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

Tianming Yang*, Hong Jiang†, Dan Feng*‡, Zhongying Niu*, Ke Zhou * and Yaping Wan*
*School of Computer, Huazhong University of Science and Technology
Wuhan National Laboratory for Optoelectronics, Wuhan, 430074, China
Email: ytmzqyy@yahoo.cn, niuzhy@gmail.com
‡ Corresponding author: dfeng@hust.edu.cn
† Department of Computer Science and Engineering
University of Nebraska-Lincoln Lincoln, NE 68588, USA
Email: jiang@cse.unl.edu

***Abstract*—Driven by the increasing demand for large-scale and high-performance data protection, disk-based de-duplication storage has become a new research focus of the storage industry and research community where several new schemes have emerged recently. So far these systems are mainly inline de-duplication approaches, which are centralized and do not lend themselves easily to be extended to handle global de-duplication in a distributed environment. We present DEBAR, a de-duplication storage system designed to improve capacity, performance and scalability for de-duplication backup/archiving. DEBAR performs post-processing de-duplication, where backup streams are de-duplicated and cached on server-disks through an in-memory preliminary filter in phase I, and then completely de-duplicated in-batch in phase II. By decentralizing fingerprint lookup and update, DEBAR supports a cluster of servers to perform de-duplication backup in parallel, and is shown to scale linearly in both write throughput and physical capacity, achieving an aggregate throughput of 1.7GB/s and supporting a physical capacity of 2PB with 16 backup servers.**

***Keywords*-backup and archive; data de-duplication; post-processing;**

I. Introduction

Today, the ever-growing volume and value of digital information have raised a critical and mounting demand for the 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 [1, 2] 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 [2–11]. By eliminating duplicate data across the system, a disk-based storage system can achieve far more efficient data compression than tapes. DDFS [7], 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 data file or stream into chunks and eliminate the duplicate copies of chunks within one file or across multiple files. Duplicate chunks are identified by comparing the chunk fingerprints represented by the hash values of chunk contents. Usually, 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 disks, which for the current technology typically amounts to a few hundred fingerprints per second. The Venti system [6], 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 and updates in parallel.

Recently, several new schemes were proposed to improve de-duplication performance. So far these systems are mainly inline de-duplication approaches, which remove duplicate data before it is written to disk, as opposed to post-

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

processing de-duplication that first stores data in a temporary on-disk holding area and removes duplicate data later in a batch mode. These inline systems, such as DDFS [7] and Sparse Indexing [11], exploit duplicate locality to achieve reasonable de-duplication throughput using an appropriate capacity of RAM for a given system scale. Duplicate locality refers to the tendency for chunks in backup streams to reappear in the same groups of chunks. That is, when new backup stream contains a chunk that has been stored somewhere in the system, there is a high probability that other chunks in its locale also have been stored nearby.

DDFS [7] exploits duplicate locality for index caching as well as for laying out chunks on disk in a way that can quickly locate fingerprints that already exist in the system. In addition, DDFS uses an in-memory Bloom filter [12] to determine the existence of a coming fingerprint in the system before actually querying the disk index. If the Bloom filter does not contain the fingerprint, then the disk index does not need to be queried since the Bloom filter does not return a false negative. By using these techniques DDFS can avoid a large number of disk accesses related to index queries thus achieving backup throughput of over 100 MB/s.

Sparse Indexing [11] performs de-duplication by dividing an incoming backup stream into relatively large segments and deduplicating each segment against only a few of the most similar previously-stored segments. To identify similar segments, a small portion of the chunks in the backup stream are chosen as samples, and a sparse index is exploited to map these samples to the existing segments in which they occur. If two segments share a sample, then they may also share many duplicate chunks nearby with high probability due to the inherent duplicate locality within backup streams. By using a very low sampling rate(e.g., one sample roughly every 64 chunks), the sparse index can be made very small to reside in the server memory so that only a few disk seeks are required per segment and thus the fingerprint-lookup disk bottleneck is alleviated.

However, both DDFS and Sparse Indexing are centralized systems and do not lend themselves easily to be extended to handle global de-duplication in a distributed environment. Moreover, the maximum system physical capacity these inline systems support is limited by the server's main memory capacity. DDFS, for example, needs 1GB in-memory Bloom filter to store 2^{30} fingerprints, which results in a reasonably low false positive rate of about 2% [7, 13]. For an expected chunk size of 8KB, 2^{30} fingerprints represent a physical storage capacity of about 8TB. In order to maintain the same 2% false positive rate, the size of the in-memory Bloom filter must linearly increase with the system's physical capacity. For example, a 1-perabyte physical capacity will need at least 120GB in-memory Bloom filter. Sparse Indexing uses less than half of the memory space than DDFS for an equivalent level of system capacity, but at the expense of storing some duplicate chunks, which represents a trade off between the memory overhead and the de-duplication quality.

