# A Survey and Classification of Storage Deduplication Systems

JOÃO PAULO and JOSÉ PEREIRA, High-Assurance Software Lab (HASLab), INESC TEC
& University of Minho

The automatic elimination of duplicate data in a storage system, commonly known as deduplication, is increasingly accepted as an effective technique to reduce storage costs. Thus, it has been applied to different storage types, including archives and backups, primary storage, within solid-state drives, and even to random access memory. Although the general approach to deduplication is shared by all storage types, each poses specific challenges and leads to different trade-offs and solutions. This diversity is often misunderstood, thus underestimating the relevance of new research and development.

The first contribution of this article is a classification of deduplication systems according to six criteria that correspond to key design decisions: granularity, locality, timing, indexing, technique, and scope. This classification identifies and describes the different approaches used for each of them. As a second contribution, we describe which combinations of these design decisions have been proposed and found more useful for challenges in each storage type. Finally, outstanding research challenges and unexplored design points are identified and discussed.

## 1. INTRODUCTION

The automatic removal of duplicate data has been used for a long time in archival and backup systems [Bolosky et al. 2000; Quinlan and Dorward 2002; Cox et al. 2002] and recently has become a feature of several storage appliances [Zhu et al. 2008; Aronovich et al. 2009]. With the unprecedented growth of stored digital information, duplicate data is receiving increased attention. The amount of digital information generated in 2007 was approximately 281 exabytes, and in 2011 this amount was expected to be 10 times larger than in 2006 [Chute et al. 2008]. Undoubtedly, current usage patterns mean that multiple copies of the same data exist within a single storage system, namely when multiple users of public cloud infrastructures independently store the same files, such as media, emails, or software packages.

We define *deduplication* as a technique for automatically eliminating coarse-grained and unrelated duplicate data. Unlike traditional compression techniques that eliminate intrafile redundancy or redundancy over a small group of files, typically stored together in the same operation, deduplication aims at eliminating both intrafile and interfile redundancy over large datasets, stored at different times by uncoordinated users and activities, and possibly even across multiple distributed storage servers [Douglis et al. 2004].

Distinct storage environments have specific requirements, so deduplication systems must be designed accordingly. For instance, archived data is expected to be immutable and, in some cases, never deleted. In addition, latency is typically less important than throughput, as data will be restored only sporadically whereas large quantities of data must be archived in a limited time window [Quinlan and Dorward 2002]. On the other hand, restore operations from backups will be more frequent and old data can be deleted, so efficient reference management and garbage collection mechanisms are needed [Zhu et al. 2008; Guo and Efstathopoulos 2011].

Deduplication of *active* data in primary storage changes many of the assumptions made by backup and archival infrastructures. Active data is no longer immutable, having a profound impact on reference management and garbage collection mechanisms. Additionally, applications using primary storage have stricter requirements for the latency of disk I/O operations, limiting the overhead that deduplication may introduce in the I/O critical path. Moreover, read requests are very frequent and must be served more efficiently than the restore operations of backup and archival storage [Hong and Long 2004; Srinivasan et al. 2012].

Duplicate pages in random access memory (RAM) are already avoided in modern operating systems with process forking and shared libraries, but virtualization does not take full benefit of these approaches. In fact, duplicate pages are not shared even when several virtual machines (VMs), with the same operating system kernel and shared libraries, are running in the same host. Deduplication can be used to eliminate the redundant pages across these VMs [Waldspurger 2002]. Currently, memory is only deduplicated across VMs in the same host, so the ability to scale to larger cluster infrastructures is not a concern, unlike in other storage environments. Instead, RAM requires deduplication mechanisms that maintain strict I/O performance and use space-efficient metadata.

Deduplication is also applied within solid-state drives (SSDs) not only to increase the useful storage space of these devices but also their lifespan. Since duplicate write requests can be intercepted and shared before actually being stored, it is possible to reduce the writes to the disk and, consequently, the FLASH erase operations that limit the drive's lifespan [Chen et al. 2011]. SSD deduplication systems have a constrained amount of memory space for metadata and must achieve a minimum impact on I/O performance.

The effectiveness of deduplication is measured by the *deduplication gain*, defined as the amount of duplicates actually eliminated, and thus the reduction of storage space required to persist the same data. As an example, deduplication can reduce storage size by 83% in backup systems and by 68% in primary storage [Meyer and Bolosky 2011]. Cloud computing and particularly virtualized commodity server infrastructures bring novel opportunities, needs, and means to apply deduplication to system images stored in general purpose storage systems. Namely, a gain of 80% is achievable for VM volumes with general purpose data [Jin and Miller 2009; Clements et al. 2009]. RAM used by virtualized hosts can be reduced by 33% [Waldspurger 2002], and the storage space of SSDs can be reduced by 28% [Chen et al. 2011].

The storage space spared by deduplication reduces infrastructure costs and can also be used to improve storage reliability with additional RAID configurations.
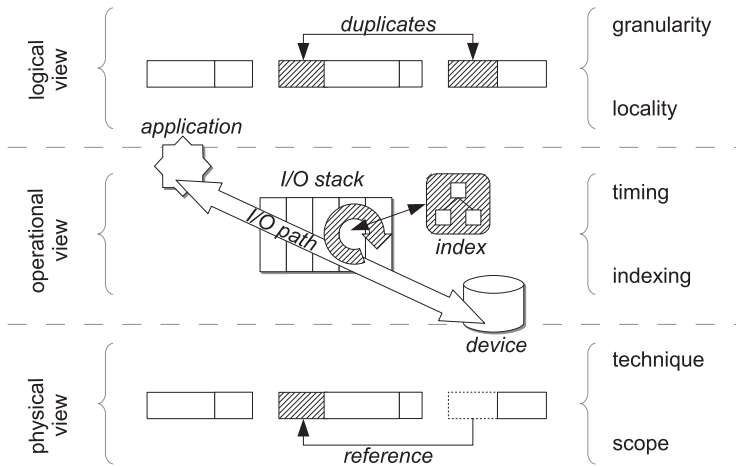
Fig. 1. Views of deduplication and key design issues.

Deduplication might also have a positive performance impact throughout the storage management stack, namely in cache and I/O efficiency [Koller and Rangaswami 2010] and network bandwidth consumption, if deduplication is performed at the client side and only unique data is sent to the storage server [Muthitacharoen et al. 2001].

## 1.1. Design Issues

Deduplication can be regarded as bidirectional mapping between two different views of the same data: a logical view containing identifiable duplicates and a physical view as stored in actual devices from which duplicates have been removed. The mapping process is embodied in the I/O path between applications that produce and consume the data and the storage devices themselves. Figure 1 depicts each of these views and identifies key issues in each of them that lead to different design decisions and trade-offs.

Thus, the logical view of data in a deduplication system is a set of assumptions on the workload that determine which duplicate content is relevant and hence which duplicates exist and which should be removed. First, all deduplication systems partition data into discrete chunks that are to be compared, identified as duplicate, and eventually removed. This partitioning can be done with different granularity, using various criteria for chunk boundaries as well as for their sizes. Although *segment* is sometimes used as a synonym of *chunk*, we avoid it because it is also used in some proposals as a higher granularity unit composed by a large number of chunks and leading to ambiguity [Lillibridge et al. 2009]. Moreover, assumptions on the likelihood of duplicate chunks being found close together, both in space or time, lead to design decisions exploiting locality that influence both the efficiency and effectiveness of the deduplication process.

On the other hand, the physical view of data in a deduplication system is first and foremost concerned with the technique used on disk to represent duplicate data that has been removed, such that efficient reconstruction of the logical view is possible. Given the current relevance of distributed storage systems, a key design decision is the distribution scope of the deduplication technique. This can be defined as the ability to represent removed duplicates across different nodes such that the reconstruction of data requires their collaboration.

Finally, deduplication as a process has to be understood as happening in the context of a storage management system. This exposes an API to client applications, such as a file system or a block device, and is composed by multiple stacked software layers and processing, networking, and storage components. The key design issue here is the *timing* of the main deduplication operations, such as searching for duplicates, regarding the critical path of I/O operations. Since finding duplicates is potentially a resource intensive operation, it is invariably based on an indexing data structure that supports efficient matching of duplicate chunks. Thus, the indexing method has a strong impact not only on the efficiency of the deduplication process but also on its effectiveness by potentially trading off exactness for speed.

The main approaches for each of these design issues determine the structure of this survey. First, in Section 2, we identify the main options for each of the key design issues and summarize the main proposals that embody them, thus introducing a classification of deduplication systems according to six axes. Then, in Section 3, for each storage type to which deduplication has been applied, we identify how the distinct design issues were explored and show how the different combinations of design decisions fit specific requirements.

## 1.2. Challenges

*Overhead versus Gain*. The main challenge in deduplication systems is the trade-off between the achievable deduplication gain and the overhead on a comparable storage system that does not include deduplication. As an example, smaller chunk sizes increase the space-saving benefits of deduplication but lead to larger index structures that are more costly to maintain. Ideally, the index would be fully loaded into RAM, but for a large storage and a relatively small chunk size, the index is too large and must be partially stored on disk. This increases the number of disk I/O operations needed by deduplication, which may interfere with the foreground I/O performance [Zhu et al. 2008].

In addition, deduplication should be performed as soon as data enters the storage system to maximize its benefits. However, finding duplicates is a resource-intensive task that will impact latency if performed in the critical path of storage writes. If deduplication is removed from the critical path and done in the background, additional temporary storage is required and data must be read back from the storage to find duplicates, thus increasing the consumption of storage I/O bandwidth [Srinivasan et al. 2012].

The more data chunks are omitted, the more the physical layout of deduplicated data differs from the original layout. Namely, deduplication introduces fragmentation that deteriorates the performance of read and restore operations [Kaczmarczyk et al. 2012]. Additional metadata is also required for correctly reconstructing deduplicated data. Thus, there is additional overhead involved in maintaining the integrity of such metadata, as one must ensure that a certain shared chunk is no longer serving any I/O request before modifying or deleting it. More specifically, this requires managing references to shared chunks, which is complex and requires a garbage collection mechanism that may also impact performance [Guo and Efstathopoulos 2011].

*Scalability versus Gain.* The most gain can be obtained when any chunk can, in principle, be compared with any other chunk and be omitted if a match is found. However, such complete matching is harder as the amount of data and components in a large-scale storage system grow. Specifically, a centralized index solution is likely to become itself very large and its manipulation a bottleneck on deduplication throughput [Clements et al. 2009]. Partial indexes that can match only a subset of duplicates improve scalability but perform only *partial* deduplication. Nonetheless, the amount of chunks that

cannot be matched can be reduced by exploring data locality [Lillibridge et al. 2009] and by grouping together chunks with greater similarity [Manber 1994].

In a distributed storage system, a simple strategy for scalability is to perform deduplication independently in each node, thus having multiple independent indexes. Again, this approach allows only partial deduplication, as the same chunk might be duplicated in multiple nodes. Missed deduplication opportunities can be mitigated by targeting chunks that have a greater likelihood of containing matching data, namely by routing similar files to the same nodes [Bhagwat et al. 2009].

