1-5-2009

DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving

Tianming Yang
Wuhan National Laboratory for Optoelectronics, China

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Dan Feng
Wuhan National Laboratory for Optoelectronics, China

Zhongying Niu
Wuhan National Laboratory for Optoelectronics, Wuhan, 430074, China

Yang, Tianming; Jiang, Hong; Feng, Dan; and Niu, Zhongying, "DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving" (2009). *CSE Technical reports*. 58.
https://digitalcommons.unl.edu/csetechreports/58

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2009-0004
Issued Jan. 5, 2009

DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving

Tianming Yang[†], Hong Jiang[‡], Dan Feng[†], Zhongying Niu[†]
[†]*College of Computer Science and Technology, HUST*
Wuhan National Laboratory for Optoelectronics, Wuhan, 430074, China
[‡]*Department of Computer Science and Engineering*
University of Nebraska-Lincoln Lincoln, NE 68588, USA
Email: ytmzqyy@yahoo.cn, dfeng@hust.edu.cn, jiang@cse.unl.edu, niuzhy@163.com

Abstract

We present DEBAR, a scalable and high-performance de-duplication storage system for backup and archiving, to overcome the throughput and scalability limitations of the state-of-the-art data de-duplication schemes, including the Data Domain De-duplication File System (DDFS). DEBAR uses a two-phase de-duplication scheme (TPDS) that exploits memory cache and disk index properties to judiciously turn the notoriously random and small disk I/Os of fingerprint lookups and updates into large sequential disk I/Os, hence achieving a very high de-duplication throughput. The salient feature of this approach is that both the system backup and archiving capacity and the de-duplication performance can be dynamically and cost-effectively scaled up on demand; it hence not only significantly improves the throughput of a single de-duplication server but also is conducive to distributed implementation and thus applicable to large-scale and distributed storage systems.

1 Introduction

Today, the ever-growing volume and value of digital information have raised a critical and mounting demand for long-term data protection through large-scale and high-performance backup and archiving systems. According to ESG (Enterprise Strategy Group), the amount of data requiring protection continues to grow at approximately 60% per year. The massive data needing backup and archiving has amounted to several perabytes [9, 28] and may soon reach tens, or even hundreds of perabytes. For example, NARA (National Archives and Records Administration) plans to make 36 perabytes of archival data accessible on-line by the year 2010 . Large-scale distributed storage systems with tens or hundreds of storage nodes may require a backup system capable of backing up perabytes of data at an aggregate bandwidth of gigabytes per second . Backup and archiving systems thus call for effective solutions to boost both storage efficiency and system scalability to meet the accelerating demand on backup capacity and performance.

In recent years, disk-based de-duplication storage has emerged as a key solution to the storage and bandwidth efficiency problems facing backup and archiving systems [13, 15, 20, 21, 30, 11, 14, 28]. By eliminating duplicate data across the system, a disk-based de-duplication storage system can achieve far more efficient data compression than tapes. DDFS [30], for example, reported a 38.54:1 cumulative compression rate when backing up a real world data center over a time span of one month. Such a high compression rate dramatically reduces the storage and bandwidth requirements for data protection, making it more cost-effective and practical to build a massive disk-based storage system for backup and archiving.

The most common de-duplication method has been to divide a file or stream into chunks and eliminate the duplicate copies of chunks. Duplicate chunks are identified by comparing the chunk fingerprints represented by the hash values of chunk contents. A disk index is used to establish a mapping between the fingerprints and the locations of their corresponding chunks on disks, which makes accessing the index a high frequent event for data de-duplication. Considering the fact that the index locations of the fingerprints to be compared are random in nature and the entire index is usually too large to fit in a server's main memory, the throughput of de-duplication will be limited by the random I/O throughput of the index disk, which for the current technology typically amounts to a few hundred fingerprints per second. The Venti system [21], for example, reported a throughput of less than 6.5MB/s , or 832 fingerprints/s given the typical chunk size of 8KB, even with a RAID (Redundant Array of Inexpensive Disks) of 8 disks for index lookups in parallel.

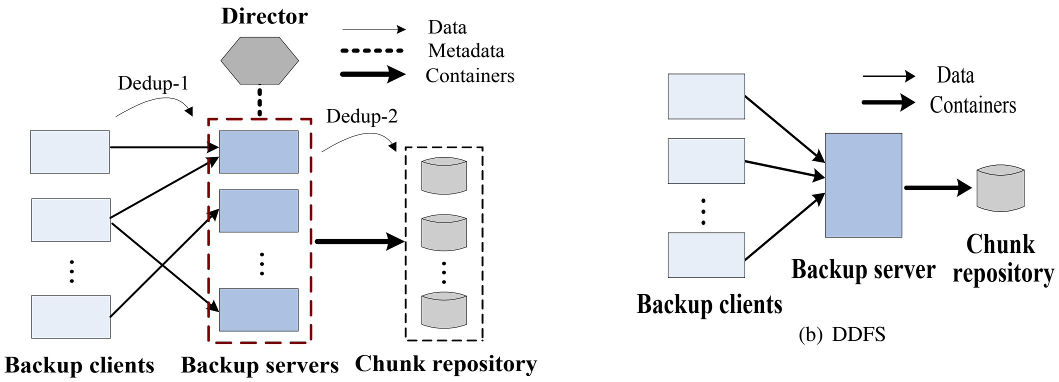
However, most of the existing data de-duplication solutions, with the exception of DDFS [30] published in 2008, have mainly focused on techniques for improving data compression rate rather than methods for improving the throughput of de-duplication. DDFS uses a combination of *summary vector* and cache techniques to speed up the data de-duplication throughput and thus achieves a throughput of over 100MB/s. It exploits an in-memory Bloom filter [3], which compactly represents the fingerprint set of the entire system, to implement the summary vector. By testing whether a data chunk is new to the system, the summary vector can avoid unnecessary lookups for chunks that do not exist in the index. In addition, DDFS employs a novel cache technique called *locality-preserved caching (LPC)* [5] to quickly find chunks that already exist in the system. In order to achieve high cache hit ratios, a *stream-informed segment layout (SISL)* is used to store groups of new data chunks (segments) and their fingerprints in the same order that they occur in a data file or stream to a *container*, i.e., the unit of storage in the system. SISL creates so much spatial locality for chunk and fingerprint accesses that disk index is only accessed for those fingerprints that can not be resolved by the summary vector and miss in the cache. If a fingerprint is found by disk index lookup, all the fingerprints in the container that stores the fingerprint are prefetched to the cache since these fingerprints are more likely to be accessed in the near future. As a result, a single disk access can result in many subsequent cache hits and thus avoid many on-disk index lookups. The DDFS scheme has been proven very efficient and effective, as evidenced by their experimental results on real world workloads , namely, more than 99% on-disk index lookups for de-duplication can be avoided. Nevertheless, the state-of-the-art DDFS scheme suffers from the following limitations in building a large-scale and distributed backup and archiving system.

First, in order to achieve a reasonably small false positive rate, the Bloom filter must be sufficiently large in size, which inevitably limits the system capacity. For an expected chunk size of 8KB, it needs 1GB in-memory Bloom filter to store 2^{30} fingerprints of about 8TB physical storage, which results in a reasonably low false positive rate of 2% [7, 30]. In order to keep the same 2% false positive rate, the size of the in-memory Bloom filter must linearly increase with the system capacity. For example, a 1-perabyte physical capacity will need at least 120GB in-memory Bloom filter. Such a memory cost is prohibitively high for most of the current systems and thus prevents DDFS from being applicable to large-scale and demanding environments in which perabytes of physical capacity is a basic requirement for backups.

Second, the summary vector is intrinsically single-server-oriented and, to the best of our knowledge, there are so far no effective and scalable solutions to ensuring data consistence among the summary vectors in a multi-sever environment. This thus prevents DDFS from being scaled up in terms of performance and capacity in a distributed backup and archiving system that may perform tens of backup jobs simultaneously and require gigabytes-per-second of aggregate bandwidth. Moreover, the summary vector cannot be dynamically and adaptively enlarged in size to accommodate possible increase in system capacity. To increase the system capacity, the summary vector has to be reconstructed by scanning the whole storage to extract all fingerprints, which can impose a very heavy overhead for the backup server in a large-scale storage system.

In summary, DDFS has addressed the important issue of improving data de-duplication throughput that has been largely ignored by most of existing data de-duplication schemes. While DDFS yields a high backup throughput in a single-server de-duplication system, it suffers from poor scalability for large-scale and distributed backup and archiving systems because of its aforementioned inherent limitations. In this paper, we present DEBAR, a scalable and high-performance *DE*-duplication storage architecture for *B*ackup and *AR*chiving, designed to improve capacity, throughput, scalability for data de-duplication. DEBAR uses a two-phase de-duplication scheme (TPDS) that exploits memory cache and disk index properties to judiciously turn the notoriously random and small disk I/Os of fingerprint lookups and updates into large sequential disk I/Os, hence achieving very high de-duplication throughput. The main contributions of this paper include:

- It proposes a novel DEBAR architecture (Section 2) and presents a detailed design of the system (Section 3). DEBAR uses a cluster of storage nodes to construct a *chunk repository*, which provides a large-scale, global de-duplication storage pool for data chunks, and a director, which handles job scheduling, metadata management and load balancing to improve the system's scalability. DEBAR can run multiple backup servers concurrently to support high backup throughput, thus readily deployable to large-scale and distributed storage systems.
- It proposes a simple but effective disk index structure for fingerprint mapping (Section 4), which offers several very useful properties such as uniform and number-ordered fingerprint distribution and capacity and performance scaling. These properties render DEBAR highly efficient, adaptive and scalable.
- It presents a novel *two-phase de-duplication scheme (TPDS)* (Section 5), which performs *sequential index lookup (SIL)* and *sequential index update (SIU)* to judiciously transform the large number of small random disk I/Os into a small number of large sequential disk I/Os for fingerprint lookup and disk index update. Hence, TPDS significantly improves the de-duplication backup performance. TPDS also uses a simple semantic-aware method in its de-duplication Phase I to perform preliminary de-duplication to reduce bandwidth requirements



(a) DEBAR

(b) DDFS

Figure 1: DEBAR architecture VS. DDFS architecture.

for backups. Consequentially, TPDS fully and effectively addresses the issues of fingerprint lookup and disk index update and thus achieves high system scalability.

- It implements a DEBAR prototype on a Linux platform. Experiments with both DEBAR and DDFS show that DEBAR significantly outperforms DDFS in terms of storage efficiency, throughput and system scalability (Section 6). In contrast to DDFS, which features a single-server system and supports a limited system capacity, DEBAR outperforms DDFS in single-server backup capacity by a factor of 8. By simply adding backup servers to the system, DEBAR can be easily deployed to support perabytes of capacity and gigabytes of aggregate throughput in a large-scale and distributed storage system.