In this paper, we present DEBAR, a scalable and high-performance DE-duplication storage architecture for Backup and ARchiving, designed to improve capacity, throughput, scalability of data de-duplication in petabyte-scale distributed backup and archiving systems. DEBAR performs post-processing de-duplication instead of inline, which separates the backup process of file fingerprint and index generation, in phase I, from the time-consuming task of de-duplication (i.e., disk index lookups and updates) in phase II. In the phase I (dedup-1), backup jobs are performed and files are transmitted from the backup client to the backup server, where the latter builds the file metadata and indices for each file, and temporarily stores the file data chunks to a local disk log. A file index, which facilitates retrieving files from the system, is a sequence of fingerprints that map to the file chunks. In the phase II (dedup-2), the backup server reads file chunks from the disk log, performs de-duplication to write new chunks to fixed-sized containers that are in turn stored to a chunk repository. Moreover, we exploit an in-memory *preliminary filter* in dedup-1 to improve bandwidth efficiency by eliminating duplicate chunks as much as possible before sending files to the backup server via the networks. So, dedup-1 can effectively alleviate both the disk and bandwidth bottlenecks for backups. We use the preliminary filter based on the fact that a large proportion of data usually does not change between backup sessions and thus substantial redundancy can be eliminated without having to resort to disk index lookups. 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.

We use two batch process algorithms, called *sequential index lookup* (SIL) and *sequential index update* (SIU) in dedup-2 to identify new data chunks and write new fingerprints to the disk index more efficiently. SIL and SIU sort a large number of fingerprints in the memory and then sequentially pass over the disk index for fingerprint lookup and update, thus eliminating random disk seeks. Compared with DDFS, the batch process has the advantage of supporting significantly larger backup capacity using a single DEBAR backup server with the same amount of memory overhead, where memory is used for the batch process in DEBAR while it is used for Bloom filter in DDFS. Compared with Sparse Indexing, DEBAR always performs perfect de-duplication with no duplicate chunks unremoved. More importantly, by decentralizing disk indexing, SIL and SIU can support a cluster of backup servers to perform de-duplication backups in parallel, and thus rendering DEBAR highly efficient, adaptive and scalable.

One disadvantage of DEBAR, relative to inline approaches, is that it requires additional disk space to keep dedup-1 data and increases non-backup-window processing time during dedup-2. However, DEBAR can be easily scaled to multiple servers with a sustained high throughput. Moreover, it supports a larger system capacity, over 8 times larger than DDFS for an equivalent level of memory overhead and throughput, combining the preliminary filter that can eliminate a large number of duplicate chunks during dedup-1. The additional disk-space overhead is minimal for representative workloads, e.g., less than 3% for a typical weekend 16-hour-window full backup with a sustained data transfer rate of 200MB/s. We believe that this small overhead is acceptable, and is more than compensated for by the high scalability of DEBAR.

The rest of this paper is organized as follows: in the next section, we describe the DEBAR architecture. The post-processing de-duplication scheme is described in detail in Section III. Section IV presents the evaluation results including a quantitative comparison with DDFS. Finally, Section V reviews related work and the paper is concluded in Section VI.

II. DEBAR Architecture

The DEBAR architecture, shown in Fig. 1, 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(job ID, job size, and file indices etc.), for the whole system. A user can define job objects through the director 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.When initiating a backup job, the director assigns a backup server to run the job. By monitoring the states of the system, the director can select appropriate backup servers for backup jobs to maintain load balancing.

Backup clients run on machines that have data to be backed up. When doing a backup job, the *Backup Engine* module reads files from the job dataset, divides the file into chunks using the *content-defined chunking algorithm (CDC)* [5], computes the SHA-1 [14] hash (160bits) of each chunk as its fingerprint, and then interacts with the corresponding backup server to send the file chunks needing backup and discard the duplicate chunks. The chunk repository provides a global disk-based storage pool for backup servers to store new chunks. New chunks are stored into fixed-sized (usually several megabytes in size) containers in the logical order that they appear in the backup stream to preserve spatial locality for chunk accesses. A disk index(Section II-B) is used to establish mapping between