The trade-off between scalability and gain can be improved by using a distributed hash table (DHT) as the index that is accessed by all nodes, eliminating duplicates globally in an *exact* fashion [Dubnicki et al. 2009; Ungureanu et al. 2010]. However, a remote invocation to the index is required to find duplicate or similar chunks. If the index is accessed in the critical I/O path, which is common in many systems, it may lead to an unacceptable latency penalty.

*Reliability, Security, and Privacy.* Distributed deduplication systems must tolerate node crashes, data loss, and even byzantine failures [Douceur et al. 2002]. Eliminating all duplicate data will also eliminate all of the redundancy necessary for tolerating data loss and corruption, so a certain replication level must be maintained. Studies show that it is possible to achieve both, but few systems address these issues [Rozier et al. 2011; Bhagwat et al. 2006]. Metadata must also be resilient to failures, thus needing to be stored persistently, which also reduces deduplication space savings. Additionally, both data and metadata must be distributed in large-scale systems to tolerate single node failures while maintaining high availability.

Some deduplication systems share data from distinct clients, raising privacy and security issues that can be solved by trading off deduplication space savings [Nath et al. 2006]. Security and privacy issues are expected not only in cloud storage infrastructures but also in remote storage appliances where data from several clients is stored [Harnik et al. 2010].

## 1.3. Scope and Outline

This survey presents an extensive overview of deduplication storage systems, explaining their main challenges and design decisions while also clarifying some misunderstandings and ambiguities in this field. As a first contribution, we extend the existing taxonomy [Mandagere et al. 2008] and identify key design decisions common to all deduplication systems. For each of them, we describe the distinct approaches taken to address the main challenges of deduplication. The second contribution is the analysis of deduplication systems in four different storage environments: archival and backup storage, primary storage, RAM, and SSDs. Each storage type has distinct assumptions that impact the deduplication system's design. Finally, this survey also gives insight about the unaccounted challenges that may be interesting points for future research.

This survey is focused only on deduplication in storage systems. Namely, we do not address network deduplication [Muthitacharoen et al. 2001], although some systems we refer to do both network and storage deduplication [Cox et al. 2002]. In fact, as we explain, most systems that perform deduplication before actually storing the data can offload some of the processing to the client and avoid sending duplicate chunks over the network.

Additionally, we do not address distributed logical volume management (LVM) systems with snapshot capabilities that already avoid creating duplicates among snapshots of the same VM or VMs created from the same snapshot [Meyer et al. 2008]. Although these systems share some technical issues with deduplication, such as reference management and garbage collection, they are fundamentally different by not

addressing the removal of unrelated duplicate chunks. Finally, we do not address delta-based versioning systems where delta encoding is only done across versions of the same file [Berliner 1990; Burns and Long 1997]. We focus on deduplication systems that eliminate both intrafile and interfile redundancy over large datasets without any knowledge about data versions.

The rest of this article is structured as follows. Section 2 introduces a classification of deduplication systems according to key design decisions, discussing the distinct approaches as well as their relative strengths and drawbacks. Section 3 surveys existing deduplication systems grouped by type of storage targeted—that is, archival and backup storage, primary storage, RAM, and SSDs. This section explains how the approaches presented in the previous section suit the distinct storage environments. Section 4 discusses the current research focus and remaining challenges for deduplication, whereas Section 5 presents the final remarks of the survey.

## 2. CLASSIFICATION CRITERIA

This section introduces a taxonomy for classifying deduplication systems by expanding on previous proposals [Mandagere et al. 2008]. This classification is based on the major design decisions implicit in all deduplication systems as summarized in Figure 1: granularity, locality, timing, indexing, technique, and distribution scope.

### 2.1. Granularity

Granularity refers to the method used for partitioning data into chunks, the basic unit for eliminating duplicates. Given its importance in the overall design of a deduplication system, it has been simply referred to as the deduplication algorithm [Mandagere et al. 2008]. However, there are significant concerns other than granularity that justify avoiding such a name.

One of the most straightforward approaches is whole file chunking, in which data is partitioned along file boundaries set by a file system [Bolosky et al. 2000]. As many backup systems are file oriented, whole file chunking avoids the partitioning effort, and by doing deduplication at a higher granularity, there are less chunks to index and be processed by the deduplication engine [Policroniades and Pratt 2004].

Another common approach has been to partition data into fixed-size chunks, also referred to as fixed-size blocks or simply blocks. This is particularly fit for a storage system that already uses such partition into fixed-size blocks [Quinlan and Dorward 2002; Hong and Long 2004]. In fact, for the cases where changed data is dynamically intercepted at a small granularity, the fixed-size block approach can offer high processing rates and generate less CPU overhead than other alternatives with identical sharing rates [Policroniades and Pratt 2004; Constantinescu et al. 2011]. By adjusting the size of chunks, deduplication gain can be increased at the expense of additional overhead in processing, metadata size, and fragmentation [Policroniades and Pratt 2004; Kaczmarczyk et al. 2012].

Consider now two versions of the same file where version $A$ only differs from version $B$ by a single byte that was added to the beginning of the latter version. Regardless of files being considered as a whole or partitioned into fixed-size chunks, in the worst case scenario, no chunks from version $A$ will match chunks from version $B$. This issue is referred to in the literature as the boundary-shifting problem [Eshghi and Tang 2005].

The third option, which solves this problem, is to partition data into variable-sized chunks with boundaries set by the content itself, also called *content-defined chunking* (CDC) [Muthitacharoen et al. 2001]. The first version of the algorithm uses a sliding window that moves over the data until a fixed content pattern defining the chunk boundary is found. This approach generates variable-sized chunks and solves the issue of inserting a single byte in the beginning of version $B$. More precisely, only the first

chunk from version $B$ will differ from the first chunk of version $A$ due to the byte addition, whereas the remaining chunks will match and be deduplicated.

In this version of the algorithm, a minimum and maximum size restriction was introduced for preventing too small or too large chunks. This modification raises, once again, the boundary-shifting problem for large chunks whose boundaries are defined by the maximum size threshold instead of using the content-based partitioning. The two thresholds–two divisors (TTTD) algorithm uses two thresholds to impose a maximum and minimum chunk size, as in previous work, but also uses two divisors for defining chunk boundaries [Eshghi and Tang 2005]. The first divisor is similar to the one chosen in the original CDC algorithm, whereas the second divisor has a larger probability of occurrence. The chunk is calculated with the sliding window as in the original algorithm; however, whenever the second divisor is found, the last occurrence is registered as a possible breakpoint. When the maximum size of a chunk is reached, meaning that the first divisor was not found, then the chunk boundary is defined by the last time the second divisor was found in the chunk. Therefore, the probability of occurring the boundary-shifting problem is significantly reduced.

The previously mentioned algorithms produce variable-sized chunks within a predefined size, yet other algorithms increase the variability of chunk size to reduce metadata space without losing deduplication gain. Fingerdiff is a dynamic partitioning algorithm that creates large chunks for unmodified regions of data, which cannot be shared, and smaller chunks (subchunks) for changed data regions to increase space savings [Bobbarjung et al. 2006]. As an example, when a new version of a previous stored file is received, subchunks will be small enough for capturing small changes in the file and sharing them, boosting space savings, whereas the unmodified data will still be stored as larger chunks, reducing indexing space costs. Two other algorithms aimed at increasing chunk size variability without significantly affecting deduplication gain were presented in bimodal CDC [Kruus et al. 2010]. The *breaking apart* algorithm divides backup data streams into large-size chunks and then further divides the chunks into smaller sizes when deduplication gain justifies it. On the other hand, the *building-up* algorithm divides the stream into small chunks that are then composed when the deduplication gain is not affected. Moreover, a variant of the breaking apart algorithm can be combined with a statistical chunk frequency estimation algorithm, further dividing large chunks that contain smaller chunks appearing frequently in the data stream and consequently allowing higher space savings [Lu et al. 2010].

Although frequently discussed together, each method described here can be combined with techniques that eliminate exact duplicates or that can cope with similar but not fully identical chunks, as in delta encoding [Quinlan and Dorward 2002; Policroniades and Pratt 2004; Nath et al. 2006; Aronovich et al. 2009]. Specifically, both aliasing and delta encoding, detailed in Section 2.5, can be applied to whole files, fixed-size chunks, or variable-sized chunks. However, the optimal chunk size is related to the technique being used—for instance, chunks in delta-encoding deduplication can be larger than in exact deduplication without reducing the deduplication gain.

## 2.2. Locality

Locality assumptions are commonly exploited in storage systems, namely to support caching strategies and on-disk layout. Similarly, locality assumptions on duplicates can be exploited by deduplication, potentially making deduplication gain depend on the workload's locality characteristics. However, there are systems that do not make any locality assumptions on duplicates [Dubnicki et al. 2009; Yang et al. 2010a; Clements et al. 2009].

*Temporal locality* means that duplicate chunks are expected to appear several times in a short time window. More specifically, if chunk $A$ was written, it will probably be

written again several times in the near future. Temporal locality is usually exploited by implementing caching mechanisms with least-recently used (LRU) eviction policies [Quinlan and Dorward 2002]. Caching some of the entries of the index in RAM can reduce disk accesses while reducing memory usage. In workloads that exhibit poor temporal locality, however, the LRU cache is inefficient and most accesses are directed to the disk index, thus creating a bottleneck.

*Spatial locality* means that chunks present in a specific data stream are expected to appear in subsequent streams in the same order. For example, if chunk $A$ is followed by chunks $B$ and $C$ in a data stream, the next time that chunk $A$ appears in another stream, it will probably be followed by chunks $B$ and $C$ again. Spatial locality is commonly exploited by storing groups of chunks in a storage layout that preserves their original order in the stream. Then, the signatures of all chunks belonging to the same group are brought to a RAM cache when one of the signatures is looked up in the disk [Zhu et al. 2008; Rhea et al. 2008]. For example, if a stream has chunks with content signatures $A$, $B$, and $C$, then these chunks and their signatures are stored together on disk. When a chunk with signature $A$ is written, the signatures for chunks $A$, $B$, and $C$ are brought to memory, because chunks $B$ and $C$ will probably appear next in the stream due to spatial locality, and additional disk accesses to the index can be avoided. Furthermore, temporal and spatial locality can be exploited together [Srinivasan et al. 2012].

## 2.3. Timing

Timing refers to when detection and removal of duplicate data are performed—more specifically, if duplicates are eliminated before or after being stored persistently. Inline deduplication, also known as in-band deduplication, is done in the critical path of write requests. This approach requires intercepting storage write requests, calculating chunk boundaries and signatures if necessary, finding a match for the chunk at the index, and, if found, sharing the chunk or delta encoding it. Otherwise, if the match is not found, the new chunk signature must be inserted in the index. Only then is the I/O request completion acknowledged.

Inline deduplication is widely used in several storage back ends [Quinlan and Dorward 2002; Rhea et al. 2008] and file systems [Zhu et al. 2008; Ungureanu et al. 2010]. In addition, inline deduplication is possible only if I/O requests can be intercepted. One of the main drawbacks of inline deduplication is the overhead introduced in the latency of write requests, as most of the processing is done in the write path. In fact, one of the major bottlenecks is the latency of operations to the on-disk index, which could be solved by loading the full index to RAM, but that does not scale for large datasets. There are some scenarios where this overhead may not be acceptable, such as in primary storage systems with strict I/O latency requirements [Srinivasan et al. 2012]. Nevertheless, there are proposals for reducing this impact with optimizations that explore locality, as discussed in Section 3.