2 DEBAR Architecture

The DEBAR architecture, shown in Figure 1(a), uses a cluster of backup servers to provide large-scale and high-performance data backups. A director is designed to provide global management, such as job scheduling, load balancing and metadata management, for the whole system. DEBAR employs a two-phase de-duplication scheme (TPDS). In the de-duplication Phase I (dedup-1), files are transmitted from the backup clients to the backup servers, where the latter build the file metadata and indices for each file and temporarily stores the file data chunks to their local disks. A file index, which facilitates retrieving files from the system, is a sequence of fingerprints that reference to the file chunks. The deduplication Phase II (dedup-2) is exclusively performed by the backup servers. In dedup-2, the backup servers first perform sequential index lookup (SIL) to identify new chunks needing backup. And then the new chunks are written to fixed-sized containers, which are in turn stored to a chunk repository. Finally, the backup servers execute sequential index update (SIU) to write new fingerprints to the disk index.

The main idea behind TPDS is to collect sufficient number of data chunks and fingerprints in dedup-1 so that the system can perform SIL and SIU in dedup-2 to identify new data chunks and write new fingerprints to the disk index more efficiently. Using SIL and SIU, a single disk I/O can handle a large number of fingerprints, thus significantly outperforming the random lookup and update methods in traditional de-duplication systems, such as Venti [21], in which one disk I/O can handle only one fingerprint. In addition, SIL and SIU support a cluster of backup servers to perform parallel sequential index lookups (PSIL) and parallel sequential index updates (PSIU) for fingerprint lookups and disk index updates, and thus outperforming DDFS, shown in Figure 1(b), which supports only one backup server due to its inherent limitations (see Section 1). With the support of more backup servers running in parallel, DEBAR can achieve high aggregate throughput in dedup-2. More importantly, with a simple but effective disk index structure, which provides fundamental supports to the DEBAR architecture with a number of useful properties, and the SIL and SIU techniques, DEBAR can be dynamically and cost-effectively scaled up on demand in terms of the system capacity and the de-duplication performance.

A chunk repository provides a global de-duplication storage pool for data chunks. In a DEBAR system with a single de-duplication server, the chunk repository can be built on the block storage devices of backup servers. As more backup servers are added to the DEBAR system, the chunk repository can be enlarged to support a cluster of storage nodes with potentially perabytes of capacity and gigabytes-per-second of aggregate throughput. In contrast to DEBAR, DDFS supports a very small capability, usually several terabytes to several tens of terabytes, which has upper bounded by the size of the summary vector and cannot be dynamically scaled up.

TPDS postpones the de-duplication data storage to dedup-2, and thus releases dedup-1 from the timeconsuming task of disk index lookups and updates. Moreover, TPDS exploits an in-memory preliminary filter (see Section 5.1 for details) in dedup-1 to improve bandwidth efficiency by eliminating duplicate chunks as much as possible. Since multiple backup servers can run in parallel in dedup-1 to receive data, dedup-1 can effectively avoid both the disk and bandwidth bottlenecks for backups and thus DEBAR can support more backup clients than DDFS, as the latter can support only a limited number of backup clients due to its single-backup-server bottleneck. By eliminating a significant number of duplicate chunks in dedup-1, the number of data chunks that need to be further processed in dedup-2 is substantially reduced, hence further improving the system performance of DEBAR.

It must be noted that DEBAR, in addition to drawing inspirations from DDFS, has adopted a major optimization technique from DDFS. More specifically, the SISL technique proposed in DDFS has been employed in DEBAR to fill containers with backup data for creating chunk locality to improve the LPC [5] hit rate, which enables a high read throughput for data restoration.

3 Debar Design

In this section we describe the design of the main functional components of DEBAR, *director*, *backup clients*, *backup servers* and *chunk repository*, as shown in Figure 2.

3.1 Director

The director is a dedicated control center of the whole system. It supervises the entire backup, restore, verify and resource management operations by means of job objects that consist of attributes and methods to specify *what*, *where*, *how* and *when* for tasks to do. The backup clients and backup servers provide *Job Interfaces* to establish the communication mechanism between jobs. A user can define job objects through the User Interface to backup their data to the system automatically. A backup job object includes at least three attributes, namely, a *client* attribute that specifies a backup client for the job, a *dataset* attribute that specifies the list of files and directories needing backup on the backup client host, and a *schedule* attribute that specifies when the backup job should be scheduled to run. For example, a schedule of ”daily at 1.05am” specifies that the backup job should be scheduled to run at 1.05am each day. The director uses a *Job Scheduler* module to schedule job objects and to assign backup jobs to backup servers to maintain load balancing, and a *Metadata Manager* module to manage job metadata, such as job ID, job size, and file metadata and indices, that provides useful information to the backup and restore processes. An important function of the director is to monitor the states of the backup servers, when necessary; the director initiates a dedup-2 job in which all the backup servers cooperate to store new chunks to the chunk repository in parallel.

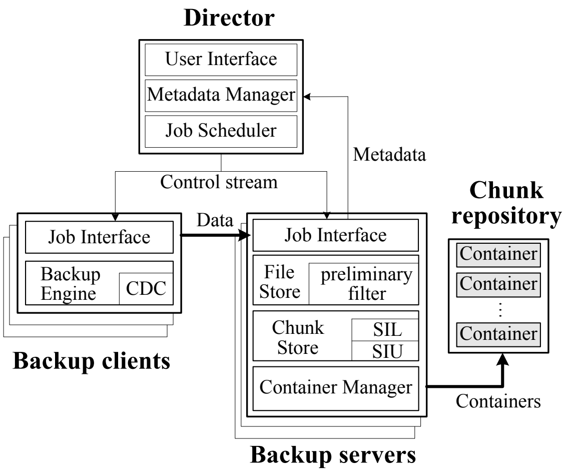


Figure 2: Block diagram of DEBAR.

3.2 Backup Client

Backup clients run on machines that have data to be backed up. The list of files and directories needing backup is specified by the dataset attribute of the job object. The *Backup Engine* module is responsible for reading files from the job dataset and then transmitting them to the corresponding backup server. To backup a file, Backup Engine performs the following operations:

To backup a file, Backup Engine performs the following operations:

- *Metadata backup* sends the file metadata to the Backup Server.
- *Anchoring* divides the file contents into variable-sized chunks using the *content-defined chunking algorithm* (CDC) [20].
- *Chunk fingerprinting* computes the SHA-1 [27] hash (160bits) of each chunk as its fingerprint.
- *Content backup* interacts with the Backup Server to backup the file chunks. It sends chunk fingerprints to the Backup Server to be checked using a preliminary filter to determine whether the corresponding chunks need to be backed up. Then the Backup Engine transfers the chunks needing backup and discards the duplicate chunks.

To restore a file, Backup Engine retrives the file metadata and contents from the Backup Server and then restores the file to a designated directory.

We choose the CDC chunking scheme because it can eliminate the inefficiency of the fixed-sized blocking method, which limits the number of potential duplicates that can be detected. With the fixed-sized blocking method, even a small change to a file, such as inserting data into the beginning of the file, will result in a change to all fixed-sized blocks in the file.

CDC computes the Rabin fingerprintsRABIN81:6, RABIN93:8 of all overlapping fixed-sized (usually 48 bytes) substrings of a file. When the low-order k bits of a substring's Rabin fingerprint is equal to a predetermined constant, the substring constitutes an anchor [17]. A file may contain many non-overlapping anchors. These anchors are used as chunk boundaries to divide a file into variable-sized chunks. The expected chunk size is determined by the parameter k and is 2^k bytes. In DEBAR, we choose an expected chunk size of 8 KB, the same chunk size chosen in DDFS and most of the other existing de-duplication systems. In order to eliminate the possibility of pathological cases described in LBFS [20], DEBAR also imposes a lower bound of 2KB and an upper bound of 64KB on the chunk size in the chunking process.

We use the SHA-1 algorithm to calculate chunk fingerprints since it is not only a collision-resistant hash function, which practically eliminates the probability of two different inputs producing the same output, but also a cryptographic hash function, which makes it computationally infeasible to intentionally create two distinct chunks that hash to the same value [18].

3.3 Backup Server

The backup servers perform data de-duplication using the two-phase de-duplication scheme (TPDS). Backup jobs are performed and finished in de-duplication Phase I (dedup-1), while all the backup servers cooperate to store de-duplicated chunks to the chunk repository in de-duplication Phase II (dedup-2).

Dedup-1 is performed by the File Store module, which receives data stream from the backup clients. To backup a file, File Store performs the following operations:

- *File indexing* manages the file metadata, builds the file index, which is a sequence of fingerprints that reference to the file chunks, and sends the file index and metadata to the director.
- *Preliminary filtering* performs preliminary de-duplication to the fingerprint stream to determine which chunks need to be backed up and thus transferred form the backup client. The received chunks, which need to be further de-duplicated through disk index lookups in dedup-2, are then temporarily appended to a local *on-disk chunk log*.

To restore a file from the system, File Store retrieves the file index from the director, then the file chunks from the Chunk Store module using the fingerprints contained in the file index, and finally sends the restored file to the corresponding backup client.

Dedup-2 is initiated by the director, and executed by the Chunk Store modules of all the backup servers concurrently. Each Chunk Store module performs the following operations in dedup-2 to store 'unique' chunks:

- *Sequential index lookup (SIL)* looks up the disk index to identify new chunks. SIL avoids the small random disk I/Os for fingerprint lookups and thus significantly improves the disk index lookup efficiency.
- *Chunk storing* reads chunks from the local on-disk chunk log and refers to the SIL results to write new chunks to *containers* supported by the Container Manager module.
- *Sequential index update (SIU)* updates the disk index that maps chunk fingerprints to the container holding the chunks after all the identified new chunks have been stored to the Container Manager.

To retrieve data chunks from the system, Chunk Store adopts the *locality preserved caching (LPC)* [30] technique proposed in DDFS. It first looks up the chunk in an in-memory cache. The desired chunk is directly read from the cache if found there. Otherwise, it looks up the disk index to find the container that stores the requested chunk, reads the container to the cache using the Container Manager, and retrieves the desired chunk from the container.

Container Manager is in charge of writing/reading containers to/from the Chunk Repository. When Container Manager stores a container, a unique container ID is generated to identify the container in the Chunk Repository.

The Container Manager module is responsible for writing or reading containers to or from the chunk repository. When a container is written into the chunk repository, a container ID will be generated to uniquely identify the container.