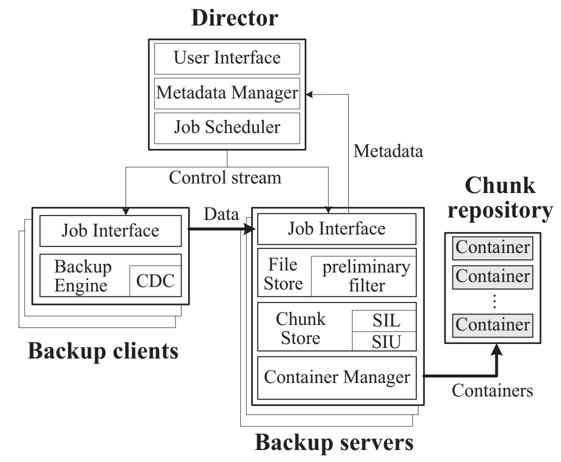

Figure 1. Block diagram of DEBAR.

a chunk and the container holding this chunk within the chunk repository.

A. Backup Server

The backup server performs data de-duplication in two phases (dedup-1 and dedup-2). Backup jobs are performed and finished in dedup-1, while de-duplicated chunks are stored to the chunk repository in 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, and sends the file index and metadata to the director.
- *Preliminary filtering* (see Section III-A) performs preliminary de-duplication to the fingerprint stream to determine which chunks need to be backed up and thus transferred from 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 the backup servers. Each Chunk Store module performs the following operations in dedup-2 to store 'unique' chunks:

- *Index lookup* cooperates with other backup servers to perform parallel *sequential index lookup (SIL)*(see Section III-B) 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.

- *Index updating* cooperates with other backup servers to perform parallel *sequential index update (SIU)*(see Section III-D) to write new fingerprints to the disk index 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)* [7] 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.

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.

B. Disk Index

DEBAR uses a disk index to locate a chunk within the chunk repository.The DEBAR disk index is implemented as a hash table that contains a list of fixed-sized buckets with each bucket containing entries for fingerprints mapped to it. Each entry stores a mapping between a fingerprint and its container ID. Since a fingerprint itself is numerically random in nature, for an index containing a total of 2^n buckets, we simply take the first n bits of a fingerprint as the bucket number to map this fingerprint to its corresponding bucket. In other word, each bucket is responsible for fingerprints with prefix equal to its bucket number.

The disk index should be sufficiently well occupied (i.e., highly utilized) by inserted fingerprints before it needs to be enlarged, since a higher disk index utilization enables a relatively small disk index for a given system capacity, which in turn improves the batch processing (SIL and SIU) performance due to the fact that it consumes less time to scan through the index. Based on both theoretical analysis and extensive experiments [15], we have selected 8KB-sized buckets for the DEBAR disk index in order to achieve over 80% disk index utilization while introducing little additional overhead to the DEBAR system. This bucket size implies that each bucket contains 16 512-byte disk blocks with each disk block storing up to 20 25-byte entries(we use 5-byte container ID), giving rise to an index bucket capacity of up to 320 entries. Given a system backup capacity, the required DEBAR disk index can be determined. For example, a backup capacity of 8TB (with an expected chunk size of 8KB), requires a disk index that can store a maximum of 2^{30} fingerprints. Using the DEBAR disk index that can support a utilization of over 80%, just 2^{22} buckets, for a total of 32GB in size, will be sufficient.

III. Post-processing De-duplication

We have briefly described the post-processing de-duplication scheme in Section I and II. In this section, we further detail the design and implementation of preliminary filtering, SIL, chunk storing and SIU.

A. Preliminary filtering

The *preliminary filtering* is designed to eliminate duplication 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. Since the director maintains job metadata, including the directory structure and file indices of all backup jobs, we can filter duplicate chunks in backups by just loading a small fraction of previously stored fingerprints to the memory.To implement the preliminary filter, one can employ the directory synchronization techniques, such as HHT [10] and HDAG [8], provided that the directory structures are encoded as HHTs or HDAGs by the system. But at present, we just implement a flat filter that stores nonstructural fingerprints for simplicity. Specifically, we implement the preliminary filter 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 $Job_x(t_0)$, $Job_x(t_1)$, $\ldots$, $Job_x(t_n)$ $(t_0 < t_1 < \ldots < t_n)$, which contains all historic versions of the dataset of Job_x, 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 $Job_x(t_n)$.