A variant of inline deduplication in client-server systems partitions data and computes content signatures at the client side, sending first only compact chunk signatures to the server [Bolosky et al. 2000; Waldspurger 2002]. Then, the server replies back to the client, identifying missing chunks that were not present at the server storage, and must be transmitted. This way, only a subset of the chunks is sent and network bandwidth is spared [Cox et al. 2002]. This issue has been referred to as placement [Mandagere et al. 2008]; however, it is not considered in this survey as a general design decision shared by all deduplication systems.

As an alternative to inline deduplication, some systems do offline deduplication, where data is immediately written to the storage and then scanned in the background to search and eliminate duplicates. This technique is also referred to as off-band or postprocessing deduplication. Since deduplication is no longer included in the write

critical path, the overhead introduced in I/O latency is reduced. This approach requires less modifications to the I/O layer but needs additional resources to scan the storage, searching for changed chunks that need to be deduplicated. Moreover, as data is first stored and then shared asynchronously, offline deduplication temporarily requires more storage space than inline deduplication.

Scanning the storage in offline deduplication can be avoided by intercepting write requests to determine which chunks have been written and may be deduplicated. Concurrently in the background, the deduplication mechanism collects modified addresses, reads the corresponding data from the storage, and eliminates duplicates. Moreover, calculation of content signatures may be done in the write path, thus reducing the need of reading the chunk content from disk. However, this approach increases I/O latency overhead, and if chunks can be updated, a copy-on-write mechanism is necessary to prevent changing the content of processed chunks. These optimizations detect modified content without requiring a storage scan while still introducing negligible overhead in I/O operations [Hong and Long 2004; Clements et al. 2009]. Finally, in some offline deduplication systems, I/O and deduplication operations concurrently update common metadata structures, leading to lock mechanisms that result in fairness and performance penalties for both aliasing and I/O operations if implemented naively [Clements et al. 2009].

## 2.4. Indexing

Indexing provides an efficient data structure that supports the discovery of duplicate data. With the exception of some systems that index actual chunk content [Arcangeli et al. 2009], most systems summarize content before building the index [Bolosky et al. 2000; Quinlan and Dorward 2002]. A compact representation of chunks reduces indexing space costs and speeds up chunk comparison.

Summarizing content by hashing leads to identity signatures that can be used to search for exact duplicates. As a drawback, hash computation needs additional CPU resources, which may be problematic for some systems [Chen et al. 2011]. Furthermore, it is possible to compare the content of two chunks with the same identity signatures before aliasing them, thus preventing hash collisions [Rhea et al. 2008]. However, byte comparison of chunks will increase the latency of deduplication and I/O operations if deduplication is done in the storage write path, although the probability of hash collisions is negligible [Quinlan and Dorward 2002].

The similarity of two chunks can be assessed by computing a set of Rabin fingerprints for each chunk and then comparing the number of common fingerprints [Manber 1994], which we refer to as similarity signatures herein. Rabin fingerprints can be calculated in linear time and are distributive over addition, thus allowing a sliding window mechanism to generate variable-sized chunks and compose fingerprints efficiently [Rabin 1981; Broder 1993]. Comparing a large number of fingerprints to find similar chunks may present a scalability problem and require a large index, so a set of heuristics was introduced for coalescing a group of similarity fingerprints into superfingerprints. Two matching superfingerprints indicate high resemblance between the chunks, thus scaling the index to a larger number of chunks [Broder 1997].

Signatures are then used to build the indexing data structure. With a full index, all computed signatures are indexed, thus having an entry for each unique chunk at the storage. This finds all potential candidates for deduplication [Bolosky et al. 2000; Quinlan and Dorward 2002], but the size of the index itself becomes an obstacle to performance and scalability. Namely, it becomes too large to be kept in RAM, and storing it on disk has a profound impact on deduplication throughput [Quinlan and Dorward 2002].

This problem has been addressed by using a sparse index, in which a group of stored chunks are mapped by a single entry at the index. As an example, a sparse index can be built by grouping several chunks into segments that are then indexed with similarity signatures instead of identity signatures [Lillibridge et al. 2009]. Since segments are coarse grained, the size of this primary index is reduced and can be kept in RAM. Then, each segment may have an independent secondary index of identity signatures, corresponding to its chunks, that is stored on disk. When a new segment is going to be deduplicated, its similarity signature is calculated and only a group of the most similar segments have their identity secondary indexes brought to RAM. By only loading the secondary indexes of the most similar segments to RAM, deduplication gain is kept acceptable while using less RAM. There are also other proposals of sparse indexes that, for example, exploit file similarity [Bhagwat et al. 2009]. We discuss specific designs in Section 3. Sparse indexes are able to scale to large data sets but restrict the deduplication gain, because some duplicate chunks are not coalesced, performing only partial deduplication. As RAM footprint is reduced, smaller chunk sizes can be used to increase space savings while still scaling for larger datasets.

A third alternative is a partial index, in which each index entry maps a single unique chunk but only a subset of unique stored chunks are indexed, unlike in the full index approach. Therefore, the RAM utilization is always under a certain range by sacrificing space savings and performing only partial deduplication [Guo and Efstathopoulos 2011; Chen et al. 2011; Gupta et al. 2011; Kim et al. 2012]. The eviction is made based on a predefined policy, for example, by using an LRU policy or by evicting the less referenced signatures.

## 2.5. Technique

Two distinct representations of stored data that eliminate duplicate content are discussed in the literature. With aliasing, also known as chunk-based deduplication, exact duplicates can be omitted by using an indirection layer that makes them refer to a single physical copy. I/O requests for aliased chunks are then redirected accordingly.

Alternatively, delta encoding eliminates duplicate content among two similar but not fully identical chunks. Namely, only one chunk is fully stored (the base chunk), whereas the distinct content necessary to restore the other chunk is stored separately as a delta or diff. Therefore, the duplicate information is stored only once in the base chunk, and the other chunk can be restored by applying the diff to the base version.

Aliasing requires less processing power and has faster restore time than delta deduplication because no fine-grained differences need to be calculated or patched to recover the original chunk [Burns and Long 1997]. On the other hand, delta encoding saves additional space in chunks that do not have the same exact content, thus allowing the chunk size to be increased without reducing the deduplication gain [You and Karamanolis 2004; Aronovich et al. 2009]. In addition, delta encoding is performed among a pair of chunks, so it is important to deduplicate chunks with the most similar content to achieve higher deduplication factors. Therefore, the mechanism used for detecting similar chunks is key for improving space savings. Finally, the performance of delta deduplication changes with the delta-encoding algorithms used [Hunt et al. 1998].

Most storage deduplication systems use aliasing, with Microsoft Single Instance Storage (SIS) [Bolosky et al. 2000] and Venti [Quinlan and Dorward 2002] being the pioneers. On the other hand, although there are some studies regarding the efficiency of applying delta deduplication on large file collections [Ouyang et al. 2002; Douglis and Iyengar 2003], the first complete deduplication system based exclusively on delta deduplication was proposed by IBM Protect Tier [Aronovich et al. 2009]. However, there are other systems that combine both techniques by first applying aliasing, which eliminates all redundant chunks, and then delta deduplication for chunks that did not

exactly match any other chunk but could be stored more efficiently if delta encoded [You et al. 2005; Shilane et al. 2012]. Moreover, other proposals also combine chunk compression with the previous two techniques for reducing the storage space even further [Douglis et al. 2004; Gupta et al. 2010; Constantinescu et al. 2011; El-Shimi et al. 2012].

Both aliasing and delta encoding require metadata structures for abstracting the physical sharing from the logical view. For instance, many storage systems store and retrieve data at the file-level abstraction, even if files are then partitioned into smaller chunks for deduplication purposes. In these systems, it is necessary to have, for example, tree structures that map files to their chunk addresses and that must be consulted for file restore operations [Quinlan and Dorward 2002]. Other systems intercept I/O calls and deduplicate at the block-level abstraction, already having metadata for mapping logical blocks into storage addresses [Hong and Long 2004; Chen et al. 2011]. In these cases, aliasing engines must update these logical references to the same physical address, whereas delta engines must update the references to point to the base chunks and deltas. In fact, in systems where content to be read does not have an associated signature that allows searching directly for chunk addresses in indexing metadata, additional I/O mapping structures are necessary to translate read requests to the corresponding chunks. Finally, as some systems delete or modify chunks, knowing the number of references for a certain aliased or base chunk is important, because when a chunk is no longer being referenced, it can be garbage collected [Guo and Efstathopoulos 2011]. Both I/O translation and reference management mechanisms must be efficient to maintain low I/O latency overhead and reclaim free space.

## 2.6. Scope

Distributed systems perform deduplication over a set of nodes to improve throughput and/or gain while also scaling out for large datasets and a large number of clients. Unlike in centralized deduplication, some distributed deduplication systems need to define routing mechanisms for distributing data over several nodes with independent CPU, RAM, and disks. Moreover, by having several nodes, it is possible to increase the parallelism and, consequently, increase deduplication throughput while also tolerating node failures and providing high availability [Cox et al. 2002; Douceur et al. 2002; Bhagwat et al. 2009]. Other distributed systems assume nodes with individual CPU and RAM that have access to a shared storage device abstraction where nodes perform deduplication in parallel. This allows the sharing of metadata information between nodes by keeping it on the shared storage device, which otherwise would have to be sent over the network [Clements et al. 2009; Kaiser et al. 2012]. Finally, distinct nodes may handle distinct tasks. For instance, whereas some nodes partition data and compute signatures, other nodes query and update indexes, thus parallelizing even further the deduplication process [Yang et al. 2010a, 2010b].

The key distinction is the scope of duplicates that can be matched and represented after being removed. In distributed deduplication systems with a local scope, each node only performs deduplication locally, and duplicate chunks are not eliminated across distinct nodes. This includes systems where nodes have their own indexes and perform deduplication independently [You et al. 2005]. Some systems introduce intelligent routing mechanisms that map similar files or groups of chunks to the same node to increase the cluster deduplication gain [Bhagwat et al. 2009; Dong et al. 2011]. Deduplication is still performed at a smaller granularity than routing and in a local fashion, thus not eliminating all duplicate chunks globally across distinct cluster nodes.

In contrast, in distributed deduplication systems with a global scope, duplicate chunks are eliminated globally across the whole cluster. In this case, an index mechanism accessible by all cluster nodes is required so that each node is able to deduplicate

its chunks against other remote chunks. Some approaches use centralized indexes that have scalability and fault tolerance issues [Hong and Long 2004], whereas other solutions use decentralized indexes but increase the overhead of lookup and update operations [Douceur et al. 2002; Dubnicki et al. 2009; Hong and Long 2004; Clements et al. 2009]. When compared to local approaches, global distributed deduplication increases space savings by eliminating duplicates across the whole cluster. However, there is additional overhead for index accesses that, for example, may impose unacceptable I/O latency in primary storage systems [Ungureanu et al. 2010].

