# TDDFS: A Tier-Aware Data Deduplication-Based File System

ZHICHAO CAO and HAO WEN, University of Minnesota, Twin Cities, USA
XIONGZI GE, NetApp, USA
JINGWEI MA, College of I.T., Nankai University, Tianjin, China
JIM DIEHL and DAVID H. C. DU, University of Minnesota, Twin Cities, USA

With the rapid increase in the amount of data produced and the development of new types of storage devices, storage tiering continues to be a popular way to achieve a good tradeoff between performance and cost-effectiveness. In a basic two-tier storage system, a storage tier with higher performance and typically higher cost (the fast tier) is used to store frequently-accessed (active) data while a large amount of less-active data are stored in the lower-performance and low-cost tier (the slow tier). Data are migrated between these two tiers according to their activity. In this article, we propose a Tier-aware Data Deduplication-based File System, called TDDFS, which can operate efficiently on top of a two-tier storage environment.

Specifically, to achieve better performance, nearly all file operations are performed in the fast tier. To achieve higher cost-effectiveness, files are migrated from the fast tier to the slow tier if they are no longer active, and this migration is done with data deduplication. The distinctiveness of our design is that it maintains the non-redundant (unique) chunks produced by data deduplication in both tiers if possible. When a file is reloaded (called a reloaded file) from the slow tier to the fast tier, if some data chunks of the file already exist in the fast tier, then the data migration of these chunks from the slow tier can be avoided. Our evaluation shows that TDDFS achieves close to the best overall performance among various file-tiering designs for two-tier storage systems.

CCS Concepts: • **Information systems** → **Hierarchical storage management**; **Deduplication**; • **Software and its engineering** → *File systems management*; *Software performance*;

Additional Key Words and Phrases: Data deduplication, tiered storage, data migration, file system

**ACM Reference format:**
Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David H. C. Du. 2019. TDDFS: A Tier-Aware Data Deduplication-Based File System. *ACM Trans. Storage* 15, 1, Article 4 (February 2019), 26 pages.
https://doi.org/10.1145/3295461

## 1 INTRODUCTION

According to International Data Corporation (IDC), more than 22 zettabytes (billion terabytes) of various storage media will need to be shipped between 2018 and 2025 to meet world data storage demands [1]. The amount of data created by social media, e-business, and other large-scale

This work was partially supported by NSF awards 1421913, 1439622, 1525617, and 1812537.
Authors' addresses: Z. Cao, H. Wen, J. Diehl, and D. H. C. Du, University of Minnesota, Twin Cities, 4-192 Keller Hall, 200 Union Street SE, Minneapolis, MN, 55455; emails: {caoxx380, wenxx159, jdiehl, du}@umn.edu; X. Ge, NetApp, 7301 Kit Creek Road, Research Triangle Park, NC, 27709; email: gexx132@umn.edu; J. Ma, Baidu Online Network Technology Co., Ltd., Beijing, China; email: mjwtom@gmail.com.

1553-3077/2019/02-ART4 $15.00
https://doi.org/10.1145/3295461

IT systems continues to grow exponentially [2]. This growth will cause the scale of some future file systems to become extremely large with billions of files stored in a single system. Several file systems such as GPFS [3], ZFS [4], HDFS [5], and Ceph [6] are already designed to satisfy this extremely large-scale storage requirement. Usually, except for a small number of active files that are used very frequently, most files are not re-accessed for a long period of time after they are first written [7]. Keeping all files, even infrequently accessed ones, in high-performance storage negatively impacts the overall cost-effectiveness of the storage system [8]. Thus, we are looking for approaches to improve the file system performance while reducing the total storage cost of large-scale file systems.

In recent years, different types of storage devices have been developed such as Non-Volatile Memory (NVM), Solid State Drives (SSD), and hard drives that use Shingled Magnetic Recording (SMR). These devices are used to construct different storage tiers with distinct performance characteristics. Typically, in a two-tier storage system, the fast tier has higher performance, higher cost, and smaller capacity while the slow tier has lower performance, lower cost, but bigger capacity [9]. The fast tier is used for storing active, or hot, data blocks while the slow tier is used to store less-active, or cold, blocks. Due to dynamic changes in data activity, data blocks are migrated between the two tiers to make more data accesses occur in the fast tier while most inactive files are stored in the slow tier. Therefore, the performance of the whole system can be close to that of the fast tier, but its cost-effectiveness can be close to that of the slow tier.

To manage storage tiers at the file system level and to achieve a good tradeoff between performance and cost-effectiveness, we propose a Two-tier aware Data Deduplication-based File System called TDDFS. TDDFS allocates and migrates files between the two tiers. TDDFS conducts file-based activity evaluation and manages file-based migrations efficiently and accurately. Files are always accessed from the fast tier as in a regular file system to achieve high performance. An access to a file in the slow tier triggers the file to be migrated to the fast tier. TDDFS also deduplicates the files that are going to be migrated from the fast tier to the slow tier. Since data deduplication identifies redundant data and replaces it with corresponding metadata, TDDFS reduces the amount of data being migrated to the slow tier and improves its space utilization.

TDDFS creates and operates on new files in the fast tier the same as files in file systems without deduplication to gain the best performance. After a period of time, some less-active files are migrated to the slow tier. These files are deduplicated with a Content Defined Chunking (CDC) process, while their file metadata are still maintained in the fast tier. If a file (deduplicated) in the slow tier is requested by a file *open*, then the file will be migrated back to the fast tier. We have designed special ways of handling the files to be migrated from the fast tier to the slow tier or vice versa. To reduce the amount of data to be migrated from the slow tier to the fast tier, TDDFS does not assemble the file back to its original state; instead, it keeps the structure of the data chunks generated by data deduplication. That is, the file recipe created during the deduplication is used to index the data chunks in the fast tier. Therefore, data chunks that already exist in the fast tier do not need to be migrated. In this way, the amount of data to be migrated from the slow tier to the fast tier is reduced. To further reduce the data to be migrated from the fast tier to the slow tier, TDDFS keeps track of whether an existing data chunk in the fast tier has been modified. If a data chunk is not modified and a copy of the chunk exists in the slow tier, then it does not need to be migrated from the fast tier back to the slow tier. Consequently, the amount of data to be migrated between the two tiers is reduced by 80% to 90% compared with the design without deduplication or the design that only applies deduplication to the slow tier.

This article describes the design, prototype implementation, and evaluation of TDDFS. We implemented TDDFS based on FUSE and compared it with other file system level tiering designs such as *AllDedup* (all files in both tiers are deduplicated) and *NoDedup* (no files in either tier are deduplicated). The evaluation results show that the overall performance of the proposed system is close to that of a comparable file system with the *NoDedup* tiering design, and its cost-effectiveness is close to that of the *AllDedup* design. This means TDDFS achieves close to the best overall performance and cost-effectiveness among various file-tiering designs for a two-tier storage system. For example, as shown in Section 5, the throughput of TDDFS can be close to or even better than that of *NoDedup*, and the per-GB cost of TDDFS is only about 2% higher than that of *AllDedup*. Our key contributions include:

- By keeping deduplicated unique data chunks in both tiers and creating new files as regular files in the fast tier, TDDFS simultaneously achieves high performance and high cost-effectiveness.
- When a file is migrated from the slow tier to the fast tier, if some of the data chunks of the file already exist in the fast tier, then these chunks will not be migrated. Thus, TDDFS reduces the transfer costs of migrating data between the slow tier and the fast tier.
- Data chunks are shared among reloaded files in the fast tier. Consequently, TDDFS achieves a good space saving in the fast tier (no duplicate data chunks) with only a small read/write performance penalty.
- TDDFS improves the deduplication performance and migration efficiency. This is because the unmodified data chunks of the reloaded files in the fast tier do not need to be deduplicated, and these chunks will not be migrated to the slow tier if they already exist in the slow tier.

The rest of the article is arranged as follows: We discuss both background and challenges in Section 2. Section 3 presents the proposed system design and architecture in detail, and our prototype implementation is described in Section 4. We evaluate and analyze the performance of our approaches in Section 5. The related work is reviewed in Section 6. Finally, we conclude our work in Section 7.

## 2 BACKGROUND AND CHALLENGES

In this section, we first briefly discuss tiered storage, file activity, and file access locality. Then, the challenges of integrating data deduplication with file migration are discussed.

### 2.1 Tiered Storage

Some of the tiered storage products are managed at the block level [10–13]. Typically, a tiered storage system uses either flash-based storage drives or high-performance hard disk drives as a fast tier and uses either low-performance hard disk drives or tape drives as a slow tier [10–13]. The migration granularity in a tiered storage system is a data chunk with the size varying from 4KB to 1GB or even larger [11–13]. A smaller chunk size is usually more accurate and efficient for selecting data chunks to be migrated, but its maintenance cost is higher, and it may cause more data fragmentation. A larger chunk size reduces the management overhead; however, the data migration efficiency and accuracy are relatively low.

Deploying file systems on top of a block-based tiered storage system is a straightforward approach to achieve good cost-effectiveness. As the number of files continuously increases, their corresponding data volume will go beyond the capacity of the fast tier. Therefore, a number of files have to be migrated from the fast tier and stored in the slow tier. In this case, data migration is completely managed by the block-based tiered storage system [9, 14, 15]. The physical location of a file is transparent to the file system, and existing file systems and applications can be directly used in a tiered storage environment without any changes.

When deploying a file system on top of a tiered storage system, some potential issues still need to be considered. First, since data activity measurements and data migration are chunk based, the need to migrate one file may cause the whole data chunk to be migrated. This would consume more storage I/O bandwidth of the two tiers than necessary. Similarly, a small but active file may prevent a big data chunk from being migrated to the slow tier. This would mean that the valuable space of the fast tier is not economically used. Second, a file system may easily collect certain information like file access patterns, file activity, and file boundaries that can benefit data allocation and migration between different tiers. However, the collected file system information cannot be easily passed to the block level and used by the tiered storage management system, though there are existing efforts focusing on passing high-level hints to low-level storage layers [16, 17]. Therefore, moving the migration management logic up to the file system level can potentially solve some of the aforementioned issues and make the management more efficient and accurate. Third, some file system–related data like file metadata, which is small but latency sensitive, should not be migrated to the slow tier if space is adequate. A tiered storage system, without knowing the semantics of the data, may migrate the file metadata to the slow tier if the data chunk that contains the metadata has not been used for a long time. This will cause higher latency when the file is accessed later and for other file system operations that depend on the metadata (e.g., the *ls* command used to list directory contents).

Although file systems that can manage multiple storage tiers have been developed such as GPFS [18], ReFS [19], and Oracle HSM [20], determining policies that allow a file system to manage multiple storage tiers with good tradeoffs between performance and cost remains challenging. Deduplication adds another tradeoff. By deduplicating the files that are migrated to the slow tier, migration cost is reduced and the space utilization of the slow tier is improved. However, data deduplication brings a performance penalty, especially when applications are accessing the deduplicated files. Thus, the challenges of integrating data deduplication with file migration need to be carefully addressed. In this article, we combine the management of tiered storage at the file level with data deduplication and address the associated challenges.