The preliminary filter is implemented as an in-memory hash table that stores 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 marked as 'new'; meanwhile, its corresponding data chunk $I(F)$ is transferred from the backup client to the on-disk chunk log as group $< F, I(F) >$. Otherwise, the fingerprint F is discarded since it already exists in the filter. 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. Using a preliminary filter with 2^{20} buckets with each bucket storing an array of fingerprint entries, we can store a fingerprint entry using 20 bytes with 2 bytes for cache management and the remaining 18 bytes for the fingerprint(the first two bytes of a fingerprint are implied in the bucket number). Using such a preliminary filter,1-GB memory can store about 53 millions fingerprints, 408GB worth of 'unique' data chunks for an expected chunk size of 8KB. So, 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.

B. *Sequential index lookup*

The SIL workflow is illustrated in Fig. 2. The disk index contains 2^n buckets, and the index cache is an in-memory hash table with $2^m (m \leq n)$ buckets. To perform sequential index lookup, the DEBAR system first reads fingerprints from the undetermined fingerprint files and uses the first m bits of these fingerprints as bucket numbers to insert them to the corresponding buckets of the index cache. 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, since these disk index buckets store the fingerprints whose first m bits constitute the binary number k. So, for the fingerprint lookup, the system just needs to sequentially read large bulks of consecutive buckets from the disk index to the memory. If a fingerprint in the index cache is found in the corresponding disk-index bucket that has been read to the memory, 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 will be used in the chunk storing operation.

SIL is highly scalable, as it intrinsically supports a cluster of backup servers to perform parallel fingerprint lookups. In a large-scale DEBAR system with 2^w backup servers, the disk index can be divided into 2^w parts with backup server $k(k = 0, 1, \ldots, 2^w - 1)$ storing index part k, which maps the fingerprints whose first w bits constitute the binary number k. These 2^w backup servers cooperate to perform *parallel sequential index lookup (PSIL)*, as discribed below. First, each backup server's undetermined fingerprints are divided into subsets according to their first w bits. Then, these 2^w backup servers exchange their subsets to ensure

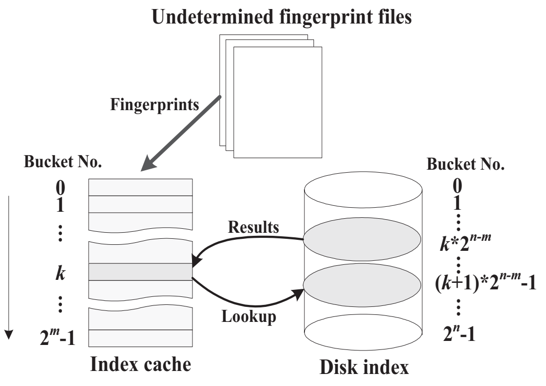

Figure 2. Sequential index lookup.

that backup server k processes the fingerprints whose first w bits constitute the binary number k. After the exchange is finished, backup server k 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 SIL. 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.The above PSIL mechanism effectively eliminates duplicate chunks not only on a single backup server but also across multiple backup servers thus guaranteeing system consistency in distributed de-duplication environments.

C. *Chunk storing*

After completing index lookups, backup server sequentially reads the $< F, I(F) >$ groups (see Section III-A) from the chunk log and refers to the index cache to write new chunks to the containers. Specifically, for a $< F, I(F) >$ group read from the chunk log, Chunk Store checks whether fingerprint F is in the index cache. If not found there, it discards $< F, I(F) >$ as F is not new. Otherwise, it checks whether its corresponding container ID is null. If container ID is null, it writes $< F, I(F) >$ to the container.If the container is full, Chunk Store first creates a new container in the memory for chunk writing, and then submits the full container to the Container Manager to get the container ID. Finally, it writes the container ID to those index cache nodes whose fingerprints and chunks have been stored in this container.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. 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.

D. Sequential index update

We use the SIU technique to improve the disk index update efficiency. The principles of SIU are similar to those of SIL. It first reads the unregistered fingerprints to the index cache and a large bulk 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. 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 and a parallel chunk storing process are performed is usually smaller than that of the undetermined fingerprints checked by PSIL, we can merge small SIU requests with each PSIU servicing the outcomes of multiple PSIL and parallel chunk storing processes. However, merging SIU requests may result in duplicate chunks stored in the system because the metadata of newly stored chunks, such as fingerprints and container IDs, is not written to the disk index in a timely fashion. To solve this problem, we use a simple mechanism as the following:

- 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 by SIU.

In the SIU process, the disk index can become full, that is, there exist a bucket that overflows and finds both of its two adjacent buckets full [15], in which case, DEBAR automatically scales up the index by performing the *capacity enlarge algorithm* that contains only simple bucket-copying operations. Specifically, constructing a new index with 2^n+1 buckets from an old index with 2^n buckets, the enlarge algorithm does the following: copying the entries in bucket $k(k = 0, 1, \ldots, 2^m - 1)$ of the old index 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. 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.