Finally, storage systems that perform deduplication in a single node are centralized, even if they support data from a single or multiple clients [Quinlan and Dorward 2002; Rhea et al. 2008]. In a cluster infrastructure, these approaches do not take any processing advantage from having several nodes and do not eliminate duplicate chunks across remote nodes.

## 3. SURVEY BY STORAGE TYPE

This section presents an overview of existing deduplication systems, grouped by storage type, and their main contributions for addressing the challenges presented in Section 1. Moreover, each system is classified according to the taxonomy described in the previous section. As each storage environment has its own requirements and restrictions, the combination of design features changes significantly with the storage type being targeted.

### 3.1. Backup and Archival Storage

As archival and backup storage have overlapping requirements, some solutions address both [Yang et al. 2010b]. In fact, these two storage environments have common assumptions regarding data immutability and allow trading off latency for throughput. Nonetheless, restore and delete operations are expected to be more frequent from backups than from archives, where data deletion is not even supported by some systems [Quinlan and Dorward 2002]. Distinct duplication ratios are found in archival and backup production storage. For instance, archival redundancy ranges from 35% to 79% [Quinlan and Dorward 2002; You and Karamanolis 2004; You et al. 2005], whereas backup redundancy ranges from 35% to 83% [Meister and Brinkmann 2009; Meyer and Bolosky 2011].

Deduplication in backup and archival systems was introduced by Microsoft SIS [Bolosky et al. 2000] and Venti [Quinlan and Dorward 2002]. More specifically, SIS is an offline deduplication system for backing up Windows images, which can also be used as a remote install service. Stored files are scanned by a background process that shares duplicate files by creating links, which are accessed transparently by clients, and point to unique files stored in a common storage. References to each shared file are also kept as metadata on the common storage and enable the garbage collection of unused files. A variant of copy-on-write, named *copy-on-close*, is used for protecting updates to shared files. With this technique, the copy of modified file regions is only processed after the file is closed, thus reducing the granularity and frequency of copy operations and, consequently, their overhead. With a distinct design and assumptions, an inline deduplication content-addressable storage (CAS) for immutable and nonerasable archival data was introduced by Venti. Unlike traditional storage systems, data is stored and retrieved by its content instead of physical addresses, and fixed-size chunking is used instead of a content-aware partitioning, although it is possible to configure the system to read/write blocks with distinct sizes. Unique chunk signatures are kept in an on-disk full index for both systems. Since deduplication in SIS is performed in the background and at file granularity, the index is smaller and accessed less frequently, while aliasing is also performed outside the critical write path. On the other hand, Venti queries the

on-disk index for each write operation, presenting a considerable performance penalty for deduplication throughput and for write operations since inline deduplication is being performed. This overhead penalty was addressed in Venti by using an LRU cache that exploits temporal locality and disk stripping, reducing disk seeks and allowing parallel lookups.

*The Index Lookup Bottleneck.* With no temporal locality, Venti's performance is significantly affected because most index lookups must access the disk. This problem is known as the index lookup bottleneck and has been addressed by new indexing designs [Eshghi et al. 2007], by exploiting spatial locality [Zhu et al. 2008; Lillibridge et al. 2009; Guo and Efstathopoulos 2011; Shilane et al. 2012], and by using SSDs to store the index [Meister and Brinkmann 2010; Debnath et al. 2010].

Hash-based directed acyclic graphs (HDAGs) were introduced as a first optimization for representing directory trees and their corresponding files by their content together with a compact index of chunk signatures. The HDAG structures efficiently compare distinct directories to eliminate duplicates among them while the compact index representation can be kept in RAM, significantly boosting lookups. These optimizations were introduced in Jumbo Store, an inline deduplication storage system designed for efficient incremental upload and storage of successive snapshots, and also the first complete storage deduplication system to apply the TTTD algorithm [Eshghi et al. 2007].

Despite the reduction of the index size in Jumbo Store, the amount of RAM needed was still unacceptable for large storage volumes, thus limiting scalability [Lillibridge et al. 2009]. This led to designs that maintain the full index on disk, similarly to Venti, while introducing optimizations to improve the throughput of lookup operations, as in DDFS [Zhu et al. 2008]. First, a RAM-based Bloom filter is used for detecting if a signature is new to the on-disk index, thus avoiding disk lookups for signatures that do not exist. Then, spatial locality is explored instead of temporal locality. Namely, a stream-informed layout is used for packing chunks into larger containers that preserve the order of chunks in the backup stream. Then, when a specific chunk signature is looked up, all other chunk signatures from that container are prefetched to a RAM cache. Due to the spatial locality, these signatures are expected to be accessed in the next operations, thus avoiding several disk operations. Although these optimizations also consume RAM, the memory footprint is significantly smaller than the one needed by Jumbo Store. These optimizations were also explored in Foundation, where a byte comparison operation for assessing if two chunks are duplicates was introduced to prevent hash collisions and, consequently, data corruption [Rhea et al. 2008]. Each comparison operation generates additional overhead, because full chunks must be read from disk.

Spatial locality can also be exploited by using a sparse index leading to increased performance and scalability at the expense of deduplication gain [Lillibridge et al. 2009], which works as follows. A backup stream is partitioned into segments, holding thousands of variable-sized chunks, which are the basic unit of storage and retrieval. Then, a primary index with similarity signatures for each stored segment is used for comparing incoming segments with some of the most resembling stored segments, named *champions*. Identity signatures of chunks belonging to champion segments, which are stored on disk, are brought to memory and matched to the chunks of the incoming segment. This design maintains a smaller RAM footprint by only loading into memory a subset of signatures that will provide the higher deduplication gain. Moreover, as segments have a coarse granularity, the primary index can also be kept in RAM for large datasets. Finally, by grouping contiguous chunks into segments and finding duplicate chunks among similar segments, spatial locality is also exploited to

increase deduplication gain. Although this approach does not detect all deduplication opportunities across chunks from all segments, thus performing only partial deduplication, it allows the reduction of chunk sizes and, consequently, increases duplicate detection while maintaining a small RAM footprint.

Partial or sampled indexes present another solution for the index lookup bottleneck and RAM consumption issues by keeping in cache only a subset of signatures that are evicted when a certain threshold of RAM utilization is attained. As there is no full index on disk, deduplication gain is reduced, because only a few signatures are identified, which is alleviated by exploring spatial locality with the prefetching of signatures for contiguous chunks [Guo and Efstathopoulos 2011], as in DDFS. Sampled indexes are also explored in other systems that combine chunk and delta-based deduplication to achieve higher deduplication gain as follows. A cache that takes advantage of spatial locality and an on-disk full index are used for aliasing deduplication while a partial index of similarity signatures is kept in RAM for delta-encoding chunks that were not eliminated by the previous method. Multiple levels of indirection caused by delta deduplication, when introduced on a system as the one presented by DDFS, however, have a substantial impact on restore throughput [Shilane et al. 2012].

Spatial and temporal locality may be lacking in some storage workloads, thus significantly reducing the efficiency of locality-based approaches. The index lookup bottleneck can then be addressed by storing the index on SSDs, which significantly increases throughput over traditional hard disks and can scale to larger datasets than RAM indexes, as explained in dedupv1 [Meister and Brinkmann 2010]. However, fine-grained index write and update operations are not a good fit for FLASH memory that is used in SSDs. This can be solved by organizing the index as a log, appending new entries sequentially, as in ChunkStash [Debnath et al. 2010]. In fact, the challenges for key-value stores on FLASH memory are widely researched and are present in a wider range of systems unrelated to deduplication [Anand et al. 2010; Debnath et al. 2011; Lim et al. 2011; Lu et al. 2012]. Moreover, both dedupv1 and ChunkStash explore spatial locality as the original DDFS work does. Although these two systems can exploit spatial locality to reduce FLASH accesses, their designs do not depend on locality to achieve efficient deduplication throughput and gain.

*3.1.1. Distributed Deduplication.* Peer-to-peer deduplication, where backups are made cooperatively to remote nodes, was introduced in Pastiche [Cox et al. 2002]. More specifically, nodes backup their data into other remote nodes that are chosen by their network proximity and data similarity. As inline deduplication is performed and each node has its own independent CAS, it is possible to send only the new chunks over the network to the nodes with the most resembling content, reducing both network bandwidth and used storage space. Moreover, convergent encryption derives keys from content instead of using each user's encryption key. This ensures privacy while forcing duplicate chunks to have the same cypher text for deduplication. Although this technique leaks the knowledge that a particular cipher text, and thus plain text, already exists, an adversary with no knowledge of the plain text cannot deduce the key from the encrypted chunk [Douceur et al. 2002].

Since Pastiche only performs deduplication across a specific set of cluster nodes, duplicate data still exists across the cluster. This led to systems that focus on deduplication across the cluster as a whole. As a first system proposal, all cluster nodes can have access to a distributed data structure, exported as a centralized index, where unique chunk signatures are kept and mapped to specific nodes. Then, duplicate chunks can be routed to the same nodes and eliminated locally by recurring to the Microsoft SIS system, thus achieving exact cluster deduplication. This design was proposed in Farsite, an inline global deduplication system [Douceur et al. 2002]. Farsite also proposes

that some redundant chunks must be kept to ensure data reliability as well as the use of convergent encryption for protecting files from distinct users.

Moreover, DHTs can be used for indexing chunk signatures and routing requests deterministically to the correct nodes [Dubnicki et al. 2009; Wei et al. 2010]. DHTs are used for implementing a large-scale inline CAS that applies compression and erasure codes to nonduplicated chunks to further increase space savings, as in HYDRAstor [Dubnicki et al. 2009]. Moreover, chunks are also distributed by several nodes with a replication factor defined by the user to ensure reliability. Although HYDRAstor supports data deletion, this involves a complex algorithm for maintaining the references to shared blocks. A distinct system uses the DHT to distribute files by their content to specific nodes and works as follows. Locally, each file signature is compared, and if a duplicate file exists, it is shared. If a duplicate file does not exist, then file chunks are deterministically distributed over the nodes, with each node being responsible for processing all signatures with a specific prefix. This deterministic mechanism ensures that chunks are deduplicated across remote nodes and is proposed in Mad2 [Wei et al. 2010]. Mad2 also proposes a novel metadata structure for preserving spatial locality, named *hash bucket matrix*, which is combined with a prefetching cache mechanism to reduce disk accesses to the index. Finally, a RAM-based Bloom filter, similar to the one proposed in DDFS, is also implemented to avoid looking up entries at the on-disk index that do not exist.

Global distributed deduplication may also use delta deduplication exclusively instead of traditional aliasing deduplication. By using the delta technique, the chunk size can be increased while not significantly affecting gain, thus allowing a scalable RAM index of similarity signatures where the most resembling chunks are found and delta encoded against the new chunks. This novel approach was introduced by the IBM Protect Tier system, presented as a gateway solution that intercepts data from backup stream generators and writes it into an external storage [Aronovich et al. 2009]. Although specific details are not presented, several gateways can be combined to perform deduplication over a common data repository, thus allowing global distributed deduplication. As a distinct approach, the dedupv1 centralized design can be extended over a shared storage device (SAN) where several nodes have exclusive access to their own data partitions [Kaiser et al. 2012]. Nodes are seen as independent dedupv1 nodes that export their own iSCSI interface, partition data, compute hashes, and map chunk requests to the correct nodes. Each node is responsible for storing, on its own SSD partition, a range of the signatures index, and in some cases, signature lookups must be done through the network. This leads to additional network bandwidth requirements and to overhead in I/O requests that are minimized with a write-back cache and write-ahead logs. Partition owners are only changed in case of load balancing or failure recovery.