3.4 Chunk Repository

The chunk repository provides a uniform container log storage to the backup servers. A container, i.e., the storage unit of chunk repository, is fixed-sized and self-described in that a metadata section located before the data section of the container stores metadata describing the chunks stored in the data section. The chunk metadata, which locates a particular chunk in its container, includes the fingerprint, chunk size and storage offset of this chunk.

Containers are appropriately sized. A container size of 8MB is chosen for DEBAR, implying that, for an expected chunk size of 8KB, there are about 1024 chunks in a container. In addition, a container ID of 40 bits is used for DEBAR. For an 8MB container, a 40-bit container ID can represent a maximum physical backup capacity of 8 exabytes, thus is sufficient for a PB-scale storage system.

The backup servers fill containers using a *stream-informed segment layout (SISL) algorithm* [30] described in DDFS. SISL writes new chunks to the containers in the logical order that they appear in the backup stream.

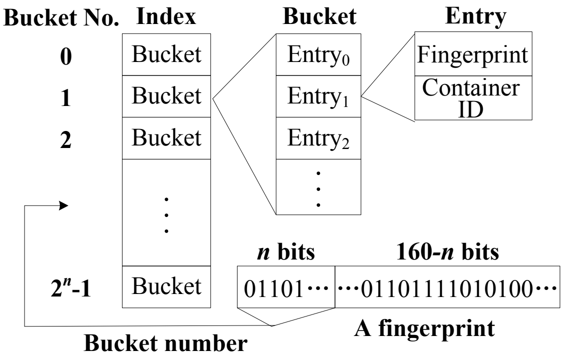


Figure 3: Structure of the DEBAR disk index. The index contains a total of 2^n buckets. We use the first n bits of a fingerprint as the bucket number to map the fingerprint to the corresponding bucket.

It hence creates a spatial locality for the chunk access to achieve a high chunk read throughput due to the increased LPC hit rate.

4 DEBAR Disk Index

In this section, we first describe the structure and unique but useful properties of the DEBAR disk index that efficiently identifies chunks within the chunk repository. We then study an important design problem of the DEBAR disk index, that is, how many fingerprints can an index of a given capacity accommodate before it begins to overflow and thus needs to be enlarged in capacity? Improving the disk index utilization can effectively reduce the index storage overhead, which is particularly important for a PB-scale de-duplication storage system that may require a disk index as big as tens of terabytes in size.

4.1 Structure and properties

The DEBAR disk index, as illustrated in Figure 3, is implemented as a hash table that contains a list of fixed-sized buckets with each bucket containing entries for fingerprints mapped to the bucket. Each entry stores a mapping between a fingerprint and its container ID. Since a fingerprint itself is numerically random in nature, we simply take the first n bits of a fingerprint as the bucket number to map this fingerprint to its corresponding bucket. Such a simple hashing method renders the DEBAR disk index a number of unique but useful properties, as described below.

Uniform fingerprint distribution: Thanks to the good randomness resulting from the SHA-1 algorithm, the fingerprints will be distributed to the index buckets in a sufficiently uniform manner. And given a sufficiently large number of appropriately sized buckets, the index can achieve a relatively high utilization before it begins to overflow. To further improve the index utilization, if a bucket does overflow, we can randomly select an adjacent bucket to place the extra entry. If a bucket's two adjacent buckets are both found to be full, it indicates that the disk index is nearly full with a high probability (see Section 4.2 for details) and needs to be enlarged.

Number-ordered fingerprint distribution: Fingerprints are automatically sorted into different index buckets in the order of their corresponding bucket numbers (i.e., their first n bits). In other words, the fingerprint distribution of the index is number-ordered. Such a number-ordered fingerprint distribution makes it possible for TPDS to perform sequential index lookups and updates, a hallmark of the DEBAR disk index scheme.

Simple capacity scaling: The index capacity can be easily enlarged by simple buckets copying operations. Specifically, constructing a new index with 2^{n+1} buckets from the old index with 2^n buckets can be done as follows: the entries in bucket $k(k = 0, 1, \dots, 2^n - 1)$ of the old index are copied to buckets $2k$ and $2k + 1$ of the new index to ensure that buckets $2k$ and $2k + 1$ store the entries whose fingerprints' first $n + 1$ bits constitute the binary numbers $2k$ and $2k + 1$ respectively. Bucket $k(k = 0, 1, \dots, 2^n - 1)$ of the old index may also contain, with a low probability, a few extra entries overflowed from its adjacent buckets. In this case, we use the first n+1 bits of the fingerprint of an overflowed entry as the bucket number to copy this entry to the corresponding new buckets of the new index. Such a simple scaling procedure enables the system to easily accommodate and adapt to larger application and capacity requirements. Another method to reconstruct the index is to scan the chunk repository to extract necessary information from the containers to the reconstructed bucket entries. But such a high-cost reconstruction method is not suitable for index scaling but only used to recover a corrupted index.

Simple performance scaling: The DEBAR disk index can also be scaled by dividing it into equal-sized parts with each part being located in a different backup server to provide parallel and distributed fingerprint index service. Supposing that the index is divided into 2^w parts, the first w bits of the fingerprints, where $w < n$, will be used as the backup server number and then the remaining $n - w$ bits will be used as the bucket number for the fingerprint mapping. Since these 2^w backup servers can perform fingerprint lookups and updates in parallel, the DEBAR system performance can be easily scaled up by simply adding the number of backup servers.

The DEBAR disk index can be initially deployed small in size, for example, 16GB or 32GB. With the system's growth, the index can become full, that is, three adjacent buckets can be found to be all full, in which case DEBAR automatically scales up the index using the property of capacity scaling. In addition, if the index also becomes so large in size that it becomes a performance bottleneck, DEBAR can further divide the index into multiple parts to be distributed among more backup servers using the performance scaling property. Since such simple scaling schemes do not need to change and scan the chunk repository, the capability and performance of the DEBAR system can be dynamically and cost-effectively scaled up.

4.2 Overflow probability

Improving the disk index utilization can not only reduce the metadata storage overhead but also improve the DEBAR backup performance, since a small disk index can effectively reduce the time overhead of the sequential index lookup and update (see Section 5 for details). So, the disk index should be sufficiently well occupied (i.e., highly utilized) by inserted fingerprints before it needs to be enlarged using the property of capacity scaling. The key to designing such a disk index is to select an appropriate size for the disk index bucket that can meet the demand of the expected disk index utilization while introducing little additional overhead. To provide a useful guide for the bucket size selection, we first estimate the upper bounds for the index overflow probability under different bucket sizes and different levels of index utilization. We then validate the theoretical estimations via experimental measurements, which allows us to finally select an appropriate bucket size for the DEBAR system.

Supposing a disk index with a total of $2^n (n > 20)$ buckets and that fingerprints are inserted into the disk index in a uniform manner, that is, the probability of a fingerprint being inserted into any given bucket is $1/2^n$. We examine two different methods with which the disk index processes the inserted fingerprints. Method A assumes that each disk index bucket has an infinite capacity and thus fingerprints are never overflowed to adjacent buckets. Method B assumes that each disk index bucket has a limited capacity of $b (b > 1)$ fingerprints. When a bucket overflows due to an incoming fingerprint, the system randomly selects one of its two adjacent buckets to place the new fingerprint. And if both of its two adjacent buckets are full, it initiates a process to enlarge the disk index using the property of capacity scaling (Section 4.1).

For method A, let C be the event that there exist three adjacent buckets that collectively contain a number of fingerprints equal to or larger than $3b$ after a total of $\eta \times b \times 2^n$ $(0 < \eta < 1)$ fingerprints have been inserted into the disk index. Then, the probability that event C happens can be expressed as:

$$Pr(C) < (2^n - 2) \times \left(1 - \sum_{k=0}^{3b-1} \frac{(3\eta b)^k e^{-3\eta b}}{k!}\right) \quad (1)$$

Table 1: Calculated upper bound of $Pr(D)$.

Bucket size (KB)	η	$Pr(D) <$
0.5	35%	1.71%
1	45%	1.02%
2	55%	1.24%
4	70%	1.59%
8	80%	1.91%
16	85%	1.93%
32	90%	2.16%
64	92%	2.08%

For method B, let D be the event that there exists a bucket that initiates a disk index capacity scaling process, as a result of finding itself and both of its two adjacent buckets full, before $\eta \times b \times 2^n$ fingerprints have been inserted into the disk index, with η being the disk index utilization. Since for method B the events that buckets overflow are not independent, we find it extremely difficult to derive an expression for $Pr(D)$, namely, the probability that event D happens. As a result, we will use $Pr(C)$ to approximately estimate $Pr(D)$ by postulating that $Pr(D) < Pr(C)$ based on the following observations. Assume that both methods A and B used to process the same fingerprint arrival process simultaneously and independently and event C has not happened. Then for method B the probability that event D happens, namely the conditional probability $Pr(D/\overline{C})$, will be sufficiently low since no three adjacent buckets in A have had a total of $3b$ fingerprints inserted for a given disk index utilization. On the other hand, if for method A event C has happened, then for method B event D will also very likely happen. Consequently, we use formula (1) to estimate the upper bound of $Pr(D)$ for a 512GB disk index under different bucket size and disk index utilization η, and the results are shown in Table 1. We construct the disk index buckets using disk blocks with each disk block (usually 512 bytes in size) storing up to 20 fingerprint entries (an entry is 25 bytes). Then, for a given bucket size listed in the table, its corresponding parameters b and n can be determined. For example, an 8KB bucket contains 16 disk blocks and thus can store up to 320 fingerprint entries, giving rise to $b = 320$ and $n = \log_2(512GB/8KB) = 26$. The result obtained from formula (1) based on these parameters implies that, for a 512GB disk index with 8KB-sized buckets, the probability that there exists a bucket that initiates a disk index capacity scaling process as a result of finding itself and its two adjacent buckets full before the disk index utilization reaches 80% is less than 1.91%. In other words, if a bucket overflows, with both its two adjacent buckets also full, and initiates a disk index capacity scaling process, then with a very high probability (over 98.09%) the disk index utilization is at or over 80%.

In order to validate the above theoretical estimations, we carry out experiments to measure the actual utilizations of the disk index, and to provide practical evidence to verify our postulation that $Pr(D/\overline{C})$ is sufficiently low. If given that method A is used and event C has not happened, then for method B a prerequisite for the occurrence of event D is the occurrence of event Q, in which there exists an incoming fingerprint whose insertion into the disk index gives rise to four or more adjacent full buckets without the prior existence of exactly three adjacent full buckets, before $\eta \times b \times 2^n$ fingerprints have been inserted into the disk index. In other words, event Q is conditioned on the existence of two two-adjacent full buckets or one two-adjacent full buckets and one single full bucket that are separated by a bucket with exactly one fingerprint short of being full. So, we have $Pr(D/\overline{C}) < Pr(Q)$, and just need to estimate $Pr(Q)$ in the experiment.