IV. Experimental Evaluation

We have implemented a prototype DEBAR in Linux and a prototype DDFS according to the description of literature [7] 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 [9], 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 a single-server (one backup server) DEBAR under a real-world workload, check the effectiveness of the preliminary filter, analyze the behavior of SIL and SIU, and compare with DDFS in terms of 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, chunk repository 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 one Highpoint Rocket 2220 RAID controller attached to 8 SATA disks. The DEBAR backup server uses 1GB memory for the index cache for SIL and SIU, while DDFS uses 1GB memory for the 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 [5]with an expected chunk size of 8KB. In the first experiment, DEBAR and DDFS each uses 32GB disk index.

A. Performance of single-server DEBAR

The real-world data center in our test is HUSt [16], 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 data several tens of terabyte, 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

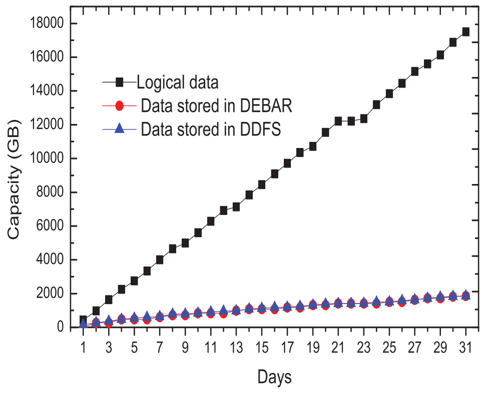

Figure 3. The amount of logical data backed up versus the amount of physical data stored.

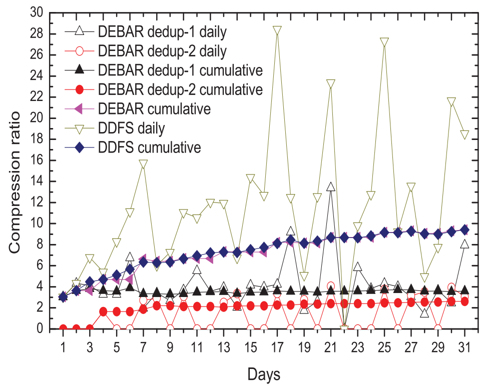

Figure 4. Data compression ratios over time.

server for the backup service, and then run in parallel to send the same data to the DDFS server for their respective backup services.

1) Effectiveness of the preliminary filter: Fig. 3 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 Fig. 4, 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 only runs for a total of 14 times as shown in Fig. 4. 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. The preliminary filter also significantly reduces the bandwidth requirement for backups, thus accelerating the DEBAR dedup-1 throughput. As shown in Fig. 5, DEBAR achieves very high dedup-1 throughputs , giving rise to a cumulative total throughput of 329.2MB/s(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)at the end of the 31st day.

2) Performance of dedup-2: Fig. 6 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 Fig. 6. In contrast to DEBAR dedup-2, which can merge small SIU requests 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 Fig. 6.

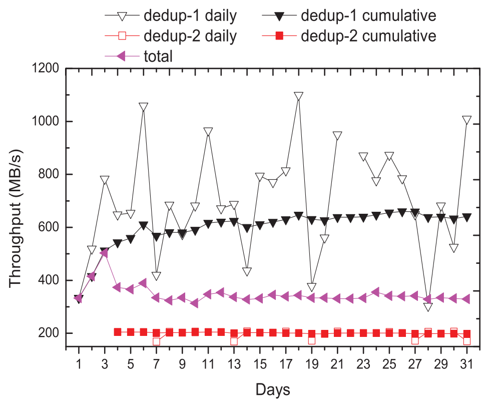

Figure 5. DEBAR throughput over time.

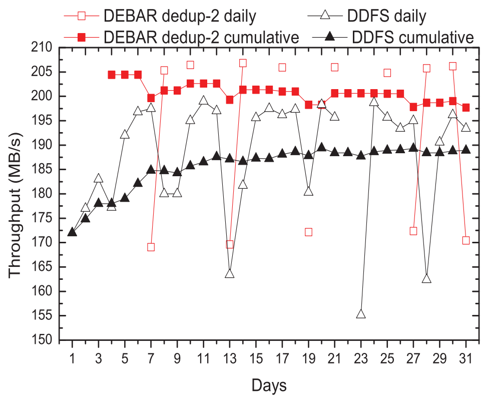

Figure 6. Throughput Comparison of DEBAR dedup-2 and DDFS.

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.