*3.1.2. Local Deduplication in Distributed Infrastructures.* Global indexes and the consequent bottlenecks can be avoided by parallel local deduplication, increasing deduplication throughput at the expense of gain [You et al. 2005; Bhagwat et al. 2009; Dong et al. 2011; Fu et al. 2012; Xia et al. 2011]. Deep Store introduced a large-scale archival storage where stored files are routed, according to their signatures, to specific cluster nodes with independent processor, memory, and disk [You et al. 2005]. In each storage node, a framework named PRESIDO divides files into variable-sized chunks and uses both aliasing and delta deduplication for locally eliminating duplicates. Another work was then proposed to route files to specific nodes according to their similarity as follows. When a file is backed up, it is sent to a backup node—for example, the one with the less load—and the file similarity and identity signatures are calculated. Then, based on its similarity signature, the chunk is routed to the correct backup node. This was

introduced by the Extreme Binning system, which proposed a two-tier index design for local deduplication [Bhagwat et al. 2009]. The first index is stored in RAM and indexes a whole file similarity and identity signature, whereas the second index is stored on disk and indexes identity signatures. When the file is routed to a node, the primary index is searched for a similar file, and if found, the file identity signature is compared to see if the whole file can be deduplicated. If not, chunk identity signatures from the most similar file are brought to memory and deduplicated against the chunks of the new file. This is another implementation of a sparse index that reduces the RAM footprint at the expense of both global and local deduplication gain. Moreover, due to the system's distributed design, several backup files can be deduplicated in parallel.

Extreme Binning relies on file similarity for achieving high deduplication ratios, whereas other solutions propose to explore spatial locality in each node. Namely, local deduplication exploring spatial locality and Bloom filters, as in DDFS, can be combined with stateless or stateful routing mechanisms at superchunk granularity [Dong et al. 2011]. In the stateless algorithm, chunks are grouped into superchunks and are then routed to the right node by their content similarity. On the other hand, the stateful routing algorithm uses information about the location of chunks and utilizes a Bloom filter to count the number of times that each chunk signature in a superchunk is stored in a given node. Then, if the node with the most chunks in common is not overloaded, thus preserving load balancing, the superchunk is routed to that node. Otherwise, the second best node in terms of space savings is chosen. The stateful routing approach chooses the best nodes to store the superchunks in terms of space savings, whereas the stateless routing approach does not need knowledge of stored chunks and uses a best-effort routing scheme with low computational requirements. The stateless routing scheme can achieve 80% of the exact deduplication values; however, this rate may drop significantly for some large datasets where the stateful approach can maintain the 80%. When compared to Extreme Binning, these approaches use a smaller routing granularity and do not highly depend on interfile similarity to achieve good deduplication ratios.

The computational and memory overhead of stateful routing at superchunk granularity can be minimized by using a probabilistic similarity metric that identifies the nodes holding the most chunks in common with the superchunk being stored [Frey et al. 2012]. This metric is based on a probabilistic set intersection. It does not require exact knowledge about chunks stored at each node to define the routing strategy that yields the most deduplication gain but has an implicit estimation error. The estimation error can then be reduced by preferentially storing superchunks in nodes with high metric similarities that were also used to store previous superchunks belonging to the same backup file, thus leveraging spatial locality.

As some workloads may exhibit poor similarity, others may have poor locality, leading to proposals that explore both simultaneously and, in this way, compensate the lack of one with the other [Xia et al. 2011; Fu et al. 2012]. More specifically, one of the approaches divides files into content-defined chunks, grouping small strongly correlated files into a single segment while dividing large files into several segments. Then a local similarity index is used for finding similar segments to deduplicate against. Similarity index size can be reduced and scalability improved by grouping small files into segments. Then, segments are grouped into locality preserving blocks that are the basic caching unit, allowing a similar approach to DDFS to exploit spatial locality. This algorithm is presented in Silo, which does not present a detailed description of the routing algorithm but is aimed at a distributed infrastructure [Xia et al. 2011]. In fact, this is addressed with Σ-Dedup where similarity-based stateful routing at superchunk granularity is presented, whereas local deduplication follows an identical approach to Silo [Fu et al. 2012].

Other cluster approaches focus on offline deduplication for splitting data partitioning and signature calculations from the global index lookup, and update operations, parallelizing both and batching accesses to the index, as in DEBAR [Yang et al. 2010b] and ChunkFarm [Yang et al. 2010a]. In both systems, the deduplication algorithm is divided in two phases. In a first phase, data is partitioned into chunks, content signatures are calculated, and a RAM cache signature is used to eliminate some of the duplicate chunks locally, without recurring to a global index mechanism. In addition, this step can be processed in parallel by several nodes and requires writing all chunks and their signatures to disk, which increases storage consumption. In DEBAR, chunk signatures are written to a disk log sorted by buckets to enable batch processing, whereas in ChunkFarm, all signatures that need to be looked up in the index are collected and sent to a metadata server that then uses hash join algorithms for both lookup and update in batch [Shapiro 1986]. Then, in a second phase, signatures are processed asynchronously, and both index lookup and insertion operations are batched, avoiding fine-grained random I/O operations. In DEBAR, the index is stored in a centralized service, whereas in ChunkFarm, it is not specified if the metadata server is distributed or how that can be accomplished. In fact, a centralized index may pose as a single point of failure and scalability bottleneck in large clusters.

*3.1.3. File Systems for Archive and Backup.* Finally, archival and backup CAS systems can be extended with file system semantics at the expense of moderate I/O performance. A distributed file system built on top of HYDRAstor, which performs inline global deduplication at content-defined chunk granularity, is proposed in HydraFS [Ungureanu et al. 2010]. HydraFS optimizations allow the system to perform well for stream I/O operations (sequential read and writes), whereas single random block write and read requests, common in most file systems, are supported but are inefficient due to the FUSE abstraction layer and the backup-oriented implementation of HYDRAstor. Similar negative impact in I/O performance was also observed when building a file system on top of Venti [Liguori and Van Hensbergen 2008]. Archival and backup deduplication has a large pool of research work and is still being actively researched, mainly on the optimization of both deduplication and I/O throughputs. Nevertheless, most systems that fit in this category assume that data is immutable and I/O throughput is preferred over I/O latency. These assumptions do not hold true in primary storage systems and explain why building file systems over backup and archival deduplication systems have poor performance for random I/O operations.

## 3.2. Primary Storage

With cloud computing, there has been a growing interest in live volume deduplication. Primary storage deduplication invalidates some of the assumptions made by archival and backup deduplication systems. Applications using primary storage have strict performance requirements for I/O latency, meaning that a deduplication system must aim at the same I/O performance as a raw storage system without deduplication. Data is no longer write-once and is expected to be modified frequently, thus requiring, for some systems, copy-on-write mechanisms for preventing updates on aliased data, and efficient reference management mechanisms for tracking chunk references that will change frequently [Hong and Long 2004]. In primary storage, the percentage of duplicate data is usually lower than the one found for backup storage, dropping from 83% to 68% [Meyer and Bolosky 2011]. Other studies show that higher deduplication ratios can be found for general purpose primary VM volumes in large clusters, where 80% of space reduction is possible [Jin and Miller 2009; Clements et al. 2009].

*3.2.1. Offline Deduplication.* The strict latency requirements of primary storage applications shift the focus to offline deduplication systems, aiming at lower latency by

reducing overhead in the write path. Distributed offline deduplication for a SAN file system was introduced in the duplicate data elimination (DDE) system [Hong and Long 2004] and works as follows. Deduplication is performed in the background outside the critical write path, which, besides reducing latency, allows temporary disabling of deduplication when the system has a higher load. DDE is implemented over the distributed IBM Storage Tank system, avoiding much of the cross-host communication overhead of systems as Farsite. Clients calculate the signatures for fixed-size chunks written to the storage and send these signatures to a server that shares duplicate chunks asynchronously. The index of unique signatures is stored on the SAN and has two versions. One is structured to favor sequential I/O and spatial locality. The other is indexed by partial bits for facilitating random searches. Copy-on-write is used to ensure that chunks are not changed and the signatures are always valid, as it would otherwise lead to data corruption. Reference counts are stored in separate metadata structures used for garbage collection. Deduplication is restricted to within a specific file set that is a subset of the global file system. This policy allows distinct file sets for applications with different performance assumptions, as some may not tolerate the performance penalty introduced by deduplication.

A proposal for distributing the centralized metadata server in DDE, which is solely responsible for detecting and coalescing duplicates and presents a single point of failure, was then introduced in DeDe [Clements et al. 2009]. Namely, an offline distributed deduplication algorithm for VM volumes on top of VMWares's VMFS cluster file system is described and uses an index structure, stored at the cluster file system that is accessible to all nodes and protected by a locking mechanism. This allows efficient batch lookups and updates, whereas index sharding across multiple nodes enables the design to scale out. Moreover, VMFS simplifies deduplication, as it already has explicit block aliasing, copy-on-write, and reference management operations, which are not commonly exposed in most cluster file systems, and are used to implement the atomic share function that verifies the content of two fixed-size blocks and replaces them with one copy-on-write block if they match. However, there are alignment issues between the block size used in VMFS and DeDe, implying additional translation metadata and a consequent impact on performance. Additionally, the impact of copy-on-write operations is also significant, and therefore the DeDe algorithm is intended to run in periods of low I/O load.

A mechanism for reducing the frequency and, consequently, the overhead of copy-on-write operations was proposed in the Microsoft Windows Server 2012 centralized offline deduplication system [El-Shimi et al. 2012]. Namely, only files that meet a certain policy—for instance, file age greater than a certain threshold—are deduplicated. This reduces the probability of rewriting highly volatile files and, consequently, invoking copy-on-write operations. Moreover, it was proposed that files are grouped according to their file extensions, for example, and then each group can be deduplicated individually. This allows loading the index of each group into RAM and performing efficient lookup and update operations. A reconciliation algorithm for eliminating duplicate data between several groups can be used to achieve exact deduplication. Mechanisms for exploring spatial locality similar to DDFS, and to store the index as a log structured file and, possibly, on SSD, as in ChunkStash, were also proposed.

*3.2.2. Inline Deduplication.* Performing deduplication in an in-line fashion requires costly lookups in the write path, which can impose a significant overhead in I/O latency. On the other hand, offline deduplication may introduce additional reads from the storage, require more storage space, raise concurrency issues, and increase the complexity of the deduplication systems. These problems motivated the emergence

of inline deduplication systems for primary storage that introduce optimizations for significantly reducing the I/O latency.