### 2.2 File Activity and Access Locality

To better understand file activity and access locality, researchers have measured and analyzed file system workloads. Most of these studies have similar findings on file activity and file access patterns [7, 21, 22]. Leung et al. [7] collected 3 months of file system workload traces from NetApp's data center used by more than 1,500 employees. They discovered that more than 90% of the over 22TB of active storage was untouched over the 3-month period. Of the files accessed, 65% were only opened once and 94% were accessed fewer than 5 times. Meyer et al. [23] collected file system traces from 857 desktop computers at Microsoft and focused on file modifications and file writes. Analysis of the month-long file traces showed that less than 10% of files were modified within the last 10 days and more than 80% of files were not modified within the last month. From these studies, it is easy to conclude that in large-scale file systems most files are less active and infrequently accessed. It also implies that these files can be migrated to the slow tier to reduce hardware costs if the tiered designs are used.

Leung et al. [7] also found that more than 60% of file re-opens occurred within 1 minute of these files being closed. This result indicates that a file may be opened and closed a couple of times within a short period. Thus, it is reasonable to migrate files from the slow tier to the fast tier to achieve better performance when the files are accessed. In addition, file operations display spatial locality. Based on the traces, Leung et al. concluded that about 21% of file operations accessed file data (reads or writes) while about 50% were strictly metadata operations. Thus, in a two-tier storage system, storing the metadata of hot files, or even all of the metadata if possible, in the fast tier can improve the overall performance. In general, all metadata plus the frequently accessed files are still a small portion of the total data in a file system. Therefore, storing these data in the fast tier can achieve good performance and high efficiency if the capacity of the fast tier is large enough. In another study, Roselli et al. [22] found that files tend to be either read-mostly or write-mostly, and the total number of file reads tends to be several times more than the total number of file writes. The proposed design of TDDFS incorporates these findings into tradeoffs between performance and cost-effectiveness for reads and writes.

## 2.3 Deduplication and Challenges of Integrating Deduplication with Migration

Data deduplication is a popular feature in many storage software products. By identifying and only storing unique data chunks, data deduplication considerably reduces the space required to store a large volume of data [24]. Data deduplication is broadly classified into two categories: primary data deduplication and secondary data deduplication. Primary deduplication usually refers to deduplicating primary workloads, and secondary deduplication refers to using deduplication in backup or archival systems [25].

Both primary data deduplication and secondary data deduplication have similar processing operations [26, 27]. Typically, the data to be deduplicated forms a byte stream and is segmented into multiple data chunks according to a chunking algorithm. Fixed-size chunking and CDC are two major types of chunking algorithms [28]. The former has a higher throughput, while the latter can achieve a better deduplication ratio (i.e., the original data size divided by the data size after deduplication, which is always $\geq$1). A chunk ID based on the fingerprint of the data chunk content is calculated by a cryptographic hash function (e.g., MD5, SHA-1, or SHA-256). Once a chunk is determined, the deduplication system searches for the chunk ID in the indexing table (i.e., the collection of all existing chunk IDs). If the chunk ID already exists, then the data chunk is a duplicate and it will not be written to the storage again. If the chunk ID is new (does not exist), then the data chunk is unique and will be stored. Then, a new indexing entry that maps this chunk ID to the physical location of the chunk will be created and added to the indexing table. Other mapping metadata (i.e., a file recipe) are created to map all data chunks (redundant or unique) to their stored locations in the same sequence as they were in the original byte stream such that it can be reconstructed in the future.

Although integrating data deduplication with file migration can improve the slow tier space utilization and potentially reduce migration cost, it also brings some performance issues such as high compute and memory resource utilization, high latency, and low throughput [29–31]. The chunking process, chunk ID generation, and chunk ID searches in the indexing table are time-consuming. Thus, data deduplication increases the I/O latency and impacts the throughput [32–34]. Caching a large portion of the indexing table in memory can increase the chunk ID search speed, but the memory space available for the file system or other applications will be reduced, which may further degrade the overall performance. To address these challenges, TDDFS does not deduplicate the files that are newly created in the file system. The performance overhead caused by data deduplication can be avoided when these files are read or written. Also, TDDFS does not assemble the reloaded files (i.e., deduplicated files to be migrated from the slow tier to the fast tier) back to their original state in the fast tier. This avoids creating a file with many duplicate data chunks, since some data chunks may already exist in the fast tier. By tracking the updates to data chunks of a file in the fast tier, extra deduplication can be avoided if the data chunks are not modified. These concepts are discussed in more detail in Section 3.

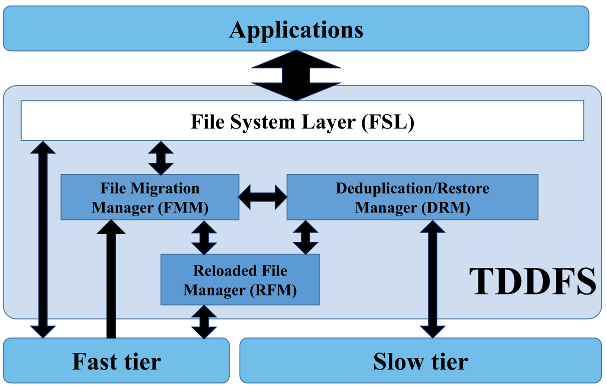

Applications
File System Layer (FSL)
File Migration Manager (FMM)
Deduplication/Restore Manager (DRM)
Reloaded File Manager (RFM)
TDDFS
Fast tier
Slow tier

Fig. 1. Overview of TDDFS architecture.

## 3 SYSTEM ARCHITECTURE AND DESIGN

In this section, we first overview the architecture and important modules of TDDFS including the File System Layer (FSL), File Migration Manager (FMM), Deduplication/Restore Manager (DRM), and Reloaded File Manager (RFM). Then, four important and unique features of TDDFS are described. Finally, several major data operations are discussed. This description of TDDFS is intended to show the design principles of our proposed approach.

### 3.1 Architecture Overview

An overview of the TDDFS architecture is shown in Figure 1. The FSL provides a generic interface identical to that of a typical file system, so upper-layer applications can use TDDFS without any modifications. The FSL allows applications to directly read files from and write files to the fast tier until the files are migrated to the slow tier. Since all metadata are stored in the fast tier, the FSL processes all metadata requests directly on the fast tier. The FMM collects file information from the FSL to evaluate the file activity and manage the migration process between the two tiers.

In addition to the FMM, two other modules, the RFM and DRM, cooperate to support the major operations of TDDFS. When file migration starts, the FMM selects the file migration candidates and, if these files are newly created, tells the DRM to read the candidate files from the fast tier as a byte stream and deduplicate them. After the files are successfully deduplicated and stored in the slow tier, their data are deleted from the fast tier. The file recipe that is used for future file restore and read/write operations is stored in the fast tier as part of the file metadata. If a deduplicated file in the slow tier is requested by a file *open* call from an application, then the FSL tells the FMM to migrate the file from the slow tier to the fast tier (this action is called a reload). During this process, the DRM identifies, reads, and then writes the required unique data chunks from the slow tier to the fast tier. Note that some of the unique data chunks may already exist in the fast tier. These data chunks will not be reloaded. A unique feature of TDDFS is that the reloaded file references its data through an index of the unique data chunks in the fast tier. These data chunks may be shared by the same file or different reloaded files. Note that read or write requests may happen during a file reloading process, and TDDFS is able to process the read and write requests concurrently with file reloading. Another optimization is that when a reloaded file is selected to be migrated from the fast tier to the slow tier again, unmodified data chunks do not need to go through the deduplication process. Only modified data chunks of a previously reloaded file that is now moving to the slow tier will be deduplicated by the DRM. File migration and reloading are transparent to the applications.

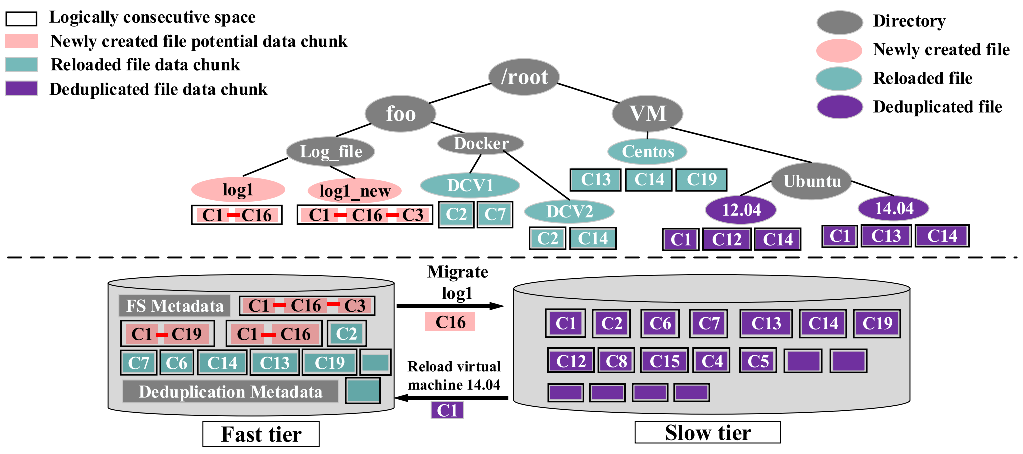


Fig. 2. An example of file states and data layout of TDDFS.

Based on the file's location and state, files in TDDFS are classified into three categories:

- **Newly created files:** These are files newly created in the fast tier. Their metadata and data are the same as files in a typical file system. Since there is no additional operation needed for newly created files, their performance is similar to files in a typical file system deployed on the fast tier.
- **Deduplicated files:** These are files migrated from the fast tier to the slow tier that went through the deduplication process against the unique chunks already stored in the slow tier. Their metadata are stored in the fast tier; however, their unique data chunks are stored in the slow tier without duplicates. TDDFS creates a file recipe for each file as part of the metadata, and the file recipe is used to access unique data chunks of this file in the slow tier.
- **Reloaded files:** These are files reloaded back from the slow tier to the fast tier. During the reloading, only unique data chunks that do not exist in the fast tier are reloaded back to the fast tier. Some data chunks in the fast tier are referenced by only one reloaded file, while others are referenced several times by one file or concurrently referenced by multiple reloaded files. After reloading, the file recipe is updated with data chunk addresses located in the fast tier.

One example of different file states and their data layout in TDDFS is shown in Figure 2. In the figure, `log1` and `log1_new` are newly created files, and `log1_new` is a new version of `log1` with newly appended content. `DCV1`, `DCV2`, and `Centos` are reloaded files. `DCV1` and `DCV2` are the data volumes of Docker containers. Chunk C2 and C14 are shared by the reloaded files. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files. Since `log1_new` is a newly created file, its data content, indicated as three chunks, C1-C16-C3, is stored in the fast tier in a logically consecutive space and managed as a regular file. Note that C1, C16, and C3 are the data chunks that will be partitioned by deduplication in the future, and C3 is the newly appended content based on `log1`. When TDDFS decides to migrate the newly created file `log1` (which has content indicated by two chunks, C1-C16) from the fast tier to the slow tier, the FMM reads the file data and passes it to the DRM. The DRM partitions the file into two data chunks, C1 and C16. Because C1 already exists in the slow tier, only C16 will be migrated and written to the slow tier. This saves I/O bandwidth and improves space utilization of the slow tier. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files and have two identical data chunks, C1 and C14, in both files. However, only one copy of C1 and C14 is stored in the slow tier. If `Ubuntu/14.04` (consisting of data chunks C1, C13, and C14) is reloaded back to the fast tier, then only one data chunk, C1, needs to be reloaded to the fast tier. This is because C13 and C14 are parts of the reloaded file `Centos`, and they are already in the fast tier. This type of optimization reduces the I/O loads of both the fast tier and slow tier.