SIL and SIU are effective techniques for improving the efficiency of fingerprint lookup and update. We have measured the time overheads of SIL and SIU by varying the size of disk index and the size of an in-memory index cache on the DEBAR backup server. In the experiment, we find that 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 times increase from 2.53 minutes and 6.16 minutes to 38.98 minutes and 97.07 minutes respectively. With a 32GB disk index and 3GB in-memory index cache that can store about 139 million fingerprints, the measured maximum speeds of SIL and SIU are about 917 and 376 thousand fingerprints per second respectively. Even with a 512GB disk index and 1GB in-memory index cache, SIL and SIU still achieve speeds of about 19660 and 7884 fingerprints per second respectively. Obviously, by using a larger index cache to store more fingerprints, the batch process efficiency can be further improved until the CPU eventually becomes a bottleneck. We measured the speed of fingerprint lookup in main memory using an Intel Xeon DP 5365 3.0 GHz CPU running at 90% utilization, and the result is a speed of about 2.749 million fingerprints per second even with each fingerprint requiring 320 comparison operations. It is precisely this high speed of in-memory fingerprint lookup that motivates us to use a disk index of 8KB-sized buckets(i.e., each bucket can contain up to 320 fingerprints), to compensate for the relatively low speed of disk I/O for SIL and SIU.

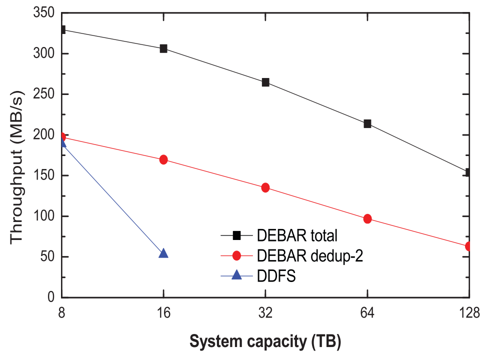

Figure 7. Throughput under different system capacities.

3) System backup capacity: The high-speed SIL and SIU provide ample room for 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 measured, 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 Fig. 7 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 256GB-sized disk index to support a system backup capacity of 64TB, while achieving a de-duplication throughput of about 214MB/s (dedup-2 throughput of about 97MB/s). It should be noted that by doubling the size of the in-memory index cache for SIL and SIU, the system backup capacity that DEBAR can support will also be doubled while achieving the same de-duplication throughput.

In contrast to DEBAR, using the same amount of main memory, DDFS can only support a relatively small physical backup capacity. Unlike DEBAR, which consumes memory space mainly for SIL and SIU, DDFS consumes memory space mainly for the Bloom filter that 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. Supposing an m-bit Bloom filter, which represents the system fingerprint set with n fingerprints, and k independent hash functions, the false positive probability (See [13]) 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 (8TB physical backup capacity for an expected chunk size of 8KB) 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 using a 1GB in-memory Bloom filter with $k = 4$ and different m/n values. The measured results, shown in Fig. 7, indicate that, for a real-world workload (e.g., the HUSt system) with about 10% new data, the DDFS 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 increases from under 8TB to over 12TB. So, using 1GB memory space, DDFS can support a system backup capacity of no more than 8TB.

B. *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 *capacity enlarge algorithm*(Section III-D) until finally the system runs under (16, 512) mode.In the experiment, we use a total of 16 HUSt storage nodes of which each has two gigabit NIC cards used 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 different backup clients in parallel.

We use synthetic datasets generated by the backup-client computers for the test. To model the real world distributed backup streams in which cross-stream duplication may exist, we generate synthetic fingerprint sets and artificially assigning an 8KB-sized data chunk (e.g. a chunk padded with all zeros) to each generated fingerprint as its payload. Using synthetic fingerprints to model real world workload is feasible in practice, since for a de-duplication storage system it is the percentage of identical fingerprints in the dataset that influences system throughput, whereas the actual data content is irrelevant. Our method is akin to 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 [14] 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 [7] 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 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 10:1. 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

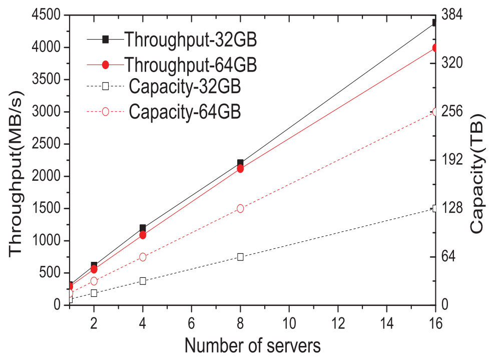

Figure 8. Aggregate write throughput and system capacities of multi-server DEBAR.