In ZFS, optimizing for reduced latency means fully loading the index in RAM. It is still possible to cache only a subset of the index in memory, but disk lookups have a significant impact on deduplication and, consequently, storage I/O performance [Wright 2011]. As an alternative to maintaining the full index in memory, a multilayer Bloom filter for speeding lookups at the index and reducing the impact in I/O latency was proposed in DBLK [Tsuchiya and Watanabe 2011]. The first Bloom filter layer detects if a certain signature might be present at the on-disk index without requiring a disk access. Then, if the signature is likely to be already indexed, the second Bloom filter layer narrows the location of the signature, thus speeding its retrieval. Another solution for increasing deduplication throughput and reducing I/O latency is to use a Bloom filter and explore spatial locality by preserving the disk layout, then prefetching contiguous chunk signatures to cache as in DDFS. These two improvements were presented for an inline centralized deduplication system along with a novel fault-tolerant journaling mechanism for tracking system transactions, and recovering data and corresponding signatures in failure scenarios [Ng et al. 2011].

Finally, spatial and temporal locality were also explored to minimize both read and write latency. Fragmentation and, consequently, read I/O latency is reduced by deduplicating groups of contiguous blocks and storing them together to preserve their layout. Spatial locality makes it likely that duplicates can still be found in such groups. On the other hand, temporal locality is exploited with an LRU cache that stores a partial list of index entries and allows some lookups without disk I/O. The on-disk index is organized in buckets and is stored as a red-black tree that reduces the number of pointers and, consequently, the storage space of traditional hash tables. This space reduction allows to increase the number of buckets to improve hash lookup speed. These optimizations were presented in Netapp's iDedup inline deduplication system, where overhead is minimized while maintaining considerable deduplication gain [Srinivasan et al. 2012]. However, this design highly depends on both spatial and temporal locality. Finally, fragmentation is also present in offline deduplication systems, as it is a consequence of aliasing duplicate data and changing the original physical layout.

### 3.3. Random Access Memory

Deduplication is also used for applications and VMs that do not benefit from classic RAM sharing provided by process forks or shared libraries. The extra memory, obtained by eliminating redundant memory pages, can be useful for launching additional applications or VMs, or it can be provided to the existing ones. More precisely, it is possible to reduce memory consumption by 33% by using RAM deduplication across a group of virtualized hosts [Waldspurger 2002]. RAM deduplication systems have different assumptions from backup and primary storage systems. For instance, since duplicate content is only found across applications or VMs running in the same server, deduplication is always performed locally in a centralized fashion. Moreover, memory pages are highly volatile and change frequently, requiring efficient copy-on-write and reference management mechanisms.

The Disco virtual machine monitor (VMM) pioneered inline transparent page sharing for memory pages from different VMs [Bugnion et al. 1997]. Memory pages loaded from a special copy-on-write disk are shared, thus eliminating the need for content-aware deduplication. Disco intercepts read requests from copy-on-write disks and finds if the page for that request is already in memory, thus avoiding the disk access and sharing pages between different VMs. Duplicate pages are mapped into a single memory page at the host's main memory. Since content-aware deduplication is not used, an index of signatures is not necessary.

*3.3.1. Nonintrusive Scan Deduplication.* Disco requires several modifications to the guest OS, which led to content-aware approaches, such as VMware ESX, that perform non-intrusive memory scans to find duplicates [Waldspurger 2002]. Namely, memory can be shared in an offline fashion by periodically scanning all memory pages, calculating their signatures, and looking for duplicate pages in an index of unique page signatures. Then, shared pages must be marked as copy-on-write for preventing updates that may cause data corruption.

Considering that sharing highly volatile pages significantly increases the amount of copy-on-write operations and, consequently, overhead, a double tree index was proposed in Linux KSM to detect such pages and avoid sharing them [Arcangeli et al. 2009]. Memory regions to be deduplicated are given as a parameter and are periodically scanned for finding and merging duplicate pages among applications and KVM VMs. A stable tree metadata structure is used for registering shared pages that are write protected and scanned first for duplicate detection. An unstable tree is used for keeping unique pages that are not write protected and scanned only if no duplicates were found at the stable tree. This mechanism detects what pages are less susceptible to be rewritten in the near future and are better sharing candidates. KSM index trees do not keep hash signatures of the pages but instead keep the actual page content. This way, duplicate pages are found with the *memcmp()* operation instead of comparing content signatures.

KSM design was then updated in the Singleton system to optimize the dual page caching mechanism in virtualized hosts and further increase space savings [Sharma and Kulkarni 2012]. In virtualized hosts, each VM has its own page cache as a first-level cache mechanism, and when this mechanism fails to service an I/O request, a second-level hypervisor cache, shared by all VMs, is used to look up the requested page. Having two caches with duplicate pages occupies unnecessary memory space, and deduplication can be used to improve the cache's space efficiency. The original KSM design was modified by replacing the search trees with hash tables that index the content signatures of pages instead of their actual content, eliminating the CPU usage of *memcmp()* operations. Hypervisor cache page signatures are calculated periodically, checked against the indexes of pages from VMs' local caches, and dropped from the hypervisor cache if duplicates are found, thus improving memory usage.

Memory deduplication space savings were further improved by introducing delta encoding and compression, besides the usual aliasing technique present in previous systems. First, duplicate pages are shared by looking for duplicate signatures in an identity index. Then, a similarity index is used for looking up similar pages and delta encoding them. Pages that were not duplicated and not expected to be rewritten in a near future are compressed to improve further space savings, as introduced in the Difference Engine [Gupta et al. 2010].

*3.3.2. Intrusive Deduplication.* Other memory deduplication systems intercept disk read operations, as in Disco, but use content-aware page sharing. Intercepting requests has the advantage of detecting short-lived sharing opportunities that in scan approaches, as the one presented by the VMware ESX server, may be discarded [Milos et al. 2009]. One of the first systems to introduce memory deduplication aimed at solving problems of dynamic libraries, such as Dependency Hell and Global Offset Table (GOT) overwrite attack, by replacing dynamic libraries with static libraries, thus creating self-contained applications [Suzaki et al. 2010]. This replacement leads to extra memory usage, because static libraries do not share any data. Slinky introduces in-line deduplication as a solution for mitigating these additional memory costs [Collberg et al. 2005]. Pages are intercepted when they are being loaded into memory, and their signatures are looked up in an index to find duplicates. Content signatures are calculated before loading the

pages into memory, and copy-on-write is not necessary because only read-only pages are shared.

Unlike Slinky, some in-band systems must support mutable pages and implement copy-on-write mechanisms. This is addressed in Satori, an inline deduplication system that modifies virtual disk implementations to intercept read requests made by VMs [Milos et al. 2009]. Namely, an image of the page cache is built by observing the content of disk reads, and when a block read request is intercepted, its content is hashed and compared with the other indexed page digests to see if the page is duplicated. If the page is duplicated, then it is shared and marked as copy-on-write. For each VM, a repayment FIFO queue holds a list of volatile pages that the operating system is willing to relinquish at any time and that are used to efficiently provide free pages for copy-on-write operations.

Besides space savings, memory deduplication can also be used to implement in hardware a memory abstraction of protected shareable segments with snapshot and atomic update operations, as explained in HICAMP [Cheriton et al. 2012]. This abstraction allows fault-tolerant and safe concurrent access to data shared by multiple threads while reducing the overhead of common interprocess communication approaches. The HICAMP system is implemented as a CAS where the memory is divided into fixed-size chunks, referred to as lines, with unique and immutable content ensured by the inline deduplication algorithm. Memory space is divided into hash buckets with content lookup operations at line granularity. This way, when a new line is written, it is checked to see if a line with that content already exists and the new line can be deduplicated, or if the new line has new content and must be inserted in the memory space. Reference counting is done by the hardware, and when the number of references to a line is zero, the line is garbage collected. This design allows to implement memory segments that are logical variable-length contiguous regions of memory and are represented as direct acyclic graphs pointing to specific lines protected with copy-on-write. With this representation, snapshots can be performed efficiently, and threads can run with snapshot-isolation guarantees.

Another work that does not present an actual deduplication system but focuses on useful optimizations for RAM deduplication is the new hypercall for the XEN hypervisor, proposed to minimize the performance impact of hashing pages [Pan et al. 2011]. Moreover, the benefits of memory deduplication in virtualized clusters can be maximized by placing VMs with similar content at the same host, as described in Wood et al. [2009]. A memory fingerprinting technique that presents a compact representation of the memory content allows the identification of VMs with high page sharing potential and migrates them to the same host to achieve higher space savings.

### 3.4. Solid-State Drives

Deduplication has also been used within SSDs, known for radically improving the performance of random I/O operations. In SSDs, I/O operations are processed at fixed page sizes, usually 4KB, but unlike in traditional hard disks, it is not possible to delete data at the same granularity. Pages are grouped into erasure blocks, usually with 64 to 128 pages, and the deletion is done at erasure block granularity. Moreover, updates cannot be done in place, so modified blocks must be appended to an erasure block with free space. Then, only when all pages in an erasure block are unused can they be erased and reclaimed by a garbage collection mechanism. The number of erase operations for each block, however, is limited in the range of 10,000 to 1,000,000 operations, thus limiting the drive's lifespan and being one of the major issues of this technology [Chen et al. 2011].

Typical SSD designs include the following components. A FLASH transaction layer (FTL) is implemented in the SSD controller and emulates the behavior of a traditional

hard drive by exporting an array of logical blocks to the host. In this layer, an indirect mapping table is kept for mapping logical to physical addresses. A log-like write mechanism is used for writing pages, and each in-place update to a logical page only invalidates the previously occupied physical page, appending the new physical page to a free erasure block and updating the corresponding entry on the indirect mapping table. A garbage collection mechanism is launched periodically to recycle unused physical pages, consolidate the valid pages into a new erasure block, and clean unused erasure blocks. Wear-leveling mechanisms are used to shuffle the hot and cold blocks to balance the number of writes and deletions in erasure blocks, thus increasing the lifespan of the SSD. A certain amount of overprovisioned spare space usually exists, which is not usable by the host, and is used by the garbage collection and wear-leveling mechanisms.

Deduplication gain observed in SSDs is also lower than in primary and backup storage, ranging from 20% to 56%. A gain of 56%, however, is the best case scenario, whereas for most most workloads, only 20% to 30% is achievable [Chen et al. 2011; Gupta et al. 2011; Kim et al. 2012]. On the other hand, finding duplicate data in SSDs has advantages other than the space-saving benefits. In fact, if inline deduplication is performed, it is possible to reduce the number of writes to the storage and increase the device lifespan. Moreover, the space-saving benefits not only allow users to store more data but also provide additional space for wear-leveling and garbage collection mechanisms. SSDs already have a mapping table for translating logical to physical addresses; therefore, the deduplication design can take advantage of it for sharing identical pages with low overhead. Since there is also a periodical garbage collection mechanism, it is possible to extend it to perform reference management and garbage collection of shared pages. Deduplication in the SSD device also has some disadvantages. These devices come with an internal DRAM memory that can be used to store the deduplication index; however, this DRAM has limited space that does not allow storage of a complete index of page signatures. SSDs also come with a limited processing capability, and calculating hash signatures may impose significant overhead. These advantages and drawbacks are further discussed when we describe existing SSD deduplication systems.