We use an in-memory counter array with each counter representing a disk index bucket, which is initially set to zero, to simulate the 512GB disk index with 2^n (n ranges from 30 to 23, representing the range of bucket size from 0.5KB to 64KB) buckets, and a 64-bit variable, which is incremented by 1 every time, as input to the SHA-1 algorithm [27] to generate a sufficiently large number of different random fingerprints. It's feasible to simulate a real world fingerprint sequence using a variable as input to the SHA-1 algorithm since the SHA-1 fingerprints are essentially random and independent of each other no matter how similar the inputs are. To insert a fingerprint, we use the first n bits of the fingerprint as the counter number to check the corresponding in-memory counter to see whether it has reached the bucket capacity b (b assumes the values of 20, 40, to 2560 respectively). If not, the counter is incremented by 1, otherwise, we randomly select an adjacent counter to increment by 1. If it's two adjacent counters have both reached b, the test exits and the disk index utilization is calculated by dividing the number of inserted fingerprints by the number $b \times 2^n$. For each bucket size, we have run the experiment 50 times, for a total of 400 times of the experiment, and the results are shown in Table 2 in which η represents the measured disk index utilization, ρ represents the average percentage of full buckets in the experiment, n_3 and n_4 record the total numbers of exactly-three-adjacent full buckets and four-or-more-adjacent full buckets in the tests respectively.

The experimental results reveal that n_3 is relatively very small and n_4 is practically zero. In all the 400 tests we found a total of 617 exactly-three-adjacent full buckets and did not find any four-or-more-adjacent full bucket. This indicates that the probability of forming four or more adjacent full buckets is already extremely low, let alone forming four or more adjacent full buckets without the prior existence of exactly three adjacent full buckets. So, it is reasonable to argue that $Pr(Q)$ is indeed extremely low, and $Pr(D/\overline{C}) < Pr(Q)$ can be even lower. Hence, if event D does happen in a real world system, then for method A event C must have happened with very high probability. In other words, our postulation that $Pr(D) < Pr(C)$ is practically correct.

Table 2: Statistics of the disk index tests.

Bucket	$\eta(Min)$	$\eta(Max)$	$\eta(Avg)$	ρ	n_3	n_4
0.5KB	38.23%	45.07%	41.45%	0.068%	147	0
1KB	52.67%	59.73%	56.79%	0.075%	124	0
2KB	64.72%	71.16%	68.04%	0.088%	106	0
4KB	75.87%	79.32%	77.58%	0.13%	97	0
8KB	82.36%	86.72%	84.23%	0.15%	83	0
16KB	87.04%	89.66%	88.25%	0.16%	78	0
32KB	91.31%	92.89%	92.14%	0.20%	67	0
64KB	93.98%	94.75%	94.43%	0.21%	62	0

In Table 2, we can also find that all the measured η values are larger than the corresponding η values calculated in Table 1 based on Formula (1), which further verifies the relative accuracy of the theoretical estimations of Table 1. The percentage of full buckets in the disk index as it approaches its maximum utilization is rather small, less than 0.3% in all the 400 tests. This is because the SHA-1 fingerprints are extremely random and thus are distributed to the index buckets in a sufficiently uniform manner.

Based on the results of Table 1 and Table 2, we have selected 8KB as the bucket size for the DEBAR disk index in order to achieve over 80% disk index utilization for a backup server that may in turn support up to 512GB disk index.

The additional computation overhead due to searching a fingerprint in a large 8KB-sized bucket, which can contain up to 320 fingerprints, is negligible for modern processors. We have tested the speed of fingerprint lookup in main memory using an Inter Xeon DP 5365 3.0 GHz CPU running at 90% utilization, and have achieved a speed of 2.749 million fingerprints per second with each fingerprint requiring 320 comparison operations. It is precisely this high speed of in-memory fingerprint lookup that motivates us to use large-sized bucket for disk index to compensate for the relatively low speed of disk I/O for the sequential index lookup and update (see Section 5 for details). The large 8KB bucket has almost no adverse impact on the performance of the random on-disk fingerprint lookup. Since the time overhead of a random small disk I/O stems mainly from the disk seek rather than data transfer, the time overhead of a random 8KB disk I/O is almost the same as that of a random 512-byte disk I/O. A random lookup in an overflowed bucket can require two random disk I/Os if the desired fingerprint is not found in the first lookup. However, the percentage of overflowed buckets, even when the disk index is nearly full, is generally very low (see the ρ values in

Table 2), so the performance impact of overflowed buckets can be negligible. Moreover, since the random disk I/Os for fingerprint lookup in DEBAR only occur during data restoration, and the LPC [30] read cache can be used to eliminate most of these random disk I/Os.

It should be noted that the results shown in Table 1 and Table 2 can also be used to guide the disk index design for other de-duplication systems. Designers can select an appropriate bucket size based on these results and their other special considerations.

5 Two-Phase De-duplication Scheme

We have briefly described TPDS in Section 1 and 2. In this section, we further detail the design and implementation of preliminary filtering, SIL, chunk storing and SIU in TPDS.

5.1 Preliminary filtering

The *preliminary filtering* in TPDS is designed to eliminate the duplicate data in dedup-1 to reduce not only the bandwidth requirement for backups but also the number of chunks needing to be further processed in dedup-2.

Although we can not definitively identify a new chunk in dedup-1 since index lookups are postponed to dedup-2 we can still definitively identify most duplicate data. For example, the internal duplication of a job dataset can be easily identified instead of resorting to the index lookup. What's more, since the director maintains job metadata, including the directory structure and file indices of all backup jobs, we can perform high-level (file-level) coarse-granularity de-duplication to eliminate identical files or directories before performing low-level chunk-level de-duplication. For example, the directory synchronization techniques, such as HHT [14] and HDAG [11], can be employed to support the preliminary filtering provided that the directory structures are encoded as HHTs or HDAGs by the file system. The traditional incremental backup scheme [2] can also be applied to eliminate the files that are not updated since the last backup. Although the preliminary filtering can be implemented using the aforementioned multiple techniques, DEBAR implements it in a simple but effective way by exploiting the job chain semantics, as described below.

Based on the fact that multiple running instances of the same job object Job_x form a chronologically ordered job chain $Job_x(t_0)$, $Job_x(t_1)$, ..., $Job_x(t_n)$ ($t_0 < t_1 < \ldots < t_n$), which contains all historic versions of the dataset of Job, we can observed that two adjacent versions of the job dataset usually share the most number of files or data chunks. For example, the backup job that periodically uploads the snapshots of a file system usually shares a large percentage of data between its two adjacent running instances. Based on the above observation, we use the fingerprints of the dataset of $Job_x(t_{n-1})$ as *filtering fingerprints* to filter duplication in the dataset of $Job_x(t_n)$.

We implement the preliminary filter as an in-memory hash table with a total of 2^m buckets. Each bucket contains a pointer to a linked list with each node storing a fingerprint mapped to this bucket. DEBAR uses the first m bits of a fingerprint as its hash to map this fingerprint to its corresponding bucket.

Before running, the preliminary filter is initialized by inserting into it the filtering fingerprints.

For an incoming fingerprint F,

- the preliminary filter checks whether F is in the filter. If not found there, then it is inserted to the filter and its node is marked as 'new'; meanwhile, its corresponding data chunk $D(F)$ is transferred from the backup client to the on-disk chunk log as group $< F, D(F) >$. Otherwise,
- the preliminary checks whether F is new. If it is not new, its node is marked as 'new'.

When the data transmission is finished, all the new fingerprints in the preliminary filter are collected to a file called *undetermined fingerprint file*, which represents the fingerprints needing to be further identified through sequential index lookups.

The preliminary filter can be set as large as possible in size (e.g. 1GB). For small jobs, the filtering fingerprints can be completely inserted into the filter in advance to ensure that the filter will not overflow during the execution and fingerprint replacement will not happen. For large jobs, the filtering fingerprints can be divided into multiple parts in their logical order and inserted into the filter group by group. When the filter is full, we use the FIFO (First-In-First-Out) replacement policy, combined with the LRU (Least-Recently-Used) replacement policy, to select victims for replacement from the filter.

5.2 Parallel sequential index lookups

The idea of SIL is based on the following observations.

Given a sufficiently large random fingerprint set, its fingerprints will be mapped to the whole disk index in a roughly uniform manner. Supposing that a fingerprint set with 2^{23} fingerprints is mapped to a disk index with 2^{26} buckets, then roughly every 8 buckets will contain a fingerprint from the fingerprint set. Considering that the fingerprint distribution of a disk index is number-ordered (see Section 3), we can sort these fingerprints in the order of their numbers and sequentially read the index buckets for fingerprint lookups. Note that we can sequentially read thousands of buckets per I/O. The data transfer rate of sequential large disk I/Os is generally over one order of magnitude faster than that of random small disk I/Os.

The index lookup workflow is illustrated in Figure 4. The DEBAR system first reads fingerprints from the undetermined fingerprint files and inserts them to an in-

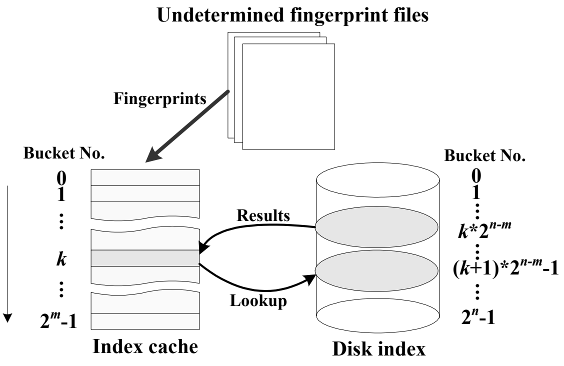


Figure 4: Sequential index lookup.