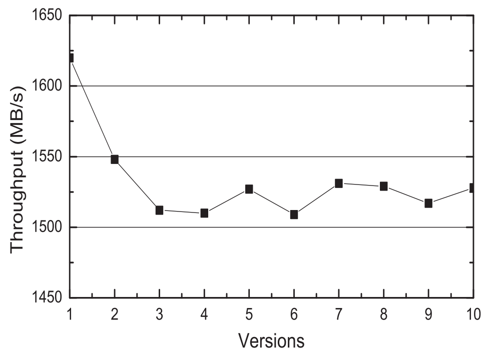

Figure 9. Aggregate read throughput of DEBAR with 16 backup servers.

read performance of DEBAR. For each backup stream 10 versions of 50GB each are generated.

Fig. 8 shows the aggregate throughput and backup capacity, respectively represented by the left and the right axis, as a function of the number of backup servers for both the 32GB- and 64GB-disk-index per server cases. As expected, the multi-server DEBAR scales nearly linearly in 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. Larger disk-index per server supports larger system backup capacity (Capacity-64GB vs. Capacity-32GB) but at the cost of relatively lower write throughput (Throughput-64GB vs. Throughput-32GB) due to the increased time consumed by PSIL and PSIU. In the 128GB-, 256GB- and 512GB-disk-index per server cases, 16 backup servers can support system backup capacity of 0.5PB, 1PB and 2PB respectively, while achieving an aggregate write throughput of 3.2GB/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 serves 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 Fig. 9.

The system delivers high aggregate read throughput for all the versions. The first version induces relatively high read throughput of 1620MB/s because all of its fingerprints are new and hence its data chunks are stored contiguously in 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, which result in file chunks being spread among multiple storage nodes of the chunk repository. But the read throughput of later versions stays stable around 1520MB/s because of containers that preserve 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 [7].

C. Discussion

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. In addition to metadata storage, other important problems such as on-demand data deletion and data reliability should also be well addressed by DEBAR, but we don't discuss them here in the interest of space.

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

Compared to inline approaches such as DDFS, DEBAR requires additional disk space to keep dedup-1 data and increases non-backup-window processing time during dedup-2. Assuming an uninterrupted weekend 16-hour-window full backup using DDFS with a sustained data transfer rate of 200MB/s, the amount of logical data backed up is about 11TB. Using DEBAR to backup the same amount of data and assuming a compression rate of about 3.6:1 by the preliminary filter(using as an example the experimental result of the HUSt system), it would need about 3.06TB disk space to temporarily hold the data in dedup-1. If the DEBAR server holds a 512GB disk index, which supports physical capacity of over 128TB, and uses 2GB memory cache for the batch process in dedup-2, it can achieve a total throughput over 200MB/s (See Section IV-A3), and hence can also complete the entire backup task (both dedup-1 and dedup-2) within 16 hours. In the above case, the additional disk space overhead due to post-processing de-duplication is reasonably small, just about 2.3% (3.06TB /(128TB+3.06TB)). If we deploy multiple backup servers, the system can be scheduled with some servers running in dedup-1 and other some servers running in dedup-2, which hence provides a 24 hour backup window for users. In the future work, we plan to use dedicated index servers (instead of locating the disk index on the backup servers) for fingerprint lookup and update, and study effective schedule algorithm for the post-processing de-duplication backup to achieve high sustained throughput.

V. Related Work

A number of techniques have been proposed to exploit data duplication to optimize the use of storage space and/or bandwidth. The EMC Centera Content Addressed Storage (CAS) [17] 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) [18] uses 128-bit file signatures derived from file size and hashing of parts of the file content to detect identical files and reclaim their storage space. Since their methods only eliminate duplicate at the file level, such systems can only achieve a limited storage efficiency.

To improve the compression ratio, block-level de-duplication strategies are commonly used in modern storage systems. Venti [6] and DDE [3] eliminate duplicate fixed-sized blocks by comparing cryptographic hashes [14] of their contents. Sapuntzakis et al. [19], computes cryptographic hashes of memory aligned pages to accelerate data transfer over low-bandwidth links and improve memory performance. Since fixed-sized blocking is sensitive to the shifted content, most of the systems such as LBFS [5], DDFS [7], Jumbo store [8] and Sparse Indexing [11] divide files into variable-sized chunks based on content instead of length, which eliminates more duplicates.