Given the above overview, the following four subsections present four features that make TDDFS special and achieve better tradeoffs between performance and cost-effectiveness.

## 3.2 File Migration Selection

As more newly created files and reloaded files are accumulated in the fast tier, the free space in the fast tier eventually becomes lower than a given threshold. At this point, some less-active files need to be migrated to the slow tier to reclaim space for more newly created or reloaded files. As discussed in Section 2.2, most files in a file system are less active, and they will not be re-accessed for a long period of time. Therefore, migrating these files to the slow tier can potentially improve the overall performance of the file system. To ensure the performance of primary workloads, migration can be done during the off-peak time. Dedicated evaluations of file activity and file selection policies can be further exploited for file migration in TDDFS. However, this is not the focus of this article. Here, TDDFS is simply based on the file access recency to select file migration candidates. More aggressive or efficient policies will be explored in our future work.

For selecting migration candidates, the least recently used files are identified by the FMM and migrated to the slow tier until the free space of the fast tier is higher than a pre-determined threshold. To speed up the deduplication process of migration, files that are smaller than a *maximal chunk size* (e.g., 16KB) will not be chunked. In this case, the whole file is treated as a single chunk and stored in the slow tier if it is unique. If these small files only occupy a small portion of the fast tier space and there is little probability of finding their duplicates in the slow tier, then migrating and deduplicating these files has little performance and space benefit. Therefore, to further reduce the migration cost, if the cumulative total size of these small files is smaller than a pre-determined threshold (e.g., 5% of the fast tier's capacity), then these small files will not be selected as migration candidates.

## 3.3 Integrating Deduplication with File Migration

When migrating a newly created file through deduplication, a file recipe is created and added to the file metadata to reference the data chunks of this file. File recipes will be used when reloading deduplicated files to the fast tier, reading/writing reloaded files, and deduplicating reloaded files. Each metadata entry corresponding to a data chunk in the file recipe stores the following information: *(Chunk ID, Chunk Size, Chunk Address, Restore Flag, and Modification Flag). Chunk ID* is used to find the corresponding Key-Value Pair (KV-pair) in the deduplication indexing table. In TDDFS, we use the hash value of the chunk as the *Chunk ID*, and this value is related to the cryptographic hash function being used. A unique data chunk may exist in both the fast tier and the slow tier. If the unique chunk is referenced by a deduplicated file, then the *Chunk Address* of this chunk in the file recipe points to the chunk location in the slow tier. If the unique chunk is referenced by a reloaded file, then the *Chunk Address* in the file recipe points to the chunk location in the fast tier. A *Restore Flag* (RF) in the metadata entry is used to indicate whether this chunk has been restored to the fast tier. The RFs of deduplicated files are 0. After a data chunk of the deduplicated file is migrated to the fast tier, the RF of this chunk in the metadata entry is set to 1. One bit is used for the *Modification Flag* (MF) in the metadata entry of a data chunk in the file recipe. When a file is reloaded from the slow tier to the fast tier, the MFs of metadata entries of these migrated chunks are set to 0. During the file's lifetime in the fast tier, if one of its chunks is modified, a copy of the modified chunk is created in the fast tier. At the same time, in the metadata entry of this new

Table 1. KV-pair Structure

| Key | Value |
|---|---|
| Chunk ID | Chunk fast tier address: P1<br>Chunk fast tier reference count: C1<br>Chunk slow tier address: P2<br>Chunk slow tier reference count: C2<br>Chunk size: L<br>Status: S |

chunk, the *Chunk ID* is cleaned and cannot be used to identify the original chunk. The *Chunk Size* and the *Chunk Address* will be changed accordingly to reference the new data chunk, and the MF of this chunk is set to 1.

In the indexing table, each unique chunk's indexing information is managed as one KV-pair as shown in Table 1. The chunk ID is used as the key, and other information is stored as the value. We use 8-byte unsigned integers to store the addresses and reference counts and 4-byte unsigned integers to store the size and status. The size of a KV-pair is 24 bytes plus the chunk ID size. For one KV-pair, unlike existing deduplication systems, we use two address pointers (one for the fast tier address and another for the slow tier address) and two reference counts (fast tier reference count and slow tier reference count) to indicate the chunk location and its usage in the fast tier and the slow tier. The fast tier (or slow tier) reference count indicates how many times the chunk appears in that tier's files. If the reference count of the chunk is 0, then it will be deleted in the corresponding tier during garbage collection. We use the Status to indicate the storing status of the chunk in both tiers: existing in the fast tier only, existing in the slow tier only, or existing in both tiers. After garbage collection is applied in the fast tier or the slow tier, the Status of the deleted chunks is updated for corresponding KV-pairs.

When a newly created file is selected to be migrated to the slow tier, the file content is first read as a byte stream and segmented into multiple data chunks via a CDC scheme. Then, the fingerprint of each chunk is generated using a cryptographic hash function (e.g., MD5, SHA-1, or SHA-256), and the fingerprint is used as its chunk ID. Each chunk ID is searched in the deduplication indexing table. If the data chunk cannot be found in either the fast tier or the slow tier, then this chunk is new to the system and the data chunk will be written to the slow tier. A new KV-pair representing this new chunk is created and inserted into the indexing table. The slow tier reference count in the new KV-pair is set to 1, and the data chunk location in the slow tier is stored in its slow tier address. If the data chunk exists in the fast tier but it does not have a copy in the slow tier, then we need to store a copy of this chunk in the slow tier. At the same time, the slow tier reference count in the KV-pair of this chunk is set to 1, and the data chunk location in the slow tier is stored in its chunk slow tier address. If the data chunk already exists in the slow tier, then the data of the chunk will not be stored. The slow tier reference count in the KV-pair of this chunk increases by 1. After the KV-pair of the data chunk is created or updated accordingly, a new metadata entry of the chunk is created and appended to its file recipe. Finally, after the whole file is processed, the metadata of the file is updated and the file is labeled as a deduplicated file. After that, the content of the file in the fast tier is deleted.

If any failure happens during the migration, then the process will restart from the beginning. If all newly created files are migrated to the slow tier and there are no other newly created files being created, then the whole fast tier space is occupied by reloaded files. In this extreme situation, the performance of TDDFS is close to that of the *AllDedup* design. The size of the file recipe is about 0.5% to 1% of the entire file size if we use 8KB as the average chunk size. If the size of the deduplicated files is very large, then the total size of the corresponding file recipes can occupy a large portion of the fast tier. In this situation, there will be less space for reloaded files and newly created files, which can cause a performance penalty. To handle this case, a threshold is set for the cumulative total size of all the file recipes in the fast tier (e.g., 10% of the fast tier's space). If the total size of the file recipes is higher than the threshold, then some of the least recently used file recipes will be migrated to the slow tier.

When a reloaded file is selected to be migrated from the fast tier to the slow tier, TDDFS sequentially processes the metadata entries in the file recipe of the file. For each metadata entry, TDDFS first checks the value of MF. For an unmodified chunk (MF is 0), we use its chunk ID to fetch the corresponding KV-pair from the indexing table. If the slow tier reference count is >0, then the chunk has a copy in the slow tier. Its *Chunk Address* in the recipe is directly changed from the chunk fast tier address to the chunk slow tier address. Thus, there are no data being read from the fast tier and written to the slow tier. If the slow tier reference count in its KV-pair is 0 and the data chunk has already been garbage-collected in the slow tier according to its Status value, then a copy of the chunk is written to the slow tier and the chunk location is stored in its slow tier address in the KV-pair. Then, the *Chunk Address* in the recipe is updated with the slow tier address. In both cases, the chunk fast tier reference count decreases by 1, and its slow tier reference count increases by 1 in its KV-pair. If the MF is 1 (i.e., the chunk has been modified during its lifetime in the fast tier), then the chunk will go through the same deduplication process as newly created files. Usually, one write will change multiple consecutive data chunks. Thus, the consecutively modified chunks will be deduplicated together to improve efficiency. New data chunks might be generated during the deduplication process of the modified chunks. Therefore, the whole file recipe will be rewritten with the updated metadata entries and newly generated metadata entries. After the data chunk is deduplicated, its RF and MF are both set to 0 in its metadata entry. Finally, the migration process of a reloaded file from the fast tier to the slow tier is completed, and the file is labeled as a deduplicated file. Files are exclusively locked from write requests during the deduplication process to maintain their consistency. To ensure data reliability and consistency, file recipe changes in memory are frequently synchronized with the copy in the fast tier (e.g., every 500ms). If a system failure happens before the most current synchronization, then the file will roll back to the state of the last synchronization.

Each unique data chunk is represented by a KV-pair in a Key-Value Store (KVS), which is used as the deduplication indexing table. Each data chunk related operation will trigger reads or updates of the corresponding KV-pair. If the data chunk is garbage collected in both tiers, then the corresponding KV-pair will be deleted. Since the KV-pairs are very frequently searched and updated, a high-performance flash-adaptive KVS is needed in TDDFS. The KVS design and implementation are not the research focus of this article, and existing flash-adaptive KVS designs such as BloomStore [35], ChunkStash [36], and RocksDB [37] can be used in TDDFS as the indexing table. As the total amount of data in TDDFS increases, the indexing table itself can grow to a large size. If we use a SHA-1 hash as the chunk ID, then one KV-pair is 44 bytes. If we use 8KB as the average chunk size, then the KVS size is about 0.5% of the total stored data size. How to design and implement a high-performance and low-storage-overhead indexing table that can store a large amount of KV-pairs can be an interesting and important research work in the future. In our prototype, we use BloomStore for the KVS as discussed in detail in Section 4.2.

### 3.4 Non-Redundant Chunk Reloading

When a deduplicated file is *opened*, the request will trigger the whole file to be reloaded from the slow tier back to the fast tier. In most primary or secondary deduplication systems, the file restoring or reloading process assembles the whole file back to its original state (i.e., one consecutive file). Thus, it requires all the data chunks of the file to be read from the slow tier and written to the fast tier. If the reloaded file is later migrated to the slow tier again, then the whole file will be deduplicated the same way as a newly created file, even if it has not been modified or was only partially modified. By having no extra indexing structure to track the updates to the file and no maintaining data chunk information in the file recipe, previously obtained deduplication information cannot benefit the migration process.