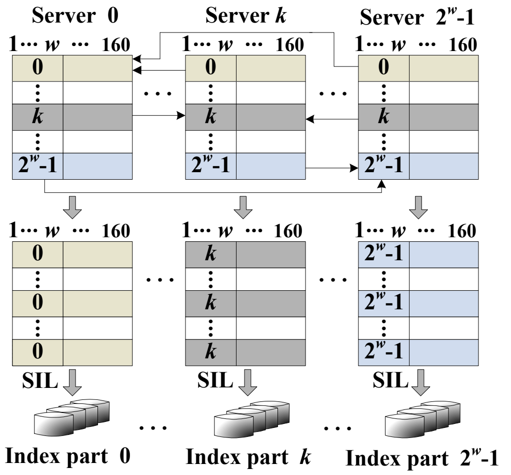


Figure 5: Parallel sequential index lookups.

memory index cache, which is a hash table similar to the preliminary filter described in Section 5.1. When the insertion is finished, all the fingerprints are automatically sorted to the buckets of the index cache in the order of their numbers. And then the fingerprints in bucket $k(k = 0, 1, \ldots, 2^m - 1)$ of the index cache are exactly mapped to 2^{n-m} consecutive buckets, namely bucket $k \times 2^{n-m}$ to bucket $(k+1) \times 2^{n-m} - 1$, of the disk index. So, for the fingerprint lookup, Chunk Store just needs to sequentially read large bulks of consecutive buckets from the disk index to the memory, and then looks up the fingerprints in the corresponding buckets of the index cache. If a fingerprint is found in the index cache, then it is duplicated and its node is deleted from the index cache. Otherwise, its node is retained in the index cache to indicate that it's a new fingerprint to the system. After the completion of the entire lookup, all the new fingerprints are retained in the index cache and used in the chunk storing operation with the chunk log.

Let f denote the number of fingerprints that SIL processes and t denote the time consumed by SIL, then, the efficiency of SIL η can be represented as $\eta = f/t$. Due to the huge performance gap between disk I/O and processor power, the SIL time t is mainly determined by the disk index size s and the sequential disk I/O transfer rate r, but usually independent of the number of fingerprints processed. Thus, we have $\eta = (fr)/s$.

The index size s is determined by the capacity requirement of the backup system. Considering that a 512-byte disk block (i.e., bucket) can contain up to 20 entries (i.e., fingerprints), with each entry occupying 25 bytes, and if we take the first 26 bits of the fingerprint as the bucket number, which can represent a total of 2^{26} buckets, a 32GB index can contain a maximum of $2^{26} \times 20$ fingerprints. For an expected chunk size of 8KB, this means a maximum physical capacity of 10 TB of backup/archival data. For a disk index supporting a 200MB/s sequential disk I/O transfer rate that most RAIDs are capable of today, the index lookup can be finished within 3 minutes. Obviously, the more fingerprints in process, the more efficient the lookup will be. Using the about 1GB memory cache, we can provide lookups for about 44 million fingerprints. For an expected chunk size of 8KB, this means about a lookup efficiency of about 240 thousand fingerprints per second. Note that such a lookup speed is over two orders of magnitude higher than conventional random index lookup approaches, whose speed is usually no more than 1000 fingerprints per second. In order to fully utilize the index cache, DEBAR usually provides synchronous lookups for more than one job.

One of the main advantages of DEBAR over DDFS is that it can support much larger physical backup/archiving capacity. Our experiments indicate that, using about 1GB memory cache, DEBAR can support about 512GB disk index, thus achieving a 128TB physical backup capacity while maintaining the same de-duplication throughput as DDFS. Such a capacity is 16 times larger than that of DDFS, which can only support a maximum physical capacity of about 8TB using 1GB in-memory summary vector. More importantly, DEBAR is highly scalable, as it intrinsically supports parallel sequential index lookups (PSIL), which can be applicable to a cluster of backup servers.

In a large-scale multi-server DEBAR system, the disk index can be divided into multiple equal-sized parts with each being located in a different backup server. Supposing a total of 2^w backup servers, then backup server $k(k = 0, 1, \ldots, 2^w - 1)$ stores index part k, which maps the fingerprints whose first w bits constitute the binary number k. These 2^w backup servers cooperate to perform PSIL, as illustrated in Figure 5.

First, each backup server's undetermined fingerprints are divided into subsets according to the first w bits of the fingerprints. Then, these 2^w backup servers exchange their subsets to ensure that backup server $k(k = 0, 1, \ldots, 2^w - 1)$ processes the fingerprints whose first w bits constitute the binary number k. After the exchange is finished, backup server $k(k = 0, 1, \ldots, 2^w - 1)$ uses its local index part k to perform SIL. Since 2^w SILs are being performed in parallel, the efficiency of PSIL can be significantly higher than that of PIL. After the completion of PSIL, the 2^w backup servers exchange their lookup results to ensure that each backup server gets its own lookup results. Then these 2^w backup servers can perform chunk storing in parallel.

PSIL is an effective technique to improve the system capacity and performance. Our experiment indicates that using a cluster of 16 backup servers, the DEBAR system can support a maximum physical capacity of over 2PB, and an aggregate throughput of over 1.7GB/s.

5.3 Chunk storing

After completing the index lookups, the Chunk Store sequentially reads $< F, D(F) >$ groups (see Section 5.1) from the chunk log and refers to the index cache to write new chunks to the containers.

Specifically, for a $< F, D(F) >$ group read from the chunk log,

- Chunk Store checks whether fingerprint F is in the index cache. If found there, it checks whether its corresponding container ID is null. If container ID is null, it writes $< F, D(F) >$ to the container. Otherwise, it discards $< F, D(F) >$.
- If fingerprint F is not in the index cache, then Chunk Store discards $< F, D(F) >$.

If the container is full, Chunk Store

- creates a new container in the memory for chunk writing;
- submits the full container to the Container Manager, which then appends the received container to the chunk repository and returns the container ID; and
- writes the container ID to those index cache nodes whose fingerprints and chunks have been stored in this container.

The chunk storing process can be very efficient since data is read from the chunk log and written to the chunk repository sequentially. Such a sequential process also preserves chunks locality, which helps improve the chunk read performance.

After the chunk storing process is completed, all the fingerprints and their corresponding container IDs in the index cache are written to a file called *unregistered fingerprint file*, which represents the fingerprints needing to be updated to the disk index.

5.4 Sequential index updating

We use the *sequential index updating (SIU)* technique to effectively solve the disk index update problem. The principle of SIU is similar to that of SIL. It first reads the unregistered fingerprints to the index cache and large bulks of consecutive buckets from the disk index to the memory, and then updates these buckets using the fingerprints and their corresponding container IDs in the corresponding buckets of the index cache. Finally, SIU writes these updated buckets to the disk index. Since, all these read and write operations are large sequential disk I/Os, SIU can be far more efficient than the conventional random index update. Like SIL, SIU naturally supports a cluster of servers to perform parallel sequential index update (PSIU), which is a similar process to PSIL. Since the number of unregistered fingerprints after a PSIL is performed is usually smaller than that of the undetermined fingerprints checked by PSIL, we can perform asynchronous PSIU with one PSIU servicing more than one PSIL using the same amount of main memory.

Unlike DDFS, which can not avoid duplicated storing incurred by asynchronous updates, TPDS can use a simple mechanism to effectively eliminate duplicate chunks due to asynchronous PSIU, as described below.

- Each backup server maintains a *checking fingerprint file*. Whenever a SIL is finished, the lookup result is further de-duplicated to eliminate the fingerprints that are also found in the checking fingerprint file. After the checking is completed, the checking fingerprint file is updated by appending it with the fingerprints in the lookup result.
- Whenever a SIU is finished, the checking fingerprint file is updated by removing those fingerprints that have been written to the disk index in SIU.

For a single-server DEBAR system, the backup server only uses the unregistered fingerprint file to further check the SIL result and does not maintain such a checking fingerprint file.

6 Experimental Evaluation

We have implemented DEBAR in the Linux platform with about 15, 000 lines of C code and a total of three software modules for the director, backup client and backup server respectively. We have also implemented a DDFS prototype according to the description of the original DDFS paper [30] for the purpose of performance comparison. Since the DDFS paper did not describe how the disk index is updated, nor could we obtain the detailed updating method from the authors due to proprietary reasons, we uses a in-memory write buffer to speedup the disk update for DDFS. During backup, new fingerprints are stored in the write buffer. When the buffer fills, the system pauses to flush the buffer to the disk index using the SIU algorithm. The idea of buffering index updates during backup has been proposed in Foundation [24], another de-duplication storage system that uses similar techniques as DDFS to avoid disk index lookup bottleneck.

In this section, we evaluate DEBAR through two main experiments. First, we evaluate the performance of

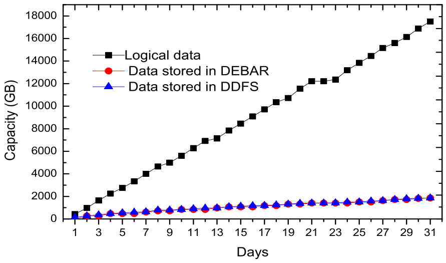


Figure 6: The amount of logical data backed up versus the amount of physical data stored.

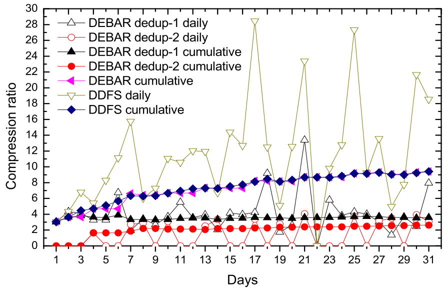


Figure 7: Data compression ratios over time.

a single-server DEBAR and DDFS under a real-world workload in terms of data compression ratio, throughput and the maximum system backup capacity. Second, we measure the aggregate throughput of the multi-server DEBAR deployment to evaluate the system scalability.

In our experiments, the DEBAR director, DEBAR backup servers and DDFS backup server run on a 18-node Linux (RedHat Linux kernel 2.6.8) cluster in which, each node was a computer with an Intel Xeon 3.0 GHz CPU, 4GB RAM, two 1-Gigabit NIC cards and two Highpoint Rocket 2220 RAID controllers with each being attached to 8 SATA disks used for chunk repository (used in the first experiment) and disk index (or chunk log) respectively. The DEBAR backup server uses 1GB memory for the index cache for SIL and SIU, while DDFS uses 1GB memory for the global-based Bloom filter and another 384MB memory for fingerprint cache–with 256 MB for buffering index writes and the remaining 128 MB for the LPC cache. Both DEBAR and DDFS employ the CDC chunking scheme with an expected chunk size of 8KB. In the first experiment, DEBAR and DDFS each uses 32GB disk index.

6.1 Performance Comparison between DEBAR and DDFS

The data center in our test is HUSt [29], a massive storage system that was built at the Wuhan National Laboratory for Optoelectronics in China. HUSt consists of a large-scale object-based storage system with 32 storage nodes and a high-performance cluster with 64 computing nodes. The current system supports several applications such as GISG, a geographic information system grid, and U-store, a WAN (Wide Area Network) resource management system and several internal data sharing platforms for scientific and engineering research and applications. At present, HUSt holds 65TB of data, including structured databases and unstructured files. Most of this data is immutable reference data; the rest are backup versions of the HUSt application servers. Each version includes one week of backup data of an application server, which usually backs up its data with a policy of daily incremental and weekly full backups. We used 8 HUSt storage nodes, each running a DEBAR backup client or a DDFS backup client, to backup the versions in their chronological order on a one-version-per-day basis for a time span of one month. Each day, the 8 nodes first run in parallel to send data to the DEBAR backup server for the backup service, and then run in parallel to send the same data to the DDFS backup servers for their respective backup services.