Recently, scalable high-performance de-duplication is becoming a new research focus of the storage industry and research community where several new schemes have emerged. So far these systems are mainly centralized inline de-duplication approaches, which usually exploit duplicate locality to achieve high throughput with affordable memory overhead. DDFS [7] and Foundation [9] employ Bloom filter and cache techniques to significantly reduce disk index accesses,which improve throughput but suffer from poor scalability for large-scale and distributed de-duplication environments. Instead of using Bloom filter, Sparse Indexing [11] exploits sampling and the inherent locality within backup streams to improve de-duplication performance. 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 system capacity. However, Sparse Indexing may store duplicate chunks, its de-duplication quality heavily depends on the inherent chunk locality of the backup stream.

DEBAR performs post-processing de-duplication by batch-processing fingerprint lookup, which requires few expensive memory for a given system scale and produces perfect de-duplication with high throughput. While DEBAR requires additional disk storage for staging area, it can make up for this drawback by parallelizing backup using a cluster of servers.

DEDE [20] is a decentralized de-duplication technique built atop a SAN clustered file system that provides a virtual machine environment via a shared storage substrate. Each host maintains a write-log to accumulate the hashes of the blocks it has written. Duplicate blocks are identified and reclaimed by querying and updating a shared index in batch in the background. The idea of performing de-duplication out-of-band via merging updates is analogous to our batch-processing approach. But DEDE was designed mainly for live file systems instead of a backup system, thus its de-duplication is best-effort and performed on fixed-sized blocks. While DEDE merges the index querying and updating into one process by taking advantage of the underlying file system abstractions, it's difficult for DEBAR to do so, since new data chunks in DEBAR are stored in the logical order that they occur in a data file or stream to preserve chunk locality for file read, but the sorted fingerprints in SIL have no such a logical order. So, DEBAR performs SIL and SIU in two separate processes and merges small SIU requests to improve efficiency.

VI. Conclusions

This paper presents DEBAR, a scalable and high-performance de-duplication storage system for backup and archiving. DEBAR employs a two-phase de-duplication scheme that separates the backup process of file indexing, in phase I, from the time-consuming task of de-duplication storage in phase II: in the de-duplication phase I, it exploits job chain semantic to eliminate substantial redundancy in backup stream to save bandwidth, while in the de-duplication phase II, it performs fingerprint lookup and update in-batch to improve throughput. The salient feature of this approach is that both the system backup capacity and the de-duplication performance can be 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.

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) support under grand numbers NSF-CCF-0621526, NSF-CCF-0937993, and NSF-IIS-0916859.

References

[1] T. E. Denehy and W. W. Hsu, "Duplicate management for reference data," Computer Sciences Department,University of Wisconsin and IBM Research Division,Almaden Research Center," Tech. Rep., Oct. 2003.

[2] L. You, K. Pollack, and D. D. E. Long, "Deep store: an archival storage system architecture," in *Proceedings of the 21st International Conference on Data Engineering*, Apr. 2005.

[3] B. HONG, D. PLANTENBERG, D. D. E. LONG, and M. SIVAN-ZIMET, "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.

[4] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *Proceedings of USENIX Annual Technical Conference*.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001, pp. 174–187.

[6] S. Quinlan and S. Dorward., "Venti: a new approach to archival storage," in *Proceedings of the USENIX Conference on File And Storage Technologies*, January 2002.

[7] B. Zhu, H. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File And Storage Technologies*, 2008.

[8] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes, "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.

[9] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," in *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2008, pp. 143–156.

[10] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the 4th USENIX Conference on File And Storage Technologies*, 2005.

[11] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of the 7th USENIX Conference on File And Storage Technologies*, 2009.

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

[13] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: a survey," *Internet Mathematics,* vol. 1, pp. 485-509, 2005.

[14] *Secure Hash Standard*, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[15] T. Yang, H. Jang, D. Feng, and Z. Niu, "Debar: A scalable high-performance deduplication storage system for backup and archiving," Department of Computer Science and Engineering, University of Nebraska-Lincoln, Tech. Rep. TR-UNL-CSE-2009-0004, Jan. 2009, http://lakota.unl.edu/facdb/csefacdb/TechReportArchive/TR-UNL-CSE-2009-0004.pdf.

[16] L. Zeng, K. Zhou, Z. Shi, D. Feng, F. Wang, C. Xie, Z. Li, Z. Yu, J. Gong, Q. Cao, Z. Niu, L. Qin, Q. Liu, Y. Li, and H. Jiang, "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, Nov. 13-17, 2006.

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

[18] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in windows 2000," in *Proceedings of the 4th Usenix Windows System Symposium*, August 2000.

[19] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[20] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized deduplication in san cluster file systems," in *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.