In contrast, TDDFS applies non-redundant chunk reloading so reloaded files can maintain a similar kind of chunk-level sharing in the fast tier as deduplicated files do in the slow tier. During file reloading, only the relevant chunks that are not in the fast tier are read from the slow tier and written to the fast tier. The performance of the whole system can benefit from non-redundant chunk reloading. One benefit is that the amount of data to be migrated is reduced, since some data chunks that already exist in the fast tier do not need to be migrated. In addition, space is saved in the fast tier due to the data chunk sharing among reloaded files.

The process of migrating deduplicated files from the slow tier to the fast tier and transforming these files to reloaded files is described by the following steps. (1) The DRM reads the file recipe from the fast tier via the RFM. (2) The DRM examines the metadata entry of each chunk and finds the corresponding KV-pair in the indexing table. (3) If the fast tier reference count of the data chunk is 0, which means the chunk currently does not exist in the fast tier, then TDDFS reads the chunk from the slow tier and then writes it to the fast tier with help from the RFM. After this, the fast tier chunk address in the metadata entry of the chunk is updated accordingly by the RFM. The DRM updates the fast tier reference count to 1 and reduces the slow tier reference count by 1 in the corresponding index table KV-pair entry. (4) If the chunk's fast tier reference count is greater than 0, which means the chunk already exists in the fast tier, then its fast tier reference count in the KV-pair entry increases by 1, its slow tier reference count decreases by 1, and the fast tier address in the metadata entry of the chunk is updated to its current fast tier address. (5) The RF and MF are set to 0 in its metadata entry. (6) Steps (2) to (5) are repeated until the whole file recipe is processed. Finally, the file state is changed from deduplicated file to reloaded file.

Usually, flash-based SSD technology is used for the fast tier, and this raises concerns about the wear-out problem. Non-redundant chunk reloading can reduce the amount of data written to the fast tier, which can potentially expand the lifetime of the SSD storage. According to Leung et al. [7], if a file is accessed, then it will probably be re-accessed frequently within a short period of time. Therefore, by reloading accessed deduplicated files from the slow tier to the fast tier, the file system's overall performance will be much better than if these files are kept in the slow tier without reloading. However, it may potentially be wasteful to reload all data of a large file to the fast tier if the file is only partially accessed. With additional information from the FSL or upper-layer applications, there could be a more efficient way to load a portion of the file to the fast tier. Since designing partial file reloading is not the main focus of TDDFS, we reload whole files from the slow tier to the fast tier when they are accessed to simplify the implementation.

### 3.5 Shared Chunk Management

Since some of the data chunks in the fast tier are shared among reloaded files, the file read and write operations should be carefully handled to ensure the data correctness, persistence, and consistency. TDDFS relies on the RFM to cooperate with the FSL to achieve the shared chunk management. Here, we describe how the read and write requests to a reloaded file are processed by TDDFS.

Several steps are needed to finish a write request. When a write request comes, according to the offset in the file and requested data size, the RFM locates the corresponding data chunks by checking the file recipe. First, for each chunk, the corresponding chunk KV-pair in the indexing table is examined by the DRM. If the chunk fast tier reference count is 1 and the chunk slow tier reference count is 0, which means the data chunk is uniquely used by this file, then the original

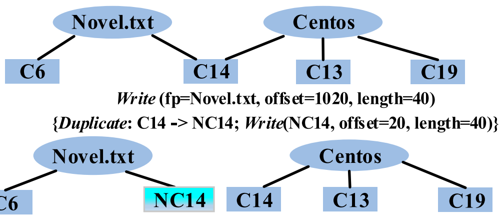

Novel.txt
Centos
C6
C14
C13
C19
*Write* (fp=Novel.txt, offset=1020, length=40)
{*Duplicate*: C14 -> NC14; *Write*(NC14, offset=20, length=40)}
Novel.txt
Centos
C6
NC14
C14
C13
C19

Fig. 3. An example of reloaded file shared chunk updating.

chunk in the fast tier can be directly updated. A new KV-pair for the updated chunk is created, and the old KV-pair of this chunk is deleted in the indexing table if this chunk is garbage collected in both tiers. If the chunk is shared by other reloaded files (fast tier reference count >1) or the chunk is used by some deduplicated files in the slow tier (slow tier reference count ≥1), then TDDFS applies Copy on Write (CoW) to create a new data chunk with the updated content. The *Chunk Address* in the metadata entry points to the new data chunk, and the MF is set to 1. At the same time, the fast tier reference count of the original chunk decreases by 1 in its KV-pair. Future updates to this new data chunk can be directly applied without changes to the metadata entry of the original chunk.

One example is shown in Figure 3. `Novel.txt` and `Centos` are reloaded files, and they share the same data chunk, C14. When a user updates the 40 bytes of `Novel.txt` from offset 1020, TDDFS calculates that C14 will be modified starting from offset 20 in that chunk. It reads C14 and updates its content in memory. Then, a new data chunk, NC14, is created, and the corresponding chunk metadata entry of the `Novel.txt` file recipe is updated to reference the new chunk. At the same time, the fast tier reference count of C14 decreases by 1 in its KV-pair.

For read requests, according to the read request offset in the file and the read size, the RFM first locates the metadata entries that correspond to the data chunks containing the requested data in the file recipe. Then, the requested data are read from these data chunks to the read buffer, and this buffer is returned to the application. The data chunks in the fast tier are not exclusively locked for read operations, and they can serve read requests from different files simultaneously.

Compared to the data of newly created files, the data of reloaded files are more fragmented. Therefore, the read performance of reloaded files is relatively lower than that of newly created files. Due to the CoW design for write requests, the write performance of reloaded files is also lower. However, according to the discussion of file access locality presented in Section 2.2, files tend to be repeatedly written by applications if they are written once. Only the first write to a shared chunk has the high performance overhead, and the overhead can be amortized over the following writes to that chunk. Also, we usually use a flash-based drive as the fast tier, and its random access performance is not much lower than its sequential access performance.

## 3.6 Major Data Operation Cases in TDDFS

In TDDFS, files fall into one of three categories: newly created file, deduplicated file, or reloaded file. TDDFS stores all the metadata in the fast tier, and most metadata operations are still processed the same way they would be processed in a regular file system. However, the processes used to respond to data requests are different because of the aforementioned four unique features of TDDFS. When handling a data request, TDDFS first checks the file category. Next, TDDFS responds to the request according to the different logic paths described below.

**Newly Created File Read and Write:** If the file is a newly created file, then the FSL directly reads or writes data in the fast tier. To maintain good performance on newly created files, there is no change in their data and metadata structures.

**Reloaded File Read and Write:** The details of the data read and write process for reloaded files are described in Section 3.5. The read and write performance of reloaded files is relatively lower than that of the newly created files due to the extra overhead of writing new chunks, file recipe updates, and KV-pair updates.

**Deduplicated File Read and Write:** The deduplicated files cannot be read from or written directly to the slow tier. When a file *open* happens on a deduplicated file, TDDFS starts a background thread to reload the file back to the fast tier. This reloading process is described in Section 3.4. If read and write requests happen during the file reloading, then TDDFS uses the following logic to respond to the requests. For a read request, according to the read request offset in the file and the read size, the RFM first locates the metadata entries that correspond to the data chunks containing the requested data in the file recipe. Then, TDDFS reads the requested data from the corresponding data chunks. Note that these data chunks can be located in either the fast tier (the data chunk has been reloaded to the fast tier and its RF is 1) or the slow tier (the data chunk will be reloaded shortly and its RF is 1). Then, TDDFS returns the requested data to the application.

For a write request, if the data chunks being written are already reloaded to the fast tier, then updates are performed directly on the unique data chunks, and CoW is performed on the shared chunks. The MFs of these chunks are set to 1 in the file recipe. If the data chunks being updated have not yet been reloaded to the fast tier (their RFs are 0), then TDDFS first reads the corresponding data chunks from the slow tier to the memory. Then, these data chunks are updated in the memory and written to the fast tier as new data chunks. The RFs and MFs of these chunks in the file recipe are set to 1. The DRM will skip a chunk whose RF has already been set to 1 and continue to reload the next data chunk. To maintain data consistency, concurrent reads or writes are not allowed during file reloading. Applications experience degraded read and write performance during the file restoration. After the file is completely reloaded to the fast tier, the read and write performance will be the same as that of reloaded files.

**Other Data Operations:** TDDFS also supports other data operations including file deletion and file truncation. The FSL responds to these operations on newly created files directly. File truncation (e.g., the file truncate interface in Linux), which is used to cut the file, is treated as a file write (update) and handled by the write operation logic. File deletions of reloaded files and deduplicated files are handled by metadata operations including metadata deletion and KV-pair updates done by the FMM and RFM. An un-referenced chunk whose fast tier or slow tier reference count is 0 in its KV-pair is cleaned in the corresponding tier by the garbage collection process.

## 4 IMPLEMENTATION

In this section, we present the implementation details of TDDFS. Rather than implementing TDDFS in kernel space, we developed the TDDFS prototype based on FUSE [38]. FUSE is a user space file system with a standard POSIX interface. The prototype consists of about 12K Lines of Code (LoC) and can be executed on top of Linux systems with FUSE installed. Although the FUSE-based file systems do not have the same performance as kernel-based file systems, it is used for simplicity and flexibility as a proof of concept. In the following subsections, we elaborate on the implementation details of the data management, indexing table, and file deduplication of TDDFS.

### 4.1 Data Management

In TDDFS, we use the underlying file system to store the data passed through by FUSE. TDDFS creates two top-level directories for the fast tier and the slow tier as shown in Figure 4. An underlying file system (e.g., Ext4) formatted SSD is mounted to a directory called Fast-tier that stores metadata, newly created files, and reloaded files. An underlying file system formatted HDD is mounted to a directory called Slow-tier, and it is used to store data chunks of deduplicated files. The data

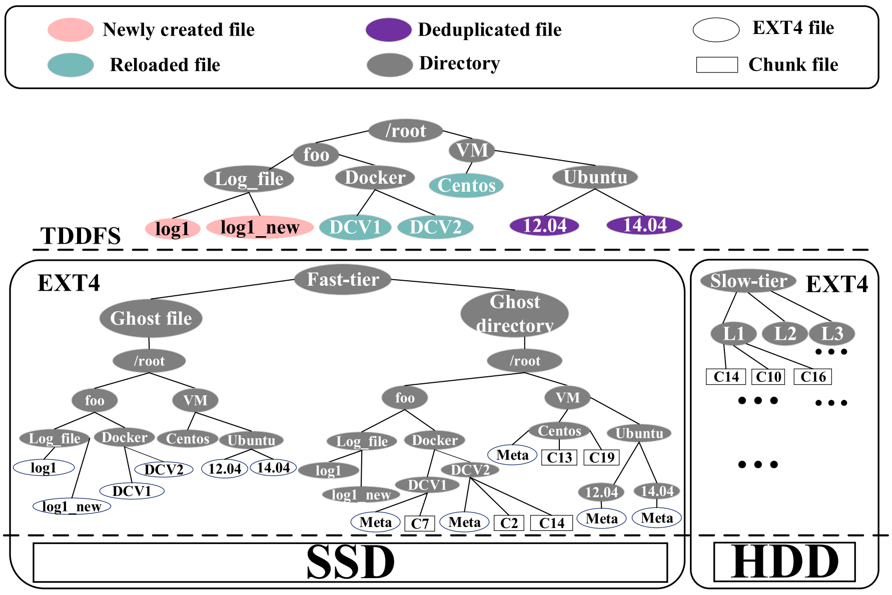


Fig. 4. Data layout of TDDFS prototype.

chunks generated by deduplication are stored as files under the Slow-tier directory with unique names (monotonically increasing unsigned integers). One directory under the Slow-tier directory acts as one chunk store, and 10,000 data chunks are stored in the same chunk store. As shown on the right side of Figure 4, directory L1 under the Slow-tier is one of the chunk stores, and data chunks are stored under this directory. In some data deduplication designs [32, 39], a large number of chunks (e.g., more than one thousand data chunks) are packed into one container to speed up the chunk write and restore performance. Container-based chunk storing can be easily adopted in TDDFS, and it can be combined with other restore performance improvement schemes [32, 33, 40, 41] to further increase the file deduplication and reloading performance.

