data (TR). The run length of data in a given state is the length of a data block in which all data is detected to be from the same state. The run-length distribution of a state is the empirical probability distribution that describes the randomly chosen length of data block in the state. We also extracted the empirical probability distribution to switch from one of these three states to any other from the trace data.

Using these probabilities, we generated a synthetic second generation of traffic data that resembles the characteristics of real world data. Given an initial random state (U, IR, TR), we at first assign a randomly chosen – according to the matching probability distribution – size of the next data range and generated the data for this range. For example, to generate temporal redundant data (TR state), we copy data from the previous generation traffic data. Next, we randomly chose the next state, also according to the observed probability distribution. We repeat this until the chosen amount data is generated (here: 128 GB).

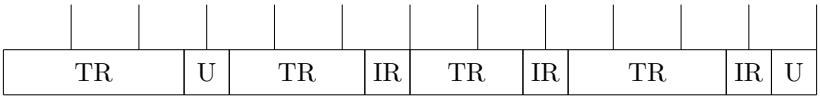


Figure 4.1: Illustration of the data pattern of the second backup generation. TR = temporal redundant data, IR = internal redundant data, U = unique data. The length of the blocks are randomly chosen according to a probability distribution extracted from real world data.

Figure 4.1 illustrates the data pattern for the second generation. A major advantage of such an approach is the scalability. We are able to generate arbitrary amounts of data for an arbitrary number of data generations.

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).

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. The SSDs and the SAN are used with the ext3 file system, mounted using the `noatime`-flag. The filesystem on the SSD is formatted using a 1 KB block size. The server uses Linux operating system with Ubuntu 9.04 using a 2.6.28 Linux kernel with SCST patches and SCST 1.0.1.

Up to four worker nodes are concurrently writing backup data to the deduplication system. The worker nodes are 4-core Xeon servers, with 12 GB RAM, a 1 GB network interface, and two 500 GB hard disks where the operating system is installed on the first and the traffic data is stored on the second hard disk. The 128 GB data is spitted into four tar files so that each worker node holds a 32 GB portion of the overall data. The evaluation setup is illustrated in Figure 4.2.

The base 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. Additionally, we allow in the base configuration, to hold all written block index data in-memory before it is written back. The only filter we use until otherwise noted is the chunk index filter. The block size is set to 128 KB.

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. Here we refer to the steady state as the interval where all worker nodes are

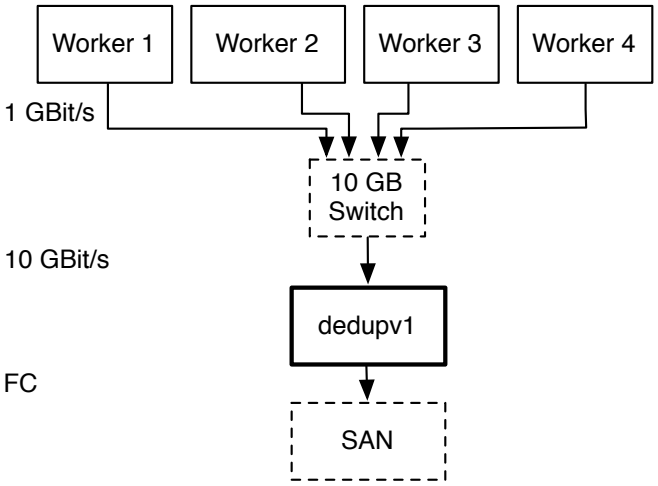


Figure 4.2: Evaluation setup

actively writing data. This is important because a newly restarted system take some time to initialize its caches (e.g. the container read cache). Also before the steady state, a fast node might write while other nodes are still pre-fetching traffic data. After the steady state, the fast nodes have already written all traffic data (32 GB per node), while a slow node is still writing.

We restarted the system and cleared all OS caches and the cache of SAN before each measurement and between the first and second generation run. We also limited the available main memory capacity to 8 GB.

4.2 Results: Index Storage System

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.