6.1.1 Data compression ratio

Figure 6 shows the amount of logical data backed up versus the amount of physical data actually stored in the system over time. The average amount of logical data needing backup each day is about 583GB, with certain days being over 800GB and certain other days being less than 150GB. At the end of the 31st day, the total amount of logical data reaches about 17.09TB. The actual physical data stored in DABAR and DDFS is the same, about 1.82TB, achieving a data compression ratio of about 9.39 to 1.

The detailed data compression radios are shown in Figure 7, which shows the effectiveness of DEBAR preliminary filter in terms of eliminating duplication in dedup-1. In the experiment, we use a 1-GB in-memory preliminary filter to store filtering fingerprints for the 8 backup jobs. In the first two days, the preliminary filter eliminated all the duplicate data (data was then directly written to containers during backup and new fingerprints were updated to the disk index using SIU after backup in these two days), thus achieving the same daily compression ratios as DDFS. This is because the DEBAR system had no history data during its initial deployment, the preliminary filter was not full, and cache replacement did not take place in these two days. In the following days, the DEBAR dedup-1 daily compression radio is lower than that of DDFS, because the preliminary filter only eliminates duplication between adjacent backup sessions. Nevertheless, the preliminary filter is still obviously effective in reducing duplication, achieving a stable cumulative compression ratio(DEBAR dedup-1 cumulative)at around 3.6:1. This illustrates that a large percentage of duplicate chunks exist within the data version itself or between two adjacent data versions. Through preliminary filtering, the amount of data needing to be further processed by DEBAR is significantly reduced. As a result, in the experiment, DEBAR's dedup-2 does not run daily, and it just runs on the 4^{th}, 7^{th}, 8^{th}, 10^{th} ,13^{th}, 14^{th}, 17^{th} , 19^{th}, 21^{st}, 25^{th}, 27^{th}, 28^{th}, 30^{th} and 31^{st} day, for a total of 14 times. On these days, DEBAR dedup-2's daily compression ratio (the daily rate of data reduction due to dedup-2 on chunks of the on-disk chunk log) changes roughly increasingly, from 1.65:1 to 4.05:1. DEBAR dedup-2 eliminates the remaining duplicate chunks in entire system, achieving a cumulative compression ratio (the cumulative ratio of data reduction due to dedup-2 on chunks of the disk log) of 2.6:1 at the end of the 31st day. Note that the DEBAR dedup cumulative compression ratio (the cumulative ratio of data reduction due to dedup on chunks of the disk log), the DEBAR cumulative compression ratio, and the DDFS cumulative compression ratios all increase over time. This is because DEBAR dedup-2 and DDFS eliminate duplicate chunks in the global system. The more the data stored in the system, the more duplicates can be eliminated in the incoming backup data stream.

6.1.2 Throughput

Figure 8 shows that DEBAR maintains a high dedup-1 daily throughput, ranging from the lowest of 303MB/s to the highest of nearly 1100MB/s. At the end of the 31^{st} day, DEBAR achieves a cumulative dedup-1 throughput of 641.6MB/s. Such a high performance is mainly attributed to the preliminary filtering, which eliminates most of the duplicated data in the network transfer and hence significantly reduces the bandwidth requirement for backups. The overall DEBAR cumulative throughput is calculated by dividing the amount of logical data by the total amount of time that the dedup-1 and dedup-2 processes consume to backup these logical data. As expected, DEBAR keeps a high cumulative throughput, up to 329.2MB/s at the end of the experiment.

Figure 9 compares the de-duplication throughput of DEBAR dedup-2 and DDFS in the one-month experiment. The DEBAR dedup-2 throughputs are calculated by taking into account the SIL and SIU time that they consume.

In the experiment, the throughput of DEBAR dedup-2 chunk storing is quite stable at about 224MB/s, which is exactly the sustained read throughput of the disk log. The average time spent on SIL and SIU are about 2.53 and 6.16 minutes respectively. In all the 14 dedup-2 processes, SIL runs 14 times and SIU runs only 5 times. Depending on whether it includes SIU, the DEBAR dedup-2 daily throughput fluctuates in a small range between about 170MB/s and 206.8MB/s, giving rise to a relatively stable cumulative throughput of about 197MB/s as shown in Figure 9. In contrast to DEBAR dedup-2, which can perform asynchronous SIU with one SIU servicing the outcomes of multiple SIL, DDFS depends on an in-memory write buffer to store new fingerprints and must occasionally pause to flush the write buffer to the disk index during backup thus degrading the inline de-duplication performance. By using a large 256MB write buffer, the buffer flush operations were limited to two times each day in the experiment, and, as a result, the DDFS throughput degradation is alleviated with a daily throughput over 155MB/s, and a cumulative throughput of about 189MB/s in the end as shown in Figure 9. Without taking into account the buffer flush time, DDFS has shown a stable throughput of about 210MB/s, which is exactly the sustained throughput of the network card in our experiment. It should be noted that, we used a 32-GB disk index for DDFS, corresponding to the 1-GB in-memory Bloom filter used in the experiment that represents a maximum system capacity of about 8TB. In the case of a larger-scale system(e.g., hundreds of terabytes), the size of the disk index will be far larger than 32GB, then the inline index updating by using a write buffer can become a serious performance problem for DDFS. Using a sufficiently large write buffer to postpone the index update to the end of the backup would be an alternative for DDFS in this case.

6.1.3 System backup capacity

SIL and SIU can be performed on a large-sized disk index to support a large system backup capacity while maintaining a high de-duplication throughput.

We have tested the efficiencies of SIL and SIU by varying the size of disk index (32GB, 64GB, 128GB, 256GB, and 512GB) and the size of an in-memory index cache (1GB, 2GB, and 3GB) on the DEBAR backup server. Figures 10 and 11 show the time overheads and the efficiencies of SIL and SIU respectively, which are obtained by averaging the results of 10 runs. As expected, the time overheads of SIL and SIU are only related to the disk index size and the disk transfer rate and are independent of the number of fingerprints processed (i.e., the size of the index cache being used). With the size of the disk index increasing from 32GB to 512GB, the SIL and SIU time increase from 2.53 minutes and 6.16 minutes to 38.98 minutes and 97.07 minutes respectively. The efficiencies of SIL and SIU are defined by the formula $=fr/s$ that we have inferred in Section 5.2, and can be effectively improved with a large index cache that can contain a large number of fingerprints. As Figure 11 shows, with a 32GB disk index and 3GB in-memory index cache (SIL-3GB, SIU-3GB), the measured maximum speeds of SIL and SIU are about 917

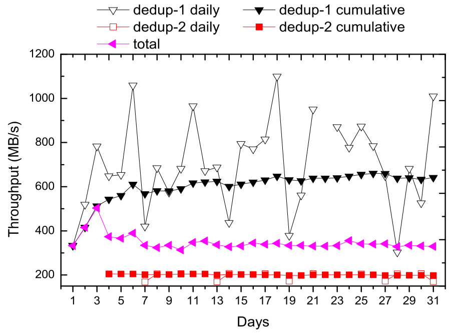


Figure 8: DEBAR throughput over time.

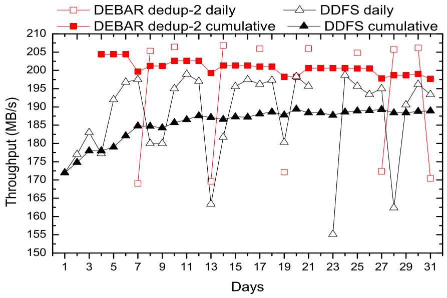


Figure 9: Throughput Comparison of DEBAR dedup-2 and DDFS.

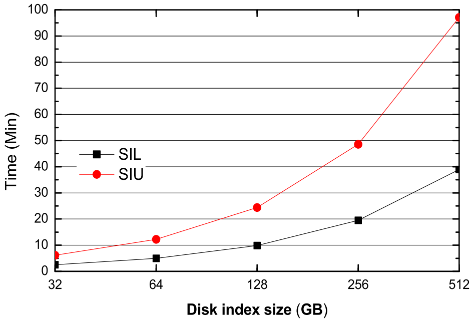


Figure 10: Time overheads of SIL and SIU.

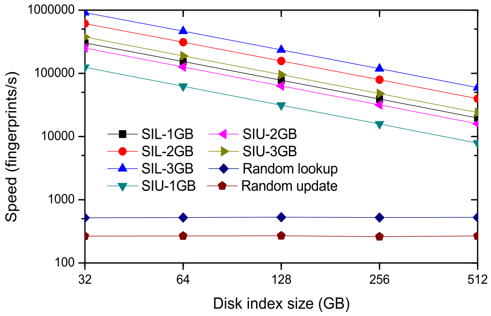


Figure 11: Efficiencies of disk index lookup and update. The y-axis uses logarithm scale to show the speed.

and 376 thousand fingerprints per second respectively, a speedup factor of 1757 and 1392 respectively over the random on-disk fingerprint lookup and update, which achieve speeds of about 522 and 270 fingerprints per second respectively. Even with 512GB disk index and 1GB in-memory index cache (SIL-1GB, SIU-1GB), the SIL and SIU still achieve speeds of about 19660 and 7884 fingerprints per second, over 37 and 29 times faster than the random on-disk fingerprint lookup and update speeds respectively.

The high-speed SIL and SIU give room to use a larger-sized disk index in the singer-server DEBAR system. Based on the results of the one-month experiment on the real-world workload (the HUSt system) and the SIL and SIU time overheads shown in Figure 10, we can easily derive the throughput of the DEBAR system running on different-sized disk index under the real-world workload (the HUSt system). The calculated result is shown in Figure 12 where the x-axis represents different system capacities (8TB, 16TB, 32TB, 64TB and 128TB) supported that in turn correspond to different-sized disk index (32GB, 64GB, 128GB, 256GB and 512GB) used in DEBAR. The result indicates that using a 1GB in-memory index cache for SIL and SIU, a single-server DEBAR can run on a 512GB-sized disk index to support system backup capacity of 64TB while achieve a de-duplication throughput of about 214MB/s (dedup-2 throughput of about 97MB/s). It should be noted that if doubling the size of the in-memory index cache for SIL and SIU, the system backup capacity that DEBAR can support can also be doubled while achieving the same de-duplication throughput.

