PAPER

Offline Selective Data Deduplication for Primary Storage Systems

Sejin PARK†a), *Nonmember and* Chanik PARK†b), *Member*

SUMMARY Data deduplication is a technology that eliminates redundant data to save storage space. Most previous studies on data deduplication target backup storage, where the deduplication ratio and throughput are important. However, data deduplication on primary storage has recently been receiving attention; in this case, I/O latency should be considered equally with the deduplication ratio. Unfortunately, data deduplication causes high sequential-read-latency problems. When a file is created, the file system allocates physically contiguous blocks to support low sequential-read latency. However, the data deduplication process rearranges the block mapping information to eliminate duplicate blocks. Because of this rearrangement, the physical sequentiality of blocks in a file is broken. This makes a sequential-read request slower because it operates like a random-read operation. In this paper, we propose a selective data deduplication scheme for primary storage systems. A selective scheme can achieve a high deduplication ratio and a low I/O latency by applying different data-chunking methods to the files, according to their file access characteristics. In the proposed system, file accesses are characterized by recent access time and the access frequency of each file. No chunking is applied to update-intensive files since they are meaningless in terms of data deduplication. For sequential-read-intensive files, we apply big chunking to preserve their sequentiality on the media. For random-read-intensive files, small chunking is used to increase the deduplication ratio. Experimental evaluation showed that the proposed method achieves a maximum of 86% of an ideal deduplication ratio and 97% of the sequential-read performance of a native file system.

key words: *data deduplication, selective deduplication, rank based deduplication*

1. Introduction

Data deduplication reduces required disk space by eliminating redundant data in storage. The redundant data is identified by a hash value. Figure 1 shows how a data deduplication scheme works. The deduplication requires additional I/O operations to calculate a hash value for data and compare its value with that of existing data. Most previous deduplication works have considered backup storage systems [1]–[9]. However, data capacity is still important for primary storage. The International Data Corporation (IDC) forecasts that total disk capacity will grow at an annual growth rate of 43.6%, because the size of individual files, such as virtual disk files, image data, video data, and rich media files, are much higher [1]. Therefore, primary storage should be considered as a data deduplication target.

When we apply data deduplication to a primary storage system, it is important to maintain a low I/O latency while achieving a high data deduplication ratio. Typically, data deduplication methods for primary storage systems are classified into two approaches: offline and inline. The offline approach executes deduplication during idle time, and the inline approach conducts deduplication at every write operation.

The advantage of inline data deduplication is that it does not require additional space for deduplication. However, this method requires additional I/O latency. To mitigate this overhead, many prior studies have suggested alternative smaller hash index tables [4], [7], [10]. Owing to the smaller hash index table size, the hash table lookup overhead can be reduced, at the cost of a deteriorated deduplication ratio. Thus, we see the trade-off between deduplication ratio and I/O latency. In addition, if a file is update-intensive, its deduplication is worthless. However, even recent research on inline primary data deduplication techniques [4], [7], [10] does not consider update-intensive files.

In contrast, offline deduplication can avoid additional I/O latency since it only runs deduplication when the system is idle. Moreover, it can easily consider each file's access characteristics during data deduplication. However, existing offline primary data deduplication techniques [11]–[13] do not consider I/O latency. That is, when data is deduplicated, the physical location of each block can be randomized, even

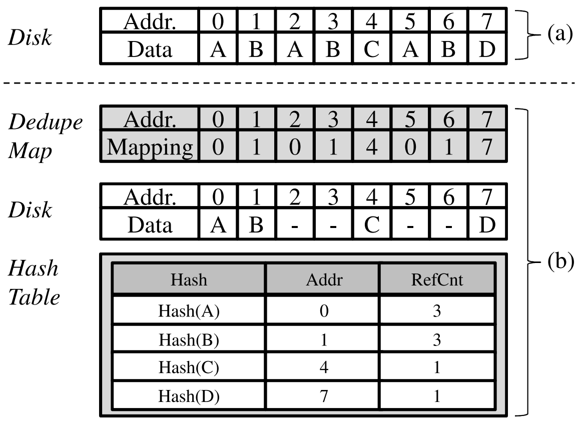


Fig. 1 Data deduplication. (a) shows a disk layout without deduplication, and (b) shows a disk layout with deduplication. In (a), 8 data blocks are used, but in (b) only 4 blocks are used. Four blocks (1, 4, 5, 7) are freed by the deduplication map. The hash table maintains a hash value and its corresponding address with a reference counter. The hash value is used to identify the data.