Figure 4.3 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

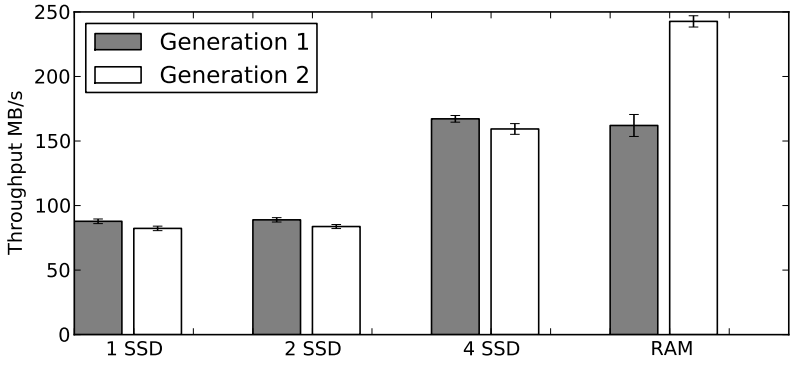


Figure 4.3: Throughput using different index storage systems

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 4.4. The figure shows the shares of different system components on the overall wall clock time on the data path. The profiling data also clearly shows that in none of the configurations the CPU-intensive chunking and fingerprinting (shown in the Figure as “Chunking”) is the bottleneck of our system. In all SSD-based configurations, the chunk index is the major bottleneck. In the RAM-based system, other bottlenecks become dominant: The disk-based block index and the storage component. “Lock” denotes the time that a request thread is blocked in order to avoid concurrent accesses to the same block. We believe that further investigations into these bottlenecks might help to further improve the performance.

We do not show the full results for a completely disk based system as the system relies heavily on the characteristics of solid-state drives. If we do store all data (chunk index, block index, chunk storage and operations log) on the SAN, we measured only

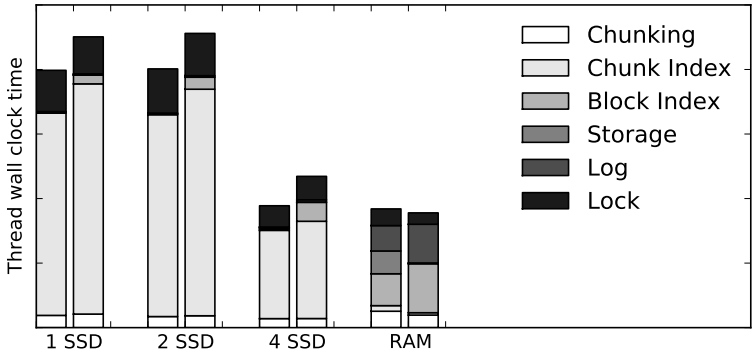


Figure 4.4: Average ratios of system components on the overall runtime in the data path. The left bars denotes the first generation runs, the right bars denote the second generation runs

a throughput of 12.9 MB/s for the first and 13.3 MB/s for the second generation. This results confirms to the results and estimates published before [2, 4].

4.3 Results: Client Node Count

In this section, we evaluate how the number of clients influences the throughput of our system. The results with a single, two, and four clients are shown in Figure 4.5. With a single client, the system is clearly limited by the 1 GB network interconnect. The throughput is 83.0 MB/s (+/− 1.2) and 84.1 MB/s (+/− 2.0 MB/s). For a single client system even a one or two SSD system would provide enough performance. In contrast to that 4 clients increase the throughput only by 16% and 8% compared to the 2 worker performance of 145.0 MB/s (+/− 3.0 MB/s) and 148.7 MB/s (+/− 4.2 MB/s).