Referring to the left side of Figure 4, the structure of the directory tree from an application's view through FUSE is mimicked twice under the Fast-tier directory in the underlying file system (e.g., Ext4). TDDFS creates one Fast-tier subdirectory called *Ghost file* and one Fast-tier subdirectory called *Ghost directory*. These two directories contain ghost files and ghost directories, respectively. The file and directory structure under *Ghost file* matches the application's view including directories, newly created files, deduplicated files, and reloaded files. *Ghost directory* is similar but has a directory entry for each file in the application's view. If the application view file is a newly created file, then its data are stored in its matching ghost file, and its ghost directory is empty. During the migration process, the data in the ghost file of a newly created file are read, deduplicated, migrated (if there are unique chunks), and deleted. The empty ghost file remains, and its file metadata information (e.g., *file_stat*), file recipe, and other information used for reloading are maintained as files under its ghost directory. When reloading the deduplicated files back to the fast tier, the unique data chunks are written to the fast tier under the ghost directory as files. We assume that the underlying file system guarantees the reliability, persistence, and consistency. TDDFS takes care of the failures that happen before the data are written to the underlying file system. When migrating the files from the fast tier to the slow tier, the least recently used files (including the files whose size is smaller than 16KB) are selected to be migrated to the slow tier.

To expand upon the above description, the same example that is used in Figure 2 is also presented here in Figure 4. The same directory structures are created under both *Ghost file* and *Ghost* *directory*. Since `log1` and `log1_new` are newly created files, their data are stored in their ghost files under *Ghost file* with the same file names, and their ghost directories are empty. `Ubuntu/12.04` and `Ubuntu/14.04` are deduplicated files, and their data are stored under Slow-tier as files of unique data chunks. `DCV1`, `DCV2`, and `Centos` are reloaded files, and the unique data chunks like C2, C7, and C14 are stored under the corresponding ghost directories of these files. The file recipe of a reloaded file or a deduplicated file is stored as a file named *Meta* under its ghost directory. A unique data chunk stored in the fast tier or the slow tier can be found using its path and name. In this prototype, we assume the fast tier's space is large enough, and we do not apply the file recipe migration design to simplify the implementation.

## 4.2 Indexing Table

We follow the design of BloomStore [35], which is optimized for flash storage, to implement the deduplication indexing table. In BloomStore, the whole key space is hashed into several instances. When inserting a KV-pair, the new KV-pair is hashed to one of the instances according to its key, and the KV-pair is logged to the KV-pair write buffer of this instance. This KV-pair write buffer is the same size as a flash page. When the buffer is full, it will be flushed to the fast tier as an immutable KV-block with a timestamp. Each KV-block has one Bloom Filter (BF) to speed up the KV-pair lookup process. Since the flash-based SSD cannot be overwritten before the whole block is cleaned [42], BloomStore does not change the KV-blocks directly. Updates and deletions are achieved by inserting new KV-pairs with the same key into the BloomStore. To delete a KV-pair, a new KV-pair with the same key and a value of *null* is inserted.

When searching for a key in BloomStore, the key is hashed to one corresponding instance. The BFs of this instance are checked first. The KV-blocks whose BFs have positive search indications are fetched to the cache and searched from the newest one (the KV-pair write buffer) to the oldest one according to the sequence of their timestamps. The first matching KV-pair with a valid value is returned. If the key cannot be found in all of the related KV-blocks or the first KV-pair has a *null* value, then a negative value is returned. The invalid KV-pairs (the KV-pairs that are updated or deleted) will be cleaned during the garbage collection process. More implementation details can be found in Reference [35]. In the current implementation, the KVS is locked when it is processing one request to ensure data consistency, which will have a performance penalty. TDDFS stores the KV-blocks in the fast tier as files in the underlying file system.

## 4.3 File Deduplication

When deduplicating newly created files and modified data chunks of reloaded files, TDDFS reads the file data into the chunking buffer in memory and uses Adler32 to calculate the 32-bit checksum of data in the CDC sliding window. If the checksum matches the pre-determined chunking mask value, then a new cutting point is applied. While we used SHA-1 as our deduplication fingerprint algorithm, it is no longer considered to be secure and practical hash collisions have been demonstrated [43]. Therefore, a practical implementation of TDDFS would have to change to a stronger hash function. When we search for a chunk ID in the indexing table, if the chunk ID already exists in the indexing table, then a metadata entry will be created and appended to the file recipe to index the data chunk. If the chunk ID is new, then the new data chunk is stored in the slow tier. Then, a new KV-pair will be created and stored in the KVS to index the new data chunk, and a metadata entry will also be appended to the file recipe.

## 5 EXPERIMENTAL EVALUATION

In this section, we present the results of the performance evaluation. First, we introduce our evaluation methodology, baselines that are compared with TDDFS, and the workloads we use in the evaluation. Then, the performance results and comparisons are discussed. Finally, the cost-effectiveness is analyzed.

## 5.1 Experimental Setup and Workloads

To evaluate TDDFS performance, we deployed our prototype on a Dell R420 server with a 2.2GHz Intel Xeon four-core processor and 12GB of memory. A Samsung 850 PRO 512GB with a SATA interface is the SSD used as the fast tier. The slow tier is a Western Digital WD RE4 WD5003ABYX 500GB 7200 RPM HDD with a 3.0Gb/s SATA connection to the motherboard. The operating system is Ubuntu 14.04 with a 3.13.0 kernel, and the underlying file system on both tiers is Ext4.

We implemented four different designs based on FUSE to compare with TDDFS. *NoDedup* manages the fast tier and the slow tier at the file system level, and there is no deduplication performed on either tier. Files are migrated between the two tiers according to the LRU policy. *AllDedup* is the design where the fast tier and the slow tier are managed at the file system level, and all files are deduplicated before these files are stored in either tier. When a file is read, the requested data chunks are located and returned. In this design, data chunks are migrated between the two tiers according to the LRU policy. *FastTierNoDedup* is a tradeoff design between *NoDedup* and *AllDedup*. Files in the fast tier are in their original states. When a file is selected to be migrated to the slow tier, it is deduplicated with the same implementation as TDDFS. If a file in the slow tier is accessed, then it will be restored to its original state and stored in the fast tier. *FastTierNoDedup* uses the design concept of the Dell SC Series hybrid storage solution with deduplication [44]. In *SlowTierOnly*, all files are stored in the slow tier to show the slow tier only performance in comparison to other tiering designs.

To measure and compare the performance of TDDFS, we use throughput, latency, and the amount of data migrated as the metrics. Throughput is the average number of requests completed in a time unit. It is calculated as the sum of the data sizes of all requests completed divided by the sum of total completion times of all file requests. Latency is the average request response time, measured at the file system level, from when the request is submitted to its completion. The amount of data migrated is the total amount of data that migrates from the fast tier to the slow tier or from the slow tier to the fast tier when the whole trace is replayed. During the evaluation, we vary the value of the fast tier ratio, the deduplication ratio, and the read ratio to measure and compare the performance of various designs. The fast tier ratio is the percentage of the whole dataset that can be stored in the fast tier. If the fast tier ratio is 100%, then it means all the data can be stored in the fast tier. If the fast tier ratio is 25%, then it means 25% of all the file data is stored in the fast tier and 75% of all the file data is stored in the slow tier. To show the capacity ratio between the fast tier and the slow tier (i.e., how many times larger the slow tier size is than the fast tier size), we increase the slow tier size from 0 times, to 1 times, to $N$ times the fast tier size. Thus, the fast tier ratio is represented from 100% to 50% to $\frac{100}{1+N}$%. For each trace, the total amount of data of all files is different. It is hard to compare the performance of different traces by using the same fast tier space size. Therefore, we define another threshold called the actual fast tier usage limit, which is calculated as the fast tier ratio times the total trace size. During the trace replay, when the total data size in the fast tier is larger than the actual fast tier usage limit, migration will be triggered. Note that the real-world capacity of the fast tier (SSD) may not be full when the amount of data in the fast tier reaches the experiment's actual fast tier usage limit. In our tests, when we refer to a fast tier ratio, we really use the calculated actual fast tier usage limit to replay the trace. The deduplication ratio is the original total data size divided by the total data size after the deduplication. The read ratio is the percentage of read requests among the all the data I/O requests (i.e., among the combined total number of read requests and write requests).

Table 2. Details of the Workload Traces

| | # files | # requests | Size(GB) | Read ratio |
|---|---|---|---|---|
| Lab_trace | 2,400 | 39,000 | 22 | 10% |
| Var_trace | 10,000 | 32,000 | 5 | 45% |
| Syn_trace | 5,000 | 100,000 | 100 | N/A |

The performance of TDDFS is closely related to the file access patterns, file size, read/write ratio, and the deduplication ratio. Most open-source file traces do not provide the file size and deduplication ratio information. File system benchmarks generate files with no meaningful content, so the deduplication ratio is still unknown. We use three different workloads to evaluate the performance of TDDFS, and the details of these three traces are shown in Table 2. Lab_trace is a file access trace collected from a file server in our lab for one week on top of *AllDedup*. We collected the size of each file and the file requests. The overall deduplication ratio of Lab_trace is 4.1. Var_trace is collected when we run the Filebench [45] workload *varmail* with the default Filebench configuration. Again, we collected the size of each file and the file requests. To comprehensively examine the relationship between performance and the deduplication ratio, read ratio, and fast tier ratio, we synthesize a set of file access traces, called Syn_trace, which has 100GB of files and 100,000 access requests. The file size and access patterns follow the Zipfian distribution (20% of the files have 80% of the total accesses and occupy 80% of the total size), which aligns with the observations in [23]. Different deduplication ratios, read ratios, and fast tier size ratios of Syn_trace are evaluated during the experiments. This means we generate a set of Syn_traces with the same file size and access distribution but different read request ratios. When the read ratio is selected, the number of read and write requests on each file is decided, and these requests are distributed to the duration of the whole trace according to the Zipfian distribution (80% of the requests happen after the first request to this file within 20% of the trace duration). The Syn_trace generated in this way with the Zipfian distribution has good locality, which can substantially benefit the tiered designs. Therefore, the evaluation of Syn_trace shows how TDDFS will perform when locality is good. In each test, the trace replays 5 times and we show the average value. Since the benchmark-generated Var_trace and the synthetic Syn_trace do not have meaningful time stamps, we replay the records of the three traces one after another (i.e., when the current request is finished and returned, the next request is issued).