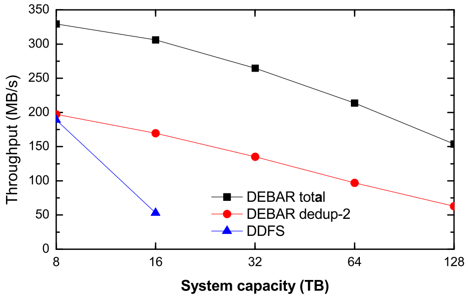


Figure 12: Throughput under different system capacities.

In contrast to DEBAR, using the same amount of memory, DDFS can only support a relatively small physical capacity. Unlike DEBAR, which consumes memory space mainly for SIL and SIU, DDFS consumes memory space mainly for Bloom filter, which represents the fingerprint set of the whole system in order to quickly determine whether an incoming fingerprint has been stored in the system with a low false positive probability. Sup-

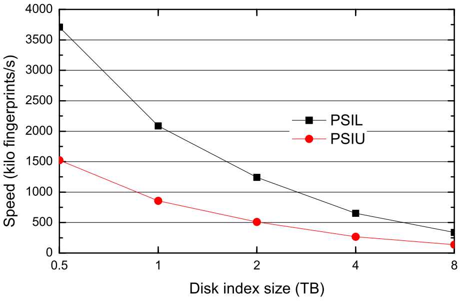


Figure 13: Speeds of PSIL and PSIU with 16 backup servers each with 1GB in-memory index cache.

posing a m-bit Bloom filter, which represents the system fingerprint set with n fingerprints, and k independent hash functions, the false positive probability (See [7]) of the Bloom filter is $\rho = (1 - e^{-\frac{kn}{m}})^k$. Given that $k = (m/n) \ln 2$, the minimum false positive probability is equal to $(1/2)^k$ or $(0.6185)^{m/n}$. Then, using a 1GB in-memory Bloom filter to support one billion fingerprints (For an expected chunk size of 8KB, one billion fingerprints imply a physical backup capacity of about 8TB) such that $m/n = 8$, the minimum false positive probability will be about 2%. If a 1GB in-memory Bloom filter is used to support a 16TB physical capacity such that $m/n = 4$, the minimum false positive probability will quickly increase to about 14.6%! Such a high false positive probability will inevitably result in a high percentage of small random disk I/Os for fingerprint lookups, thus significantly degrading the DDFS performance. We have measured the throughput of DDFS-2 using a 1GB in-memory Bloom filter with $k = 4$ and different m/n values. The measured results, shown in Figure 12, indicate that, for a real-world workload (e.g., the HUSt system) with about 10% new data, the DDFS-2 throughput will quickly drop to under 28% of the original throughput when the Bloom filter m/n value decreases from over 8 to under 5.3, that is, when the amount of data stored in DDFS-2 increases from under 8TB to over 12TB. So, using 1GB memory space, DDFS can support a system backup capacity of no more than 8TB.

6.2 Performance of multi-server DEBAR

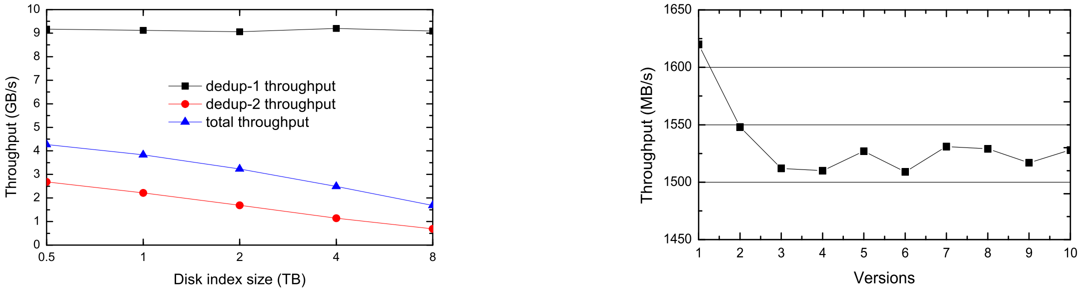
In this section, we evaluate DEBAR scalability by running the system in a total of 13 different modes denoted as (x, y), where x represents the number of backup servers used, y represents the size (GB) of disk index or disk-index part each backup server holds. Specifically, we sequentially run the system in (1, 32), (1, 64), (2, 32), (2, 64), (4, 32), (4, 64), (8, 32), (8, 64), (16, 32), (16, 64), (16, 128), (16, 256), and (16, 512) mode. When each mode finished its test, the system moves to the next higher mode using the property of capacity or performance scaling (Section 4) until finally the system runs under (16, 512) mode.

In the experiment, each backup server uses 1GB memory index cache for PSIL and PSIU, and is equipped with a Highpoint Rocket 2220 Raid controller attached to 8 SATA disks for the disk index and chunk log. In addition, we use a total of 16 HUSt storage nodes each with two gigabit NIC cards to construct a chunk repository to store containers. The backup clients run on 64 HUSt computing nodes, with each backup server receiving data from 4 backup clients in parallel.

We use synthetic datasets generated by the backup client computers for the test. The idea that using synthetic dataset to determine the de-duplication throughput has been proposed in DDFS [30], but DDFS just builds data duplication within a synthetic backup stream and ignores the data duplication among multiple distributed backup streams. To model the real world distributed backup streams in which cross-stream duplication may exist, we make a main improvement to the DDFS method by generating synthetic fingerprint sets rather than datasets and artificially assigning an 8KB-sized data chunk (e.g. a chunk padded with full zero) to each generated fingerprint as the payload of this fingerprint. Using synthetic fingerprints to model real world workload is feasible in practice, since for a de-duplication storage system, it's exactly the data duplication not the data itself that influences system throughput, more precisely, it's exactly the percentage of identical fingerprints in the dataset that influences system throughput and the actual data content is irrelevant. Our method is more like the file system tracing that records file metadata and access activities rather than the actual file data to model a file system workload.

We use a 64-bit variable as input to the SHA-1 algorithm to generate fingerprints since the SHA-1 [27] fingerprints are essentially random and independent of each other no matter how similar the inputs are. Using a variable to generate fingerprints has three main advantages. First, by simply incrementing the variable by 1, one can generate a different random fingerprint, thus eventually capable of generating a sufficient number of random fingerprints (up to 2^{64}). Second, different degrees of duplication can be easily built among distributed backup streams. Finally, the duplicate locality of the real-world backup stream, which is an important feature exploited by the SISL [30] technique, can be simulated in the synthetic model using a contiguous section of the variable value space to generate fingerprints.

The variable value space ($0 \sim 2^{64} - 1$) is divided into 64 none-intersecting contiguous subspaces with each backup-client computer holding a subspace capable of generating up to 2^{58} different random fingerprints. In



(a) Write

(b) Read

Figure 14: Aggregate throughput of DEBAR with 16 backup servers.

each run mode, each backup client simulates a backup stream that is made up of an ordered series of synthetic fingerprint versions where each successor version is generated by performing a series of modification operations on its predecessor version. The version-to-version modification includes, 1) reordering and deleting some of the existing fingerprints, 2) adding new fingerprints using a contiguous section of the variable's corresponding subspace, and 3) adding duplicate fingerprints using a number of small contiguous sections of the variable value space from the previous run modes of the current subspace or other subspaces to simulate the cross-stream duplication. During backup, the backup streams write their synthetic fingerprint versions, along with the data payloads, to the DEBAR backup servers in parallel, with each stream following its version order.

In the experiment, we build about 90% duplicate fingerprints, of which about 30% are cross-stream duplicate fingerprints, to each version, amounting to an average version compression ratio of 10. This compression ratio is reasonably realistic given the real-world examples of the HUSt system and the DDFS experiment in which a compression ratio of over 30 was reported. The cross-stream duplication causes file chunks to spread among distributed storage nodes of the DEBAR chunk repository, thus adversely affecting the read performance of DEBAR. For each backup stream 10 versions of 50GB each are generated.

Figure 13 shows the measured PSIL and PSIU speeds. As expected, multiple backup servers can deliver very high aggregate fingerprint lookup and update speed. Using 16 backup servers each with 1GB in-memory index cache storing fingerprints, PSIL and PSIU achieve about 3710 and 1524 thousand fingerprints per second respectively under 0.5TB-sized disk index. With the increment of the disk index size, both PSIL and PSIU speeds decrease, but even with an 8TB-sized disk index, the PSIL and PSIU still achieve high speeds of about 338 and 135 thousand fingerprints per second respectively.

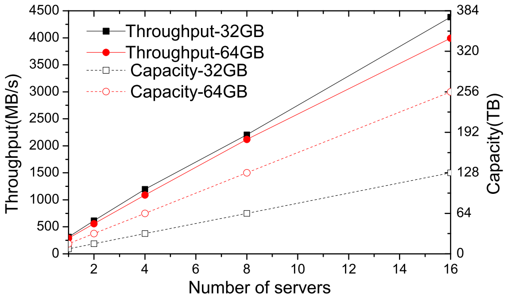


Figure 15: Write throughput and system capacities of multi-server DEBAR.

Figure 15 shows the measured results of the first 10 DEBAR modes. As expected, the system delivers well scalability, using more backup servers, both the aggregate write throughput (calculated by dividing the total amount of logical data backed up by the total amount of time consumed by dedup-1 and dedup-2) and the total system backup capacity supported increase linearly. Larger disk-index part supports larger system backup capacity (Capacity-64GB vs. Capacity-32GB) but at the cost of relatively low write throughput (Throughput-64GB vs. Throughput-32GB) due to the increased time consumed by PSIL and PSIU.

Figure 14(a) shows the aggregate write throughput of the DEBAR with 16 backup servers. The system maintains dedup-1 throughputs over 9GB/s in all the 5 run modes. This is because the preliminary filtering eliminates most of the duplicate chunks in the backup stream thus significantly saving bandwidth for backups. In the experiment, for each run mode, the system has performed 2 dedup-2 processes including 2 PSIL and 1 PSIU (except the 0.5TB disk index mode which includes 2 PSIU and 1 PSIL during the initial deployment of the system). Larger disk index supports larger system capacity but also results in relatively lower write throughput since larger disk index consumes more time for PSIL and PSIU. As shown in Figure 14(a), both dedup-2 throughput and the system total throughput (calculated using the amounts of logical data backed up divides the total amounts of time the dedup-1 and dedup-2 consume) decrease with the increment of the disk index size. Using 0.5TB, 4TB and 8TB-sized disk indexes, with a maximum system capacity of 128TB, 1PB and 2PB respectively, the system achieves total write throughputs of 4.3GB/s, 2.5GB/s and 1.7GB/s respectively.