4.4 Results: Chunk Sizes

In this section, we evaluate the system with a larger chunk size. A larger chunk size has two effects. On the one hand, it reduces the number of requests performed on the (chunk) index. On the other hand, larger chunk sized lead to a decreased redundancy. Figure 4.6 clearly indicates that – from a throughput perspective – 16 KB chunks are better. In the first generation the 16 KB chunks configurations has a throughput of 209.4 MB/s (+/− 13.3 MB/s) and in the second generation it has 250.5 MB/s (+/− 11.4 MB/s). This improvement can be explained by looking at how much data is additionally classified as new and written to disk. In the first generation, the 8 KB chunks variant classified 87 GB of the 128 GB working set as unknown (deduplication ratio of 32%). When the chunk size is doubled, only 3 GB more are classified as unknown (deduplication ratio of 29%). In the second generation, on average 14.8 GB are classified as unknown (deduplication

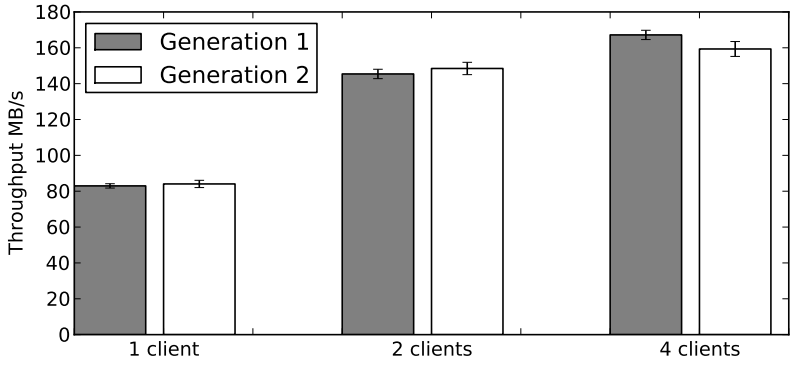


Figure 4.5: Throughput using base configuration (4 SSDs) with a varied number of client nodes

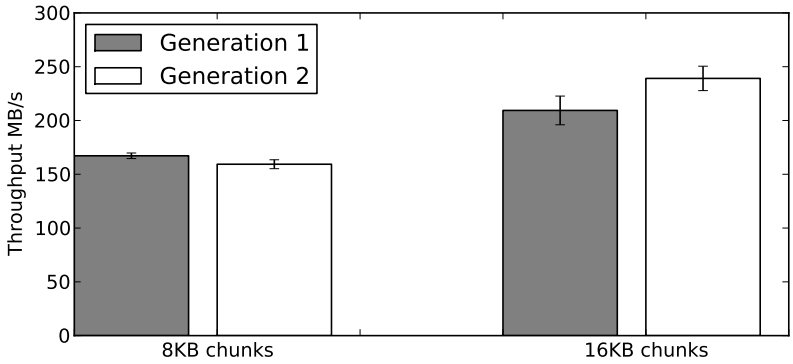


Figure 4.6: Throughput using an average chunk size of 8 KB (base configuration) and 16 KB

ratio of 89%) with 8 KB chunks while the system stores 25.8 GB when 16 KB chunks are used (deduplication ratio of 80%).

4.5 Results: Auxiliary Index Size

Up to now, we have assumed that all new chunk and block data of a run can be hold in RAM and is written to disk after the backup run itself finished. This is advantageous as we so can avoid slow random write operations during the backup run. However, in some large-scale settings this might not be possible – especially in the first backup generation where a lot of unknown data must be processed. We additionally performed measurements allowing only extremely small auxiliary indexes. We limit the auxiliary block index to 1K block mappings in RAM and the chunk index to 16K chunks in RAM.

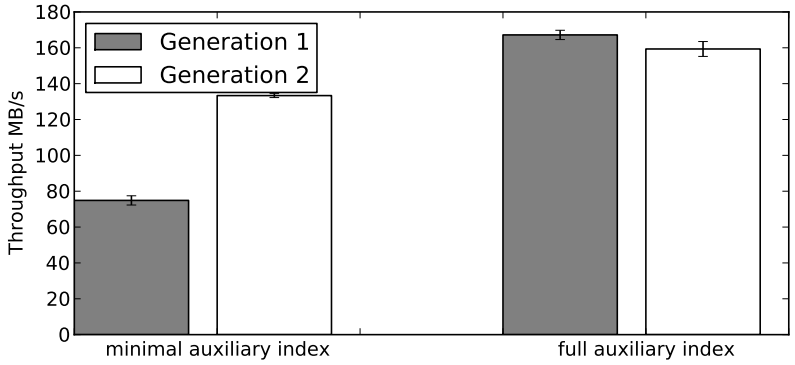


Figure 4.7: Throughput using a auxiliary index that large enough for all run data (full auxiliary index) and with a minimal auxiliary index (block index: max 1 K items, chunk index: max 16 K items)

If the size of an index grows larger, multiple background threads write the data to disk. It should be noted that even in this configuration the auxiliary indexes store data that is not allowed to be written to disk because the container data is not yet committed.