A trace provides both file information and access information. To replay the traces, we make some changes to the prototype implementation such that the files are accessed according to the trace content. Before we replay the trace in each test, all the files are written to the prototype system in the same order as they appear in the trace with the collected size and no meaningful content. The earliest created files will be migrated to the slow tier if the fast tier space usage hits the actual fast tier usage limit. Since the trace does not have the fingerprint of the file content, we can only emulate the deduplication process by making small changes in the FMM, DRM, and RFM. During deduplication, the byte stream is still processed by the chunking and fingerprint generating steps. However, the real cutting points are randomly generated between a *minimal chunk size* (512B) and a *maximal chunk size* (16KB). Suppose the deduplication ratio is $R_{dr}$ in this test. Then $(\frac{100}{R_{dr}})\%$ of the chunks of this file are randomly selected as unique chunks, and the remaining $(100 - \frac{100}{R_{dr}})\%$ of the file's chunks are duplicate chunks. The fingerprint of a unique chunk is assigned as a unique increasing integer, $U_i$, which increments from 0. The fingerprints of duplicate chunks are randomly selected between 0 and the largest $U_i$ generated to show that they are duplicates. The data of a unique chunk are stored in the slow tier as a file. After we have the fingerprint of the chunk, the corresponding KV-pair is generated and inserted into the

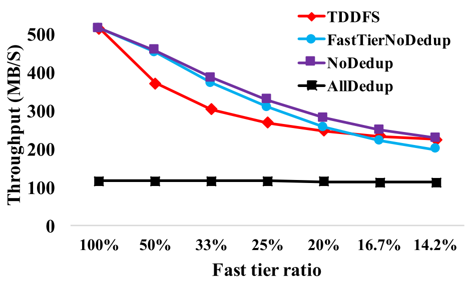
KVS, or the existing KV-pair is fetched from the KVS and updated. Then, the metadata entry with the corresponding content is written to the file recipe. In our experiment, we did not migrate the recipes to the slow tier if their size is very large. However, in a production scenario, especially when the accumulated files are very large, migrating the recipes to the slow tier is recommended.

The same emulation process is also used during the trace replay when reloading deduplicated files and updating reloaded files. For example, if the deduplication ratio is $R_{dr}$, when reloading a file from the slow tier to the fast tier, $(\frac{100}{R_{dr}})\%$ of the chunks of this file are read and written to the fast tier as unique data chunks. We assume that all reloaded files have the same deduplication ratio. Therefore, $(\frac{100}{R_{dr}})\%$ of the chunks are treated as new chunks that are not already in the fast tier. When writing the reloaded file, there is a $(100 - \frac{100}{R_{dr}})\%$ chance that the chunk will be CoW, since there will be $(100 - \frac{100}{R_{dr}})\%$ of the chunks of this file that are shared chunks. Although the emulated deduplication process is slightly different from deduplicating the files with real content, it has similar latency, throughput, and resource consumption to real deduplication. Therefore, we can use it to emulate the migration process with different deduplication ratios, and it can show the performance, cost-effectiveness improvement, and tradeoffs of TDDFS. In the real-world workloads, when more and more data chunks are accumulated in the slow tier, the restore performance penalty caused by the data chunk fragmentation will lower the overall throughput.

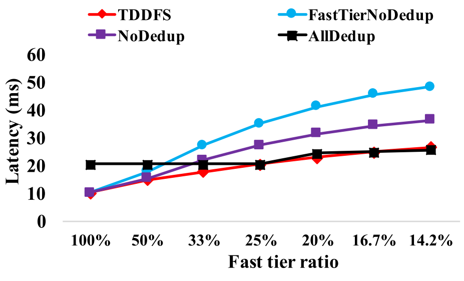
## 5.2 Throughput and Latency

When the deduplication ratio and read ratio are high, TDDFS can deliver better performance according to our tests. In Lab_trace, as shown in Figure 5(a), the throughput of TDDFS is slightly lower than that of *FastTierNoDedup* and *NoDedup*. As the fast tier ratio decreases from 100% to 20%, more files are migrated and stored in the slow tier. Thus, the throughput decreases and latency increases in all four tiering designs. In Lab_trace, most of the requests (90%) are write requests due to the heavy uploading and updating of files in the file server. High write ratio causes lower average throughput in TDDFS, which is caused by the CoW and recipe updates of the reloaded files. However, compared with the other three tiering designs when the fast tier ratio is below 33%, as shown in Figure 5(a) and (b), the rate of throughput decrease and rate of latency increase of TDDFS is better, since TDDFS can store more files in the fast tier than the other designs. By storing more files in the fast tier, the performance degradation of TDDFS is alleviated. If read requests dominate the file operations, then TDDFS can achieve better performance. In Syn_trace, we configure the read ratio as 80% and the deduplication ratio as 3. As shown in Figure 5(c) and (d), when the fast tier ratio is higher than 33%, the throughput of TDDFS is close to that of *FastTierNoDedup* and *NoDedup*. When the fast tier ratio is lower than 33%, the throughput of TDDFS is the best among the four designs. Again, this is because more reloaded files can be stored in the fast tier, and most of the file accesses are read requests. Note that in a typical tiered storage system, the fast tier ratio is low.

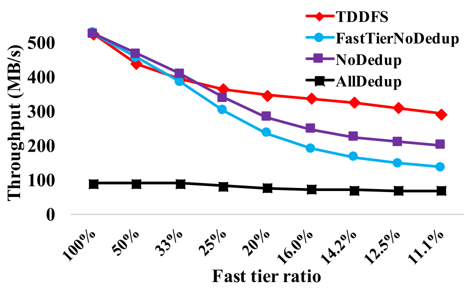
The performance of TDDFS is closely related to the deduplication ratio of the files. Since the fast tier ratio of 25% is the performance turning point in Lab_trace and Syn_trace, we fix the fast tier ratio to 25% in Var_trace to evaluate the performance with different deduplication ratios. As shown in Figure 5(e) and (f), the throughput and latency of *FastTierNoDedup* and *NoDedup* are nearly the same when the deduplication ratio varies from 1 to 4.6. Since *FastTierNoDedup* and *NoDedup* store the original files in the fast tier, the only difference is that *FastTierNoDedup* deduplicates files in the slow tier and restores files from the slow tier to the fast tier. Thus, deduplication ratio has little impact on the overall performance of *FastTierNoDedup* and *NoDedup*. Compared to the relatively flat throughput and latency of those two designs, *AllDedup* has a large change when the deduplication ratio is about 4. If the deduplication ratio is 4 or larger, then *AllDedup* can store



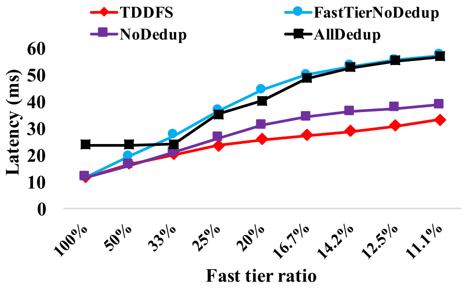
(a) Throughput of Lab_trace



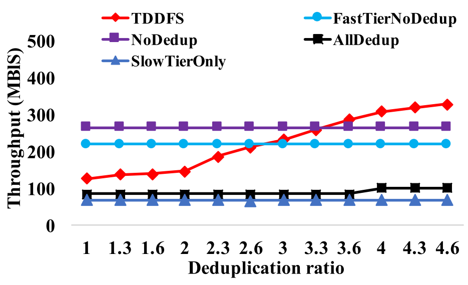
(b) Latency of Lab_trace



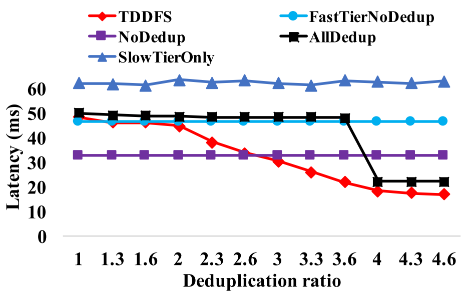
(c) Throughput of Syn_trace



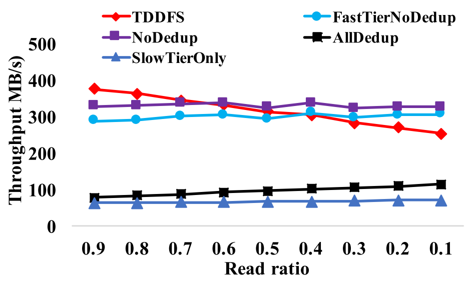
(d) Latency of Syn_trace



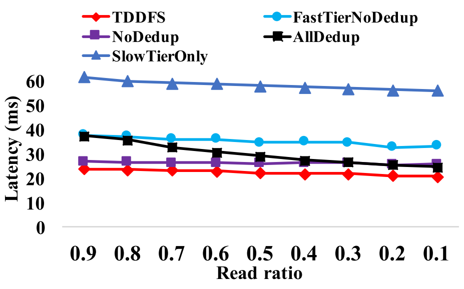
(e) Throughput of Var_trace



(f) Latency of Var_trace



(g) Throughput of Syn_trace



(h) Latency of Syn_trace

Fig. 5. Performance comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup*, and *SlowTierOnly*.

all the files in the fast tier, since the fast tier ratio is configured at 25% in this test. Therefore, its throughput increases and the latency has a large reduction. In contrast to these three designs, the throughput of TDDFS increases and the latency decreases gradually as the deduplication ratio increases from 1 to 4.6. As the deduplication ratio increases, more reloaded files can be stored in the fast tier with the help of non-redundant chunk reloading and unique chunk sharing as described in Section 3. Thus, the performance degradation caused by the data migration is alleviated. When the deduplication ratio is higher than 3.3, the throughput and latency of TDDFS are the best among the four designs. *SlowTierOnly* shows that if we only use the slow tier, its overall performance will be the worst. Its throughput can be more than 50% lower than that of TDDFS, and its latency can be 100% higher than that of TDDFS. In general, the performance of TDDFS is closely related to the deduplication ratio of files, and TDDFS performs better when the deduplication ratio is higher.

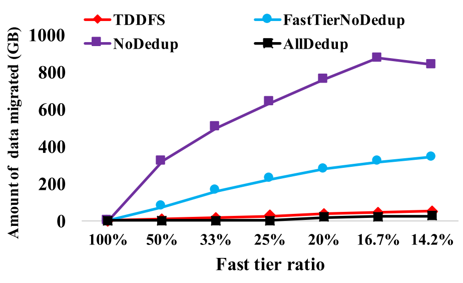
Since the read ratio is fixed in Lab_trace and Var_trace, we synthesize a set of Syn_traces with the same files, deduplication ratio, and access patterns but different read ratios. The access distribution we use is Zipfian, so most of the operations are on a small portion of the files. The results are shown in Figure 5(g) and (h). As the read ratio decreases from 90% to 10%, the performance of *FastTierNoDedup* and *NoDedup* does not change much, since files are accessed when they are in their regular format in the fast tier. The throughput of TDDFS decreases from 370MB/s to about 250MB/s as the read ratio decreases. Due to the low performance of the write operation on the reloaded files, a lower read ratio (higher write ratio) will cause a lower throughput in TDDFS. When the deduplication ratio is 3, the fast tier ratio is 25%, and the read ratio is higher than 70%, the throughput of TDDFS is higher than the others. According to Roselli et al. [22], the number of read requests in a typical file system is usually several times higher than the number of write requests (e.g., the read ratio is 80% or even higher). Thus, TDDFS can deliver good performance in a real production environment.