A pioneer CAS for SSDs that performs best effort inline deduplication was presented by CAFTL [Chen et al. 2011]. The indirect mapping table of the SSD device is used for implementing the I/O translation mechanism for the deduplication engine. Inline deduplication is performed with a best effort approach, whereas the SSD performance is not significantly affected. If the load reaches a certain threshold, inline deduplication is turned off, and stored duplicate pages are then shared with an offline deduplication algorithm that runs in the same periods as the garbage collector. The offline approach spares storage space but does not avoid write requests to the storage. Fixed-size chunks are used and their hash signatures are stored in a DRAM partial index that only contains the most referenced signatures. An additional metadata table is used for mapping the references to each shared page and performing reference management. All of these metadata structures are stored in the SSD, because the DRAM is not persistent over reboots or power failures. However, metadata updates are buffered and flushed only when the buffer is full, leading to metadata loss when a power failure occurs. This can be solved by using a capacitor for the DRAM. Since hash calculations are heavy for the SSD built-in processor, a content-based sampling algorithm that explores data spatial locality is presented, allowing to check the probability of a set of pages being duplicates. Then, hash signatures are only calculated for pages with high probabilities that will benefit the most from deduplication.

The hash computation overhead is also avoided in CA-SD with a built-in hardware hashing unit [Gupta et al. 2011]. The mapping structures and the partial index, similar to the ones presented in CAFTL, are stored in a fast persistent storage. This persistent storage must not be the SSD device itself in order to increase SSD performance and

space savings. CA-SD proposal also differs from CAFTL, because only inline deduplication is performed. Finally, it was observed that temporal locality was present in the studied workloads, which allowed to implement the partial index of chunk signatures as an LRU cache.

Temporal locality was then further researched in subsequent work by proposing a sampling-based filter for written pages that are still in the DRAM buffer and were not flushed to the SSD yet. This sampling mechanism detects what page contents are being written more than once and will benefit more from deduplication. This way, and since the hash calculation and deduplication can introduce significant overhead in the SSD processor, only the pages that probably will achieve the most benefits from being shared are actually processed, thus reducing deduplication overhead. Moreover, this work implemented the first real prototype and evaluated it without recurring to simulation [Kim et al. 2012].

SSD deduplication is still emerging and raises interesting challenges that are not present in other storage environments. First, the limitations of DRAM space and computational power raises the need for new designs for metadata structures and hashing algorithms. SSD partial indexes decrease RAM requirements, but they also detect fewer duplicates and highly depend on locality assumptions. On the other hand, data fragmentation is not an issue, as in other storage environments, since random read requests are efficient in SSD devices. Regarding the deduplication overhead in I/O requests, the system described previously shows that for significant duplication ratios (more than 5%), the write performance can even be increased, whereas read performance has no significant overhead [Kim et al. 2012]. These values consider that efficient hashing mechanisms are being used instead of the common one presented in most SSDs.

## 4. CURRENT TRENDS AND OPEN ISSUES

Table I classifies deduplication storage systems according to the taxonomy described in Section 2 while grouping these systems by storage environment. This table highlights the relevance of each design option, characterizes the design space for each storage type, and points out unexplored designs that should be researched.

Aliasing deduplication is used in all storage types, whereas delta deduplication by itself is used only in backup systems. In both backup storage and RAM, aliasing and delta deduplication are also combined for increasing space savings. Backup storage systems use several chunk granularities, with variable-sized chunks being the most common one. Fixed-sized chunks are preferred in all other storage environments. Several different indexing designs are used in backup deduplication, including combinations of full and partial indexes [Shilane et al. 2012]. In primary and RAM deduplication systems, except in the Disco non–content-aware approach [Bugnion et al. 1997], full indexes are always used. In contrast, SSD deduplication uses only partial indexes due to DRAM space restrictions. Locality is explored in backup, primary, and SSD deduplication systems; however, in RAM deduplication, it is not assumed. In RAM and SSD storage, deduplication is performed only in a centralized fashion, whereas in backup deduplication, local and global distributed approaches are also available. On the other hand, in primary deduplication all distributed systems perform global deduplication. Finally, all storage types have systems using offline and inline deduplication, whereas in SSD deduplication, both are also combined.

Most archival and backup systems assume that stored data has a write-once policy, and in some archival systems, this data cannot be deleted at all. Some of these systems, however, can be used as back end for implementing file system syntax and, consequently, support data updating with limited performance for random I/O operations [Nath et al. 2006; Ungureanu et al. 2010]. Since deletion and update operations are expected to be less frequent than in other storage environments, reference

Table I. Classification of Deduplication Systems for All Storage Environments

| | Granularity | Locality | Timing | Indexing | Technique | Scope |
|---|---|---|---|---|---|---|
| SIS | W | N | O | F | A | C |
| Farsite | W | N | O | F | A | G |
| Venti, Liguori and Van Hensbergen [2008] | F | T | I | F | A | C |
| Foundation | F | T | I | F | A | C |
| Guo and Efstathopoulos [2011] | F | S | I | P | A | C |
| Jumbo Store | V | N | I | F | A | C |
| DEBAR, ChunkFarm | V | N | O | F | A | G |
| HYDRAstor, HydraFS, Kaiser et al. [2012] | V | N | I | F | A | G |
| Pastiche | V | N | I | F | A | L |
| DDFS, dedupv1, Chunkstash | V | S | I | F | A | C |
| Dong et al. [2011], Silo, Σ-Dedup | V | S | I | F | A | L |
| Lillibridge et al. [2009] | V | S | I | S | A | C |
| Mad2 | WV | S | I | F | A | G |
| Extreme Binning | WV | N | I | S | A | L |
| IBM Protect Tier | F | N | I | F | D | G |
| Shilane et al. [2012] | V | S | I | FP | AD | C |
| Deep Store | V | N | I | F | AD | L |
| ZFS, DBLK | F | N | I | F | A | C |
| DeDe | F | N | O | F | A | G |
| DDE | F | S | O | F | A | G |
| Ng et al. [2011] | F | S | I | F | A | C |
| iDedup | F | TS | I | F | A | C |
| MS Windows Server | V | S | O | F | A | C |
| Disco | F | N | I[2] | n/a[1] | A | C |
| Slinky, Satori, HICAMP | F | N | I[2] | F | A | C |
| VMware ESX, KSM, Singleton | F | N | O | F | A | C |
| Difference Engine | F | N | O | F | AD | C |
| CAFTL | F | S | IO | P | A | C |
| CA-SD, Kim et al. [2012] | F | T | I | P | A | C |

Granularity: (W)hole file; (V)ariable; (F)ixed. Locality: (N)one; (S)patial; (T)emporal. Timing: (I)nline; (O)ffline. Indexing: (F)ull; (P)artial; (S)parse. Technique: (A)liasing; (D)elta. Scope: (C)entralized; (G)lobal; (L)ocal.

management and garbage collection mechanisms are also active for shorter periods and their overall overhead is reduced. Moreover, in these deduplication systems, throughput is preferred over latency, and as most deduplication systems perform inline deduplication, I/O latency is significantly increased.

These assumptions are not valid in primary storage, RAM, and SSD, where I/O latency overhead must be negligible even if deduplication throughput is reduced. Thus, data is updated in place, requiring a copy-on-write mechanism to protect updates on shared data and potential data corruption. Although data is not updated in place, in SSDs it is necessary to ensure that shared blocks are not erased and collected by the garbage collector while still being referenced. In RAM deduplication, most pages are highly volatile, thus changing more often than in other storage environments, and increasing copy-on-write and reference management overhead. Finally, existing

---

[1]Disco does not require an index, as it does not perform content-aware deduplication.
[2]Disco and Satori perform inline deduplication, because I/O read requests to the persistent storage are intercepted and redirected, if possible, to duplicate shared memory pages before actually loading the pages from the storage. Slinky performs inline deduplication, because pages loaded from static libraries are shared before actually being loaded to memory.

SSD deduplication systems present deduplication embedded in the SSD device, which significantly restricts the computational power and RAM space available.

Most strikingly, inline distributed deduplication has not been explored in distributed primary storage systems that use commodity server disks instead of a shared storage. In fact, there are few proposals on distributed deduplication, and several contributions and combinations of techniques are possible. For instance, all primary deduplication systems use full index approaches, whereas partial and sparse indexing mechanisms can still be explored. An interesting challenge is finding scalable distributed solutions that obtain deduplication space savings closer to the ones achieved in distributed systems, performing global deduplication, while maintaining the high throughput achieved by distributed systems that parallelize deduplication among all nodes.

It is also important to notice that primary deduplication must be evaluated differently from backup deduplication. In archival and backup deduplication systems, static VM images, containing the operating system and application binaries, are widely used to assess the system's performance, as many of these systems are designed to store such content [Guo and Efstathopoulos 2011]. In primary storage deduplication, dynamic traces should be used to accurately simulate real workloads. In fact, modifying evaluation workloads changes data locality assumptions. For example, in a scenario where VM images with a common ancestor are backed up integrally, spatial locality can be explored efficiently. In a primary storage scenario where specific blocks of VM images are being changed independently with no correlation, for instance, in a virtual desktop infrastructure (VDI) where each user has his own workstation mapped to an independent VM, spatial locality will be significantly reduced.

Mechanisms that replay dynamic traces or recur to I/O benchmarks can be used to simulate primary storage environments if real workloads are not available. Traditional I/O benchmarks do not simulate realistic content distributions for their tests, which limits their realism. Considering that only a few proposals address this challenge [Tarasov et al. 2012; Paulo et al. 2012], novel contributions in this topic should be expected. In addition, workloads for RAM and SSD storage must be considered.

Fault tolerance and security are two other topics that deserve further attention for some deduplication systems and storage environments. In fact, the impact of deduplication in reliability was already studied, and it was shown that a level of data replication must be enforced to ensure data reliability [Rozier et al. 2011; Bhagwat et al. 2006]. Moreover, the reliability of many systems can also be improved to tolerate crash and byzantine faults. Security must also be enforced when deduplication can be performed across data from distinct users. For example, in cloud computing infrastructures, users should be able to protect their data while still allowing cloud providers to perform cross-user deduplication [Rashid et al. 2012]. Convergent encryption is commonly used for enforcing this property [Cox et al. 2002; Douceur et al. 2002], but it is also possible to propose other security mechanisms that on the one hand reduce deduplication gain, and on the other hand increase data privacy and security [Nath et al. 2006]. For instance, it is possible to achieve security models that hide the users identities from the storage providers while still enabling deduplication [Storer et al. 2008]. Finally, security has also been identified as a challenge for RAM deduplication [Suzaki et al. 2011].

## 5. CONCLUSION

Deduplication is an effective technique for reducing storage costs in distinct storage environments, more specifically in backup, primary, RAM, and SSD storage systems. Although all existing deduplication systems can be classified by a common taxonomy, each storage type has different assumptions that lead to distinct design decisions. This survey explores such taxonomy and how system designs change according to the storage environment.

As a first contribution, we present a taxonomy of existing deduplication systems according to major design decisions. More specifically, we discuss the granularity of chunks, reliance on data locality assumptions, the timing when deduplication is performed, how chunk content is indexed to find duplicates, how duplicate chunks are shared, and the distributed scope of deduplication systems.