To determine the read throughput of the multi-server DEBAR, we run the 64 backup clients to read data from the 16 backup servers (each backup server 4 clients) in parallel. Each backup client restores 10 versions, which belong to a backup stream, in the version order. The result is shown in Figure 14(b).

The system delivers high aggregate read throughput for all the versions. It achieves 1620MB/s read throughout for version 1, 1548MB/s for version 2 and stays around 1520MB/s for the later versions. The first version experiences relatively high read throughput because all of its fingerprints are new and hence its data chunks are stored continually at one storage node of the chunk repository. The later versions experience a read throughput decline because they contain more duplicate fingerprints especially the cross-stream duplicate fingerprints that result in data chunk sharing among multiple storage nodes of the chunk repository. But the read throughput of latter versions stays stable around 1520MB/s because of the SISL that preserves duplicate locality and LPC that eliminates most of the random on-disk fingerprint lookups. In our experiment, 99.3% random small disk I/Os for fingerprint lookup were eliminated by LPC. The same phenomenon has been reported in the DDFS test in Ref. [30].

6.3 Discussion

The rapid advances on building powerful many-core CPU are bringing ever-growing performance gap between CPU and storage subsystem. The proposed SIL and SIU techniques judiciously exploit CPU power to compensate for the low speed of disk access especially the notorious random disk access thus matching well against the CPU developing trend, and hence providing larger space for the improvement of the de-duplication storage system performance.

For a PB-scale de-duplication storage system, the metadata storage, such as file metadata and indices that can reach the TB-scale, is an important design issue. We have developed a metadata storage subsystem for the DEBAR director that enables over 250 backup jobs to read or write their metadata concurrently with an aggregate metadata throughput of over 100MB/s. Such a high-performance metadata storage makes it possible to use just one director to support a large-scale DEBAR system with several tens of backup servers. Using a cluster of directors to build an ultra large-scale DEBAR system that stores exabytes of logical data with hundreds of backup servers is a potential challenge for our future work.

De-duplication storage creates heavily chunk sharing among different files and as a side effect, it can make file chunks spread among multiple storage nodes of the chunk repository thus gradually reducing read performance. To solve this problem, DEBAR employs a defragmentation mechanism that automatically aggregates file chunks to one or few storage nodes, thus significantly reducing storage fragmentation and retaining high read throughput.

7 Related Work

A number of systems have proposed different techniques to exploit data duplication to optimize the use of storage space or bandwidth.

EMC Centera Content Addressed Storage (CAS) [1] performs data de-duplication at the granularity of files. It identifies files by the hash of file content to ensure that duplicate files are stored just once in the system. Single Instance Storage (SIS) [5] uses 128-bit file signature derived from file size and hashing of parts of the file content to detect identical files and reclaim their storage space. Since their methods just eliminate duplicate at file level, such systems can achieve only limited storage saving.

To improve compression ratio, block-level de-duplication strategies are commonly used in modern systems. Venti [21] and DDE [13] eliminate duplicate fixed-sized blocks by comparing cryptographic hashes [18, 27] of their contents. Sapuntzakis et al., computes cryptographic hashes of memory aligned pages to accelerate data transfer over low-bandwidth links and improve memory performance [26]. LBFS [20], a network file system, introduced the content-defined chunking algorithm (CDC) which employs Rabin fingerprint [22, 6] to divide files into variable-sized data chunks. Since CDC divides a file based on content instead of length, more duplicates can be detected than the fixed-sized blocking which is sensitive to the shifted contents. CDC has been applied to different contexts for similarity detection or duplicates suppression including backup system [8], web application [25] and distributed file protocols [20, 19]. There are also works focusing on improving chunking effectiveness of the standard CDC algorithm, including the two-threshold, two-divisor (TTTD) chunking algorithm [11, 12] and fingerdiff [4]. In addition, REBL [15], Deep store [28] and TAPER [14] combine advantageous features of different basic techniques such as file-level hashing, content-defined chunking, compression [31] and delta-encoding [23, 10] to get fine grain data de-duplication.

The above studies have been mainly focused on basic methods to achieve more data reduction but not on techniques to achieve high de-duplication throughput. DDFS [30] and Foundation [24] employ Bloom filter and cache techniques to significantly reduce disk index accesses,which improve de-duplication throughput but still suffer from poor scalability for large-scale and distributed de-duplication environments. Instead of using Bloom filter, Lillibridge et al. [16] uses a *sparse index* that exploits sampling and the inherent locality within backup streams to avoid the fingerprint-lookup disk bottleneck for de-duplication backup. The main advantage of this approach over the DDFS solution is that it needs less than half the memory space for an equivalent level of de-duplication. However, the sparse index approach may store duplicate chunks, its de-duplication quality heavily depends on the inherent chunk locality of the backup stream. Moreover, for peta-scale systems, the sparse index can still be too large (reaching tens or hundreds of gigabytes)to fit in one server's memory, and decentralizing the sparse index into multiple servers can incur large amounts of network I/Os for index lookup, which in turn degrades the inline backup performance.

8 Conclusions

In this paper, we present DEBAR, a scalable and high-throughput de-duplication storage system for backup and archiving. DEBAR uses a simple but effective hash method to construct disk index for fingerprint mapping which provides fundamental support to system scalability and high-throughput. For data write, it employs a Two- Phase De-duplication Scheme (TPDS) which uses memory cache and takes advantage of the disk index properties to judiciously turn the random small disk I/Os to sequential large disk I/Os for fingerprint lookup and updating hence achieves high de-duplication throughput.

Our experiments have demonstrated the high performance and scalability of the DEBAR design. Using about 1GB memory cache, a single-server DEBAR achieves over 329MB/s write throughput. While using a cluster of 16 Backup Servers, the system achieves an aggregate throughput over 1.7GB/s and supports 2PB physical capacity.

The DEBAR system provides a framework and basic techniques to build high-performance large-scale de-duplication storage systems for enterprises to protect their ever-growing valuable data.

Acknowledgment

This work was supported by National Basic Research Program of China (973 Program) under Grant No.2004CB318201, National Science Foundation of China under Grant No.60703046 and No.60873028, US National Science Foundation (NSF) under Grant No. CCF-0621526, the Program for New Century Excellent Talents in University NCET-04-0693 and NCET-06-0650, and Wuhan Project 200750730307.

References

[1] *EMC Centera: Content Addressed Storage System, Data Sheet*, January 2008. http://www.emc.com/collateral/hardware/data-sheet/c931-emc-centera-cas-ds.pdf.

[2] A, C., V, V., AND Z., K. Protecting file systems: a survey of backup techniques. In *Proc. of the 6th IEEE/NASA Conf.* (1998).

[3] BLOOM, B. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*. vol. 13, no. 7, pp. 422-426, 1970.

[4] BOBBARJUNG, D. R., JAGANNATHAN, S., AND DUBNICKI, C. Improving duplicate elimination in storage systems. *ACM Transactions on Storage*. vol. 2, no. 4, pp. 424-448, 2006.

[5] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single instance storage in windows 2000. In *Proceedings of the 4th Usenix Windows System Symposium* (August 2000).

[6] BRODER, A. *Some applications of rabin's fingerprinting method. In Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag, 1993.

[7] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: a survey. *Internet Mathematics,* vol. 1, pp. 485-509, 2005.

[8] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (2004).

[9] DENEHY, T. E., AND HSU, W. W. Duplicate management for reference data. Research report, Computer Sciences Department,University of Wisconsin and IBM Research Division,Almaden Research Center, October 2003.

[10] DOUGLIS, F., AND IYENGAR, A. Application-specific delta-encoding via resemblance detection. In *Proceedings of 2003 USENIX Technical Conference* (San Antonio, Texas, USA, 2003), pp. 113–126.

[11] ESHGHI, K., LILLIBRIDGE, M., WILCOCK, L., BELROSE, G., AND HAWKES, R. Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File And Storage Technologies* (2007).

[12] ESHGHI, K., AND TANG, H. K. A framework for analyzing and improving content-based chunking algorithms. Research report, HP Laboratories Palo Alto, September 2005.

[13] HONG, B., PLANTENBERG, D., LONG, D. D. E., AND SIVAN-ZIMET, M. Duplicate data elimination in a san file system. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (2007).

[14] JAIN, N., DAHLIN, M., AND TEWARI, R. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th USENIX Conference on File And Storage Technologies* (2005).

[15] KULKARNI, P., DOUGLIS, F., LAVOIE, J. D., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *Proceedings of USENIX Annual Technical Conference* (2004), pp. 59–72.

[16] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File And Storage Technologies* (2009).

[17] MANBER, U. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference* (San Fransisco, CA, USA, 1994), pp. 1–10.

[18] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of Applied Cryptography*. CRC Press.

[19] MORETON, T. D., PRATT, I. A., AND HARRIS, T. L. Storage, mutability and naming in pasta. In *International Workshop on Peer-to-Peer Computing* (2002).

[20] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (Oct. 2001), pp. 174–187.

[21] QUINLAN, S., AND DORWARD., S. Venti: a new approach to archival storage. In *Proceedings of the USENIX Conference on File And Storage Technologies* (January 2002).

[22] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, May 1981.

[23] R.BURNS, AND LONG, D. Efficient distributed backup with delta compression. In *Proceedings of the Fifth Workship on I/O in Parallel and Distributed Systems* (November 1997).

[24] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, Massachusetts, June 2008), pp. 143–156.

[25] RHEA, S. C., LIANG, K., AND BREWER, E. Value-based web caching. In *Proceedings of the 12th International Conference on World Wide Web* (2003), pp. 619–628.

[26] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2002).

[27] U.S. DEPARTMENT OF COMMERCE/N.I.S.T., NATIONAL TECHNICAL INFORMATION SERVICE, SPRINGFIELD, VA. *FIPS 180-1. Secure Hash Standard*, April 1995.

[28] YOU, L., POLLACK, K., AND LONG, D. D. E. Deep store: an archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Apr. 2005).

[29] ZENG, L., ZHOU, K., SHI, Z., FENG, D., WANG, F., XIE, C., LI, Z., YU, Z., GONG, J., CAO, Q., NIU, Z., QIN, L., LIU, Q., LI, Y., AND JIANG, H. Hust: A heterogeneous unified storage system for gis grid. In *Finalist Award, HPC Storage Challenge, the 2006 International Conference for High Performance Computing, Networking, Storage and Analysis (SC06).* Tampa, FL, November 13-17, 2006.

[30] ZHU, B., LI, H., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File And Storage Technologies* (2008).

[31] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory IT-23* (May 1977).