Also, as the read ratio decreases, the latency of TDDFS, *FastTierNoDedup*, and *NoDedup* does not change much, since most requests are performed in the fast tier. Due to the deduplicated nature of reloaded files, more hot files can be stored in the fast tier by TDDFS than in the others. Thus, its average latency is lower than that of the others. Since *AllDedup* performs deduplication to all the files, the data being written need to be deduplicated before the data are stored. Also, each read needs to assemble the requested chunks before returning the data to the application. Thus, its performance is lower than that of other tiered designs. In our implementation of *AllDedup*, which does not have any restore optimization, the throughput of read is lower than write, and the latency of read is much higher than write. Thus, as the read ratio decreases, the throughput of *AllDedup* increases slightly and its latency decreases. Clearly, without the help of the fast tier, *SlowTierOnly* shows the worst performance among all the designs.

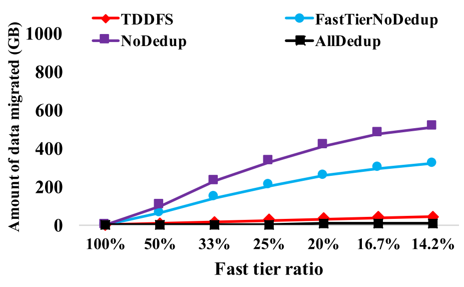
In general, according to our evaluation, the overall performance of TDDFS is better than that of *AllDedup*. In most cases, the latency of TDDFS is lower than the others, because more reloaded files are stored in the fast tier. With a higher deduplication ratio, higher read ratio, and lower fast tier ratio, TDDFS performs better, and in some cases, TDDFS performs the best, as shown in the evaluations.

## 5.3 Data Migration between the Fast Tier and the Slow Tier

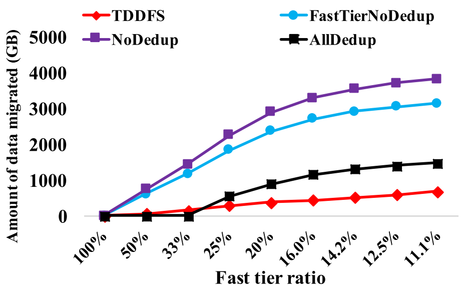
The data migration cost is measured as the total amount of data migrated from the fast tier to the slow tier or from the slow tier to the fast tier after executing the whole trace. The results are shown in Figure 6. In the four designs (excluding *SlowTierOnly*, since it does not have any data migration), *NoDedup* always has the largest data migration volume, since all the files are in their original states. Also, since fewer files can be stored in the fast tier without deduplication, file migration happens more frequently in *NoDedup*. The migration cost of *AllDedup* is the lowest,



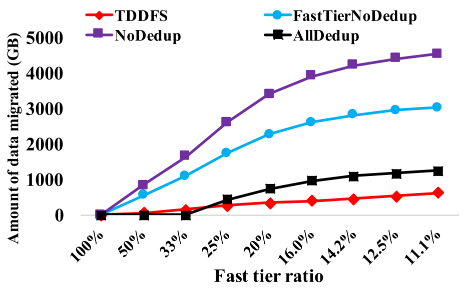
(a) Data migrated from the fast tier to the slow tier in Lab_trace



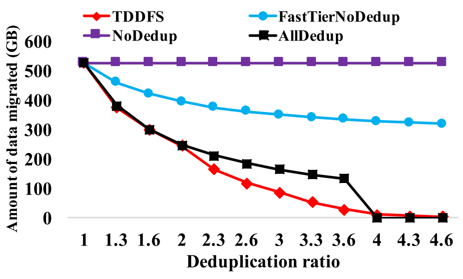
(b) Data migrated from the slow tier to the fast tier in Lab_trace



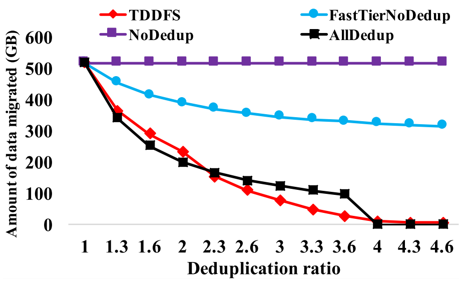
(c) Data migrated from the fast tier to the slow tier in Syn_trace



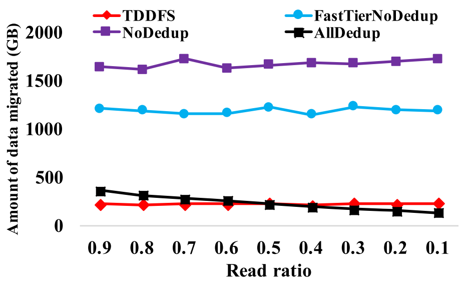
(d) Data migrated from the slow tier to the fast tier in Syn_trace



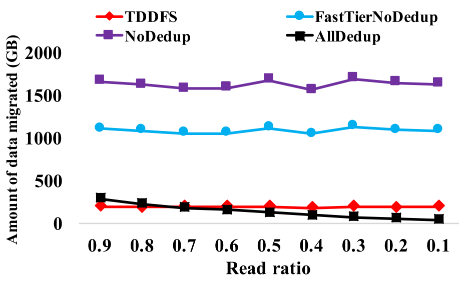
(e) Data migrated from the fast tier to the slow tier in Var_trace



(f) Data migrated from the slow tier to the fast tier in Var_trace



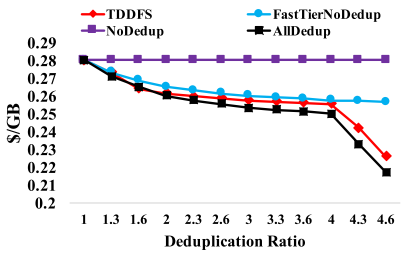
(g) Data migrated from the fast tier to the slow tier in Syn_trace



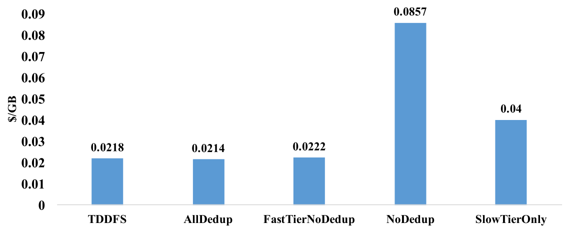
(h) Data migrated from the slow tier to the fast tier in Syn_trace

Fig. 6. Migration cost comparison of TDDFS, *NoDedup*, *AllDedup*, and *FastTierNoDedup*.

since all the files are deduplicated, and only the unique chunks are migrated between the two tiers. Also, *AllDedup* can hold more files in the fast tier than the others, and migrations happen less frequently. *FastTierNoDedup* performs better than *NoDedup*, but its total migrated data size is still much higher than that of *AllDedup* and TDDFS. *FastTierNoDedup* reduces the amount of data



(a) $/GB comparison of TDDFS, *NoDedup*, *AllDedup*, and *FastTierNoDedup* with different deduplication ratios



(b) $/GB comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup* and *SlowTierOnly* when the size of slow tier is 20 times larger than fast tier

Fig. 7. Migration cost comparison of TDDFS, *NoDedup*, *AllDedup*, *FastTierNoDedup*, and *SlowTierOnly*.

that is written to and read from the slow tier via slow tier deduplication. However, the amount of data being migrated from or written to the fast tier is the same as that of *NoDedup*. Thus, the migration cost of *FastTierNoDedup* is much higher than that of TDDFS and *AllDedup*.

As shown in Figure 6, the total amount of data migrated between the fast tier and the slow tier of TDDFS is close to that of *AllDedup*. Although TDDFS needs to read all the data of a newly created file when it is migrated to the slow tier, TDDFS avoids the re-deduplication and migration if the data chunks of a reloaded file are not changed during its lifecycle in the fast tier. Thus, the total migration data volume is close to that of *AllDedup*. Inevitably, as the fast tier ratio decreases, the fast tier will store fewer files, and data migration will occur more frequently. The rate at which the total amount of migrated data of TDDFS increases as the fast tier ratio decreases is the lowest among the four designs, as shown in Figure 6(a)–(d). With higher deduplication ratios, the migration cost of TDDFS becomes lower, as shown in Figure 6(e) and (f). The read ratio does not show a major influence on the TDDFS migration amount, as seen in Figure 6(g) and (h). On average, our evaluations show the total amount of data being migrated between the fast tier and the slow tier of TDDFS is about 10% of *NoDedup*, 20% of *FastTierNoDedup*, and close to *AllDedup* in most cases. In general, the migration cost of TDDFS is much lower than that of *FastTierNoDedup* and *NoDedup* and is close to that of *AllDedup*. Lower migration cost can reduce the storage bandwidth used by migration and alleviate the performance impact on the primary workloads during the migration. It can also reduce wear on flash-based SSDs.

## 5.4 Cost-Effectiveness Analysis

We use the average $/GB price to compare the cost-effectiveness of the four designs. We use the final utilized space in the fast tier and the slow tier after replaying the trace to calculate the average $/GB. Here, we use the $/GB of an Intel 750 Series 400GB SSD (SSDPEDMW400G4R5) and a Seagate 6TB enterprise HDD (ST6000NM0024) as the base to analyze the cost-effectiveness. For these two devices, the $/GB of SSD is about 1 [46] and $/GB of HDD is about 0.04 [47].

We use Var_trace to analyze the $/GB variation as the deduplication ratio increases, and we assume that the fast tier size is 25% of the total data size. As shown in Figure 7(a), the $/GB of *NoDedup* remains the same as the deduplication ratio increases because no deduplication is used. The $/GB of *AllDedup*, TDDFS, and *FastTierNoDedup* decreases as the deduplication ratio increases, since data deduplication reduces the amount of data being stored in the fast tier and the slow tier. When the deduplication ratio is larger than 4, the $/GB of *FastTierNoDedup* keeps a similar saving ratio while the $/GB of *AllDedup* and TDDFS has a significant decrease. When the deduplication ratio is high enough, *AllDedup* and TDDFS are able to save space in the fast tier, which causes a significant reduction in $/GB. In this calculation, when the deduplication ratio is higher than

4.3, TDDFS no longer needs any slow tier space, and all files are stored in the fast tier. As the deduplication ratio continuously increases, TDDFS can save more space on the fast tier and the rate of $/GB decrease is nearly the same as that of *AllDedup*. In general, TDDFS can achieve better cost-effectiveness than that of *NoDedup* and *FastTierNoDedup*. Its cost-effectiveness is close to that of *AllDedup*. With a high deduplication ratio, TDDFS can achieve nearly the same cost-effectiveness as that of *AllDedup*.