Manuscript received January 30, 2015.
Manuscript revised August 8, 2015.
Manuscript publicized October 26, 2015.

†The authors are with POSTECH, Hyoja-dong, Pohang, 790–784 Republic of Korea.
a) E-mail: baksejin@postech.ac.kr
b) E-mail: cipark@postech.ac.kr

DOI: 10.1587/transinf.2015EDP7034

if the original data was sequential.

In this paper, we propose a selective offline data deduplication method that efficiently controls the trade-off between the data deduplication ratio and the I/O latency. Specifically, file access characteristics are analyzed, considering both sequential access and update patterns. Each file is assigned a rank according to its access characteristics. The rank of a file determines the proper chunking method, to balance a high deduplication ratio with a low I/O latency.

The contributions of this paper are as follows.

1. A rank-based selective deduplication method effectively classifies deduplication target data according to its access pattern and time stamp.
2. The method effectively controls the tradeoff between the data deduplication ratio and I/O latency. It supports a high deduplication ratio for randomly accessed files and a low I/O latency for sequentially accessed files.
3. The proposed method can be easily applied to existing file systems.

The remainder of the paper proceeds as follows: Sect. 2 lists related work on deduplication. Section 3 describes the problem and Sect. 4 explains the detailed design of the proposed method. Section 5 evaluates the proposed method based on the prototype, and Sect. 6 concludes.

2. Related Work

Data deduplication can be classified into two methods: inline and offline.

Inline data deduplication: For inline data deduplication, many studies focused on hash index table compaction because of the large size of the hash index table. Lillibridge et al. proposed a sparse indexing method to reduce in-memory overhead [4]. However, it impacted the deduplication ratio because of the sparse set of the index table. Xia et al. proposed a similarity-locality based deduplication scheme to reduce RAM overhead and support high throughput [7]. Srinivasan et al. proposed the iDedup algorithm [10]. They claimed that data deduplication causes additional sequential-read-latency problems since data are stored randomly on the disk. That is, the data blocks of a file are fragmented by the data deduplication processing. To solve this problem, they do not deduplicate short data-block sequences. They only deduplicate long data-block sequences to reduce fragmentation. Thus, the method can reduce sequential-read latency. However, though the proposed deduplication algorithm can reduce I/O latency, it also reduces the data deduplication ratio. This is because they applied the algorithm to all data blocks without considering the file access characteristics. Moreover, it requires an additional non-volatile memory buffer for the temporary storage space of the deduplication operation, which is unusual for primary storage systems.

El-Shimi et al. found that the chunk size distribution is skewed at the minimum chunk size and maximum chunk sizes [14]. That is, the chunk boundary is forced by the maximum chunk size and it causes a low deduplication ratio. They proposed a regression chunking method for primary data deduplication to obtain a uniform chunk-size distribution.

Offline data deduplication: EMC Celerra [12] and NetApp ASIS [11] are offline data deduplication systems. EMC Celerra supports file-level data deduplication and compression for older files. However, it does not support a subfile-level method, and the compression technique requires additional decompression overhead when it is re-accessed. On the other hand, NetApp ASIS supports a subfile-level method but has a complex configuration and aims toward the enterprise environment. In addition, existing offline data deduplication techniques do not consider the I/O latency that is important for primary storage systems.

3. Problem Statements

There are several factors to be considered in designing offline data deduplication for primary storage systems.

Sequential read latency: Primary storage is sensitive to I/O latency. Unfortunately, data deduplication has a detrimental effect on sequential-read latency. In most cases, a file consists of physically contiguous blocks in the file system to support a low sequential-read latency. When a file is deduplicated, however, the file chunks are no longer physically contiguous. The location of each chunk is determined by the location of the already existing chunks. Figure 2 shows this phenomenon. Figure 2-(a) shows the original state of a storage system that has not been deduplicated. In this storage, there are six files, File 1 to File 6, and each file consists of blocks with unique values. Files 3 and 4 are sequential-read oriented.

When we conduct deduplication with big chunks (3 blocks), we will lose considerable deduplication potential, but we will achieve a good read latency, as depicted in Fig. 2