The results are shown in Figure 4.7. As expected, the throughput decreases in the first generation run due to the high amount of new chunk data that has to be merged into the persistent chunk index. On average, the system processes 74.9 MB/s (+/− 2.6 MB/s) using a minimal chunk index in the first generation in contrast to 167.2 MB/s (+/− 2.6 MB/s) with a full chunk index. However, post-processing time needed is reduced from on average 60 minutes to 2 minutes. In the second generation, the throughput reduces by 13% from 159.3 MB/s (+/− 4.2 MB/s) to 133.3 MB/s (+/− 1.1 MB/s) and the post-processing time reduces from 10 minutes on average to 1.2 minutes.

4.6 Discussion

It is hard to compare different throughput results for deduplication systems, because there is no unified test-set. However, the performance results of our SSD-based deduplication system are promising.

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 [14] based on similar assumptions as Zhu et al.. We achieve more

than 160 MB/s without depending on locality.

Lui et al. achieved a throughput of 90 MB/s with one storage node (8 cores, 16 GB RAM, 6 Disks) and an archival server (4 cores, 8 GB RAM) [24]. It is not clear how the disk bottleneck has been handled, but we assume that chunk lookups are directly answered from cache.

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.

5 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. If a larger chunk size is used, even more than 200 MB/s are possible. 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.

A future research focus may lie on the long-term behavior of deduplication systems considering aging effects as well as compression approaches for the block index and further scaling aspects.

Bibliography

[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] Fusion-IO, "iodrive spec sheet," http://www.fusionio.com/PDFs/Fusion\%20Specsheet.pdf, 2008.

[9] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 88–93, April 2007.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, May 2008.

[17] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (COM '00)*. London, UK: Springer-Verlag, 2000, pp. 1–10.

[18] F. Douglis and A. Iyengar, "Application-specific deltaencoding via resemblance detection," in *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003, pp. 113–126.

[19] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similartiy based deduplication system," in *Proceedings of 2nd The Israeli Experimental Systems Conference (SYSTOR'09)*, May 2009.

[20] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep Store: An archival storage system architecture," in *Proceedings of the 21st International Conference on Data Engineering*, 2005, pp. 804–8015.

[21] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of the annual conference on USENIX Annual Technical Conference (USENIX '04)*, 2004, p. 6.

[22] L. L. You and C. Karamanolis, "Evaluation of efficient archival storage techniques," in *Proceedings of 21st IEEE/NASA Goddard MSS*, 2004.

[23] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, 2008, pp. 12–17.

[24] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D.-S. Wang, "ADMAD: Application-driven metadata aware de-deduplication archival storage systems," in *Proceedings of the 25th IEEE Conference on Mass Storage Systems and Technologies (MSST' 08)*, 2008.

[25] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in Windows 2000," in *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, 2000, p. 2.

[26] B. Hong and D. D. E. Long, "Duplicate data elimination in a san file system," in *Proceedings of the 21st IEEE Conference on Mass Storage Systems and Technologies (MSST '04)*, 2004, pp. 301–314.

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

[28] T. Fujita and M. Christie, "tgt: Framework for storage target drivers," in *Proceedings of the Linux Symposium*, July 2006.

[29] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: making backup cheap and easy," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 285–298, 2002.

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

[31] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Symposium on Operating Systems Principles*, 2001, pp. 174–187.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley Professional, January 1995.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] M. Hirabayashi, "Tokyo cabinet: a modern implementation of dbm," http://1978th.net/tokyocabinet/, October 2009.