In a tiered storage system, the size of the fast tier is usually much smaller than that of the slow tier. We let the size of the slow tier be 20 times larger than that of the fast tier and compare the $/GB of each design when the deduplication ratio is fixed at 4. As shown in Figure 7(b), the $/GB of *NoDedup* is about 4 times that of TDDFS due to no deduplication. The $/GB of *SlowTierOnly* is also much higher than that of TDDFS, *AllDedup*, and *FastTierNoDedup*. It shows that the slow tier only system without deduplication cannot provide either good performance or good cost-effectiveness. The $/GB of TDDFS, *AllDedup*, and *FastTierNoDedup* is about 0.02 and close to each other. This means the cost-effectiveness of TDDFS, *AllDedup*, and *FastTierNoDedup* is very close. Importantly, according to the performance evaluation in Section 5.2, TDDFS usually has better performance than that of *AllDedup* and *FastTierNoDedup*. Therefore, TDDFS simultaneously achieves good tradeoffs in terms of performance and cost-effectiveness.

## 6 RELATED WORK

Some previous studies tried to address the challenges of primary data deduplication in different ways [48–51]. Most of them sacrifice the deduplication ratio by applying fixed-size chunking and make tradeoffs to improve the throughput. iDedup [48] sacrifices deduplication ratio to ensure higher throughput and lower latency. It may choose not to deduplicate non-sequential duplicate blocks to maintain locality so the deduplication and restore performance can be improved. Dmd-edup [49] is implemented in a device mapper that can be used by most existing file systems and applications. Although it did place deduplication structures like the fingerprint index in a fast device and the deduplicated data blocks on a slower device, it does not have the concept of tiered storage, and data cannot be migrated according to the data activity.

Instead of using the SSD and HDD as a tiered design, some studies use the SSD as a cache for primary storage and apply deduplication to the SSD to improve the system performance, SSD endurance, and space utilization. Nitro [52] combines deduplication and compression to improve the SSD cache capacity. By carefully designing the indexing structure and using Write-Evict Units that are the same size as the flash erase block, Nitro makes a good tradeoff between the SSD endurance and the performance of a primary storage system. CacheDedup [27] also introduces deduplication to the flash-based cache to improve the space utilization and flash lifespan. Uniquely, CacheDedup uses a separate Data Cache (for data blocks in the storage) and Metadata Cache (for fingerprints and other metadata information about the data blocks in the storage). This is done to reduce the space used by metadata and to better collect the historical metadata information for duplication reorganization. Moreover, CacheDedup proposes the D-LRU and D-ARC cache replacement algorithms, which are duplication-aware, to further improve the cache performance and endurance. Similarly, TDDFS maintains the deduplicated state of reloaded files to achieve better capacity utilization of the fast tier. However, TDDFS keeps some of the new and hot files in their original state without deduplication to gain the best performance for these files.

Other primary deduplication works [4, 53–55] are implemented at the file system level. There are three file systems with a deduplication function: SDFS [54], Lessfs [53], and ZFS [4]. SDFS and Lessfs are implemented in user space and based on FUSE. SDFS [54] combines fixed-size and variable chunking to eliminate chunk duplicates among files, especially for virtual machines, but SDFS cannot manage two storage tiers. Lessfs uses fixed-size chunking, and it is implemented in C. Thus, its overhead might be lower than that of SDFS, and it can deliver higher throughput. However, its deduplication ratio is worse than that of TDDFS. Different from SDFS and Lessfs, ZFS does not support deduplication in its basic functions. When users enable deduplication in one directory, all data written in the directory will be deduplicated to gain better space utilization. However, its performance is limited by the memory space for deduplication metadata [49]. Although these works apply deduplication at the file system level to improve space utilization, they do not distinguish active and less-active files to selectively deduplicate them, and they do not operate in two-tier storage systems.

## 7 CONCLUSIONS AND FUTURE WORK

It is challenging to develop high-performance but low-cost file systems on primary storage. Previously, researchers focused on block-based tiered storage designs and primary deduplication to achieve this goal. However, problems still exist in these solutions such as migration efficiency, deduplication penalty, and isolation between the block-level management and the upper-layer applications.

To conquer these challenges, we design TDDFS to manage a two-tier storage system and achieve efficient file-level migration. First, we reduce the storage I/O bandwidth required between tiers during migration by eliminating duplicate chunks and enabling data chunk sharing among reloaded files. Second, to solve deduplication latency and throughput issues, we selectively deduplicate less-active files. In this way, we can minimize the performance impact on the whole system. Finally, combining deduplication with data chunk sharing in both tiers further improves the space utilization and reduces costs. In our evaluation, TDDFS achieves high performance and low $/GB at the same time. In the future, we plan to further improve the deduplication ratio and minimize the performance impact by designing a more intelligent file selection policy, applying deduplicated file pre-reloading, implementing a high-performance indexing table, and using adaptive chunking algorithms.

## ACKNOWLEDGMENTS

We thank all the members in the CRIS group who provided useful comments to improve our design. We thank Shiyong Liu for the specific help with adjusting figures and formats. We also thank Gaurav Makkar and Al Andux for the comments and suggestions.

## REFERENCES

[1] David Reinsel, John Gantz, and John Rydning. 2018. The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[2] Y. Liu, M. Chen, and S. Mao. 2014. Big data: A survey. *Mobile Netw. Appl.* 19, 2 (2014), 171–209.

[3] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)* 2.

[4] J. Bonwick. ZFS deduplication. 2009. https://blogs.oracle.com/bonwick/entry/zfs_dedup.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–10.

[6] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 307–320.

[7] Andrew W. Leung, Shankar Pasupathy, Garth R. Goodson, and Ethan L. Miller. 2008. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*. 213–226.

[8] Kitchin R. 2014. *The Data Revolution: Big Data, Open Data, Data Infrastructures and Their Consequences*. Sage.

[9] Hui Wang and Peter Varman. 2014. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 229–242.

[10] Tivoli storage productivity center v5.2 documentation. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSNE44_5.2.0 /com.ibm.tpc_V52.doc/tpc_kc_homepage.html.
[11] Dell storage sc8000. Retrieved from https://www.dell.com/en-us/work/shop/cty/pdp/spd/dell-compellent-sc8000.
[12] Dell EMC glossary. Retrieved from http://www.emc.com/corporate/glossary/fully-automated-storage-tiering.htm.
[13] Automated storage tiering and the NetApp virtual storage tier. Retrieved from https://community.netapp.com/t5/Tech-OnTap-Articles/Automated-Storage-Tiering-and-the-NetApp-Virtual-Storage-Tier/ta-p/84825.
[14] Samuel Burk Siewert, Nicholas Martin Nielsen, Phillip Clark, and Lars E. Boehnke. 2010. Systems and methods for block-level management of tiered storage. (August 5 2010). US Patent App. 12/364,271.
[15] Anant Baderdinni. 2013. Relative heat index based hot data determination for block based storage tiering. (July 2 2013). US Patent 8,478,939.
[16] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. 2016. Using hints to improve inline block-layer deduplication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 315–322.
[17] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. 2015. Request-oriented durable write caching for application performance. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 193–206.
[18] Overview of GPFS. Retrieved from https://www.ibm.com/support/knowledgecenter/en/STXKQY_4.2.0/com.ibm.spectrum.scale.v4r2.ins.do c/bl1ins_intro.htm.
[19] ReFS. Retrieved from https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overview.
[20] Oracle hierarchical storage manager. Retrieved from https://www.oracle.com/storage/tape-storage/hierarchical-storage-manager/.
[21] Drew Roselli and Thomas E. Anderson. 1998. *Characteristics of File System Workloads*. University of California, Berkeley, Computer Science Division.
[22] Drew S. Roselli, Jacob R. Lorch, Thomas E. Anderson, et al. 2000. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX ATC'00)*. 41–54.
[23] Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Trans. Stor.* 7, 4 (2012), 14.
[24] Yinjin Fu, Hong Jian, Nong Xiao, Lei Tian, and Fang Liu. 2011. AA-Dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'11)*. IEEE, 112–120.
[25] Danny Harnik, Ety Khaitzin, and Dmitry Sotnikov. 2016. Estimating unseen deduplication-from theory to practice. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 277–290.
[26] Jaehong Min, Daeyoung Yoon, and Youjip Won. 2011. Efficient deduplication techniques for modern backup operation. *IEEE Trans. Comput.* 60, 6 (2011), 824–840.
[27] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. 2016. CacheDedup: In-line deduplication for flash caching. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 301–314.
[28] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. 101–114.
[29] João Paulo and José Pereira. 2014. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 11.
[30] Jyoti Malhotra and Jagdish Bakal. 2015. A survey and comparative study of data deduplication techniques. In *Proceedings of the International Conference on Pervasive Computing (ICPC'15)*. IEEE, 1–5.
[31] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710.
[32] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 183–197.
[33] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 2014. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 181–192.
[34] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. 2015. Design tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 331–344.
[35] Guanlin Lu, Young Jin Nam, and David H. C. Du. 2012. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. IEEE, 1–11.

[36] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*. 215–230.
[37] RocksDB. Retrieved from https://github.com/facebook/rocksdb.
[38] FUSE. Retrieved from http://fuse.sourceforge.net/.
[39] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, Vol. 8. 1–14.
[40] Zhichao Cao, Hao Wen, Fenggang Wu, and David H. C. Du. 2018. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 309–324.
[41] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 2019. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.
[42] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'08)*, Vol. 57.
[43] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. In *Proceedings of the Annual International Cryptology Conference*. Springer, 570–596.
[44] Dell SC series. Retrieved from http://en.community.dell.com/techcenter/extras/m/white_papers/20442763/download.
[45] Filebench. Retrieved from https://github.com/filebench/filebench/wiki.
[46] Intel. 750 series 400GB SSD. Retrieved from https://www.amazon.com/intel-single-400gb-solid-ssdpe2mw400g4x1/dp/b011i61l70.
[47] Seagate. 6T enterprise HDD (ST6000NM0024). Retrieved from https://www.amazon.com/seagate-barracuda-3-5-inch-internal-st6000dm004/dp/b01loojbh8.
[48] Kiran Srinivasan, Timothy Bisson, Garth R. Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. 299–312.
[49] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. 2014. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium*.
[50] Yoshihiro Tsuchiya and Takashi Watanabe. 2011. DBLK: Deduplication for primary block storage. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE, 1–5.
[51] Aaron Brown and Kristopher Kosmatka. 2010. Block-level inline data deduplication in ext3. University of Wisconsin-Madison Department of Computer Sciences.
[52] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. 501–512.
[53] Lessfs. 2012. Retrieved from https://fedoraproject.org/wiki/Features/LessFS.
[54] Opendedup–SDFS. 2012. Retrieved from http://www.opendedup.org.
[55] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. 2012. Primary data deduplication—large scale study and system design. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 285–296.

Received December 2017; revised September 2018; accepted November 2018