Then, as another contribution, we survey the existing deduplication systems and classify them according to the storage type—for instance, backup, primary, RAM, and SSD. Archival and backup deduplication systems are widely explored and already exploited in several commercial storage appliances. Most of these systems assume immutable data and trade latency for deduplication and I/O throughput by mainly using inline deduplication approaches. On the other hand, in primary storage, data is mutable and I/O latency is critical, so the number of inline deduplication systems is reduced and the percentage of offline approaches raised. In RAM deduplication, most systems scan memory for duplicates to avoid intrusive mechanisms for intercepting I/O calls. Additionally, RAM pages are highly volatile, thus being updated more often than in other storage systems. Finally, in SSD deduplication, the processor and DRAM included in the devices are used for deduplication, but both have limited capabilities that significantly restrict deduplication designs.

Finally, we discuss future research directions for deduplication. Primary, RAM, and SSD deduplication systems have received less attention, and further contributions for improving deduplication throughput, reducing I/O latency, and increasing deduplication space savings can still be expected. Moreover, even in backup deduplication where the amount of work is substantially larger, these issues and others, such as reference management, scalability, reliability, and security, can be further improved. To conclude, we believe that deduplication systems and their benefits are now widely accepted by the scientific and enterprise communities, but there are still several interesting open research challenges. We envision that deduplication research will continue to grow at an accelerated pace in the forthcoming years.

## ACKNOWLEDGMENTS

## REFERENCES

Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. 2010. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, Berkeley, CA, 433–449.

Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*. 19–28.

Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. 2009. The design of a similarity based deduplication system. In *Proceedings of International Systems and Storage Conference (SYSTOR)*. ACM, New York, NY, 1–14.

Brian Berliner. 1990. CVS II: Parallelizing software development. In *Proceedings of USENIX Winter Technical Conference*. USENIX, Berkeley, CA, 341–352.

Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. 2009. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, Washington, DC, 1–9.

Deepavali Bhagwat, Kristal Pollack, Darrell D. E. Long, Thomas Schwarz, Ethan L. Miller, and Jehan Franois Pris. 2006. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, Washington, DC, 1–9.

Deepak R. Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. 2006. Improving duplicate elimination in storage systems. *ACM Transactions on Storage* 2, 4 (November 2006), 424–448.

William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. 2000. Single instance storage in Windows 2000. In *Proceedings of the USENIX Windows System Symposium (WSS)*. USENIX, Berkeley, CA, 1–12.

Andrei Broder. 1997. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*. IEEE Computer Society, Washington, DC, 21–30.

Andrei Z. Broder. 1993. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*. 143–152.

Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems* 15, 4 (November 1997), 143–156.

Randal C. Burns and Darrell D. E. Long. 1997. Efficient distributed backup with delta compression. In *Proceedings of the Workshop on I/O in Parallel and Distributed Systems (IOPADS)*. ACM, New York, NY, 27–36.

Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 77–90.

David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: Architectural support for efficient concurrency-safe shared structured data dccess. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, 287–300.

Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. 2008. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC white paper, sponsored by EMC. Retrieved September 12, 2013, from http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf.

Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. 2009. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–14.

Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. 2005. Slinky: Static linking reloaded. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 309–322.

Cornel Constantinescu, Joseph Glider, and David Chambliss. 2011. Mixing deduplication and compression on active data sets. In *Proceedings of the Data Compression Conference (DCC)*. IEEE Computer Society, Washington, DC, 393–402.

Landon P. Cox, Christopher D. Murray, and Brian D. Noble. 2002. Pastiche: Making backup cheap and easy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 1–13.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–16.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the ACM's Special Interest Group on Management of Data (SIGMOD)*. ACM, New York, NY, 25–36.

Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. 2011. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 15–29.

John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. 2002. *Reclaiming space from duplicate files in a serverless distributed file system*. Technical Report MSR-TR-2002-30. Microsoft Research. 1–14 pages.

Fred Douglis and Arun Iyengar. 2003. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 113–126.

Fred Douglis, Jason Lavoie, John M. Tracey, Purushottam Kulkarni, and Purushottam Kulkarni. 2004. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–5.

Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. 2009. HYDRAstor: A scalable secondary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 197–210.

Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. 2012. Primary data deduplication large scale study and system design. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–14.

Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. 2007. Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 123–138.

Kave Eshghi and Hsiu K. Tang. 2005. *A framework for analyzing and improving content-based chunking algorithms*. Technical Report HPL-2005-30. Intelligent Enterprise Technologies Laboratory. 1–10 pages. Available at http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.pdf.

Davide Frey, Anne-Marie Kermarrec, and Konstantinos Kloudas. 2012. Probabilistic deduplication for cluster-based storage systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC)*. ACM, New York, NY, 1–14.

Yinjin Fu, Hong Jiang, and Nong Xiao. 2012. A scalable inline cluster deduplication framework for big data protection. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. ACM, New York, NY, 354–373.

Fanglu Guo and Petros Efstathopoulos. 2011. Building a high-performance deduplication system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–14.

Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 91–103.

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2010. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM* 53, 10 (October 2010), 85–93.

Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2010. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security and Privacy* 8, 6 (November 2010), 40–47.

Bo Hong and Darrell D. E. Long. 2004. Duplicate data elimination in a SAN file system. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 301–314.

James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. 1998. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology* 7, 2 (April 1998), 192–214.

Keren Jin and Ethan L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the International Systems and Storage Conference (SYSTOR)*. ACM, New York, NY, 7:1–7:12.

Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. 2012. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the International Systems and Storage Conference (SYSTOR)*. ACM, New York, NY, 1–12.

Jürgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. 2012. Design of an exact data deduplication cluster. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–12.

Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. 2012. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–12.

Ricardo Koller and Raju Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 211–224.

Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. 2010. Bimodal content defined chunking for backup streams. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 239–252.

Anthony Liguori and Eric Van Hensbergen. 2008. Experiences with content addressable storage and virtual disks. In *Proceedings of the USENIX Workshop on I/O Virtualization (WIOV)*. USENIX, Berkeley, CA, 1–5.

Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 111–123.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 1–13.

Guanlin Lu, Yu Jin, and David H. C. Du. 2010. Frequency based chunking for data de-duplication. In *Proceedings of the International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, Washington, DC, 287–296.

Guanlin Lu, Youngjin Nam, and David H. C. Du. 2012. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–11.

Udi Manber. 1994. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*. USENIX, Berkeley, CA, 1–10.

Nagapramod Mandagere, Pin Zhou, Mark A. Smith, and Sandeep Uttamchandani. 2008. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. ACM, New York, NY, 12–17.

Dirk Meister and André Brinkmann. 2009. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of the International Systems and Storage Conference (SYSTOR)*. ACM, New York, NY, 1–12.

Dirk Meister and André Brinkmann. 2010. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–6.

Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Parallax: Virtual disks for virtual machines. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, 41–54.

Dutch T. Meyer and William J. Bolosky. 2011. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 1–13.

Grzegorz Milos, Derek G. Murray, Steven Hand, and Michael A. Fetterman. 2009. Satori: Enlightened page sharing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–14.

Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 174–187.

Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. 2006. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 71–84.

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. 2011. Live deduplication storage of virtual machine images in an open-source cloud. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. ACM, New York, 1–20.

Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. 2002. Cluster-based delta compression of a collection of files. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*. IEEE Computer Society, Washington, DC, 257–268.

Ying-Shiuan Pan, Jui-Hao Chiang, Han-Lin Li, Po-Jui Tsao, Ming-Fen Lin, and Tzi-cker Chiueh. 2011. Hypervisor support for efficient memory de-duplication. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, Washington, DC, 33–39.

João Paulo, Pedro Reis, Jose Pereira, and Antonio Sousa. 2012. DEDISbench: A benchmark for deduplicated storage systems. In *Proceedings of the International Symposium on Secure Virtual Infrastructures (DOA-SVI)*. 1–18.

Calicrates Policroniades and Ian Pratt. 2004. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 73–86.

Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 1–13.

Michael O. Rabin. 1981. *Fingerprinting by Random Polynomials*. Technical Report TR-15-81. Harvard Aiken Computation Laboratory. 1–12 pages.

Fatema Rashid, Ali Miri, and Isaac Woungang. 2012. A secure data deduplication framework for cloud environments. In *Proceedings of the International Conference on Privacy, Security and Trust (PST)*. IEEE Computer Society, Washington, DC, 81–87.

Sean Rhea, Russ Cox, and Alex Pesterev. 2008. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 143–156.

Eric W. D. Rozier, William H. Sanders, Pin Zhou, Nagapramod Mandagere, Sandeep M. Uttamchandani, and Mark L. Yakushev. 2011. Modeling the fault tolerance consequences of deduplication. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, Washington, DC, 75–84.

Leonard D. Shapiro. 1986. Join processing in database systems with large main memories. *ACM Transactions on Database Systems* 11, 3 (September 1986), 239–264.

Prateek Sharma and Purushottam Kulkarni. 2012. Singleton: System-wide page deduplication in virtual environments. In *Proceedings of the Symposium on High Performance Distributed Computing (HPDC)*. ACM, New York, NY, 15–26.

Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, Berkeley, CA, 1–10.

Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 1–14.

Mark W. Storer, Kevin Greenan, Darrell D. E. Long, and Ethan L. Miller. 2008. Secure data deduplication. In *Proceedings of the Workshop on Storage Security and Survivability (StorageSS)*. ACM, New York, NY, 1–10.

Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory deduplication as a threat to the guest OS. In *Proceedings of the European Workshop on Systems Security (EuroSec)*. ACM, New York, NY, 1–6.

Kuniyasu Suzaki, Toshiki Yagi, Kengo Iijima, Nguyen Anh Quynh, Cyrille Artho, and Yoshihito Watanebe. 2010. Moving from logical sharing of guest OS to physical sharing of deduplication on virtual machine. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*. USENIX, Berkeley, CA, 1–7.

Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating realistic datasets for deduplication analysis. In *Poster Session of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 1–2.

Yoshihiro Tsuchiya and Takashi Watanabe. 2011. DBLK: Deduplication for primary block storage. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–5.

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. 2010. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 225–238.

Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. *SIGOPS Operating Systems Review* 36, SI (December 2002), 181–194.

Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. 2010. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 1–14.

Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. 2009. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the Conference on Virtual Execution Environments (VEE)*. ACM, New York, NY, 31–40.

Jeff Wright. 2011. Sun ZFS Storage Appliance Deduplication Design and Implementation Guidelines. Retrieved September 12, 2013, from http://www.oracle.com/technetwork/articles/servers-storage-admin/zfs-storage -deduplication-335298.html.

Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, 26–30.

Tianming Yang, Dan Feng, Zhongying Niu, and Ya ping Wan. 2010a. Scalable high performance deduplication backup via hash join. *Journal of Zhejiang University—Science C* 11, 5 (November 2010), 1–13.

Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. 2010b. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Washington, DC, 1–12.

Lawrence You and Christos Karamanolis. 2004. Evaluation of efficient archival storage techniques. In *Proceedings of the Conference on Mass Storage Systems (MSST)*. IEEE Computer Society, Washington, DC, 227–232.

Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. 2005. Deep Store: An archival storage system architecture. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Washington, DC, 1–11.

Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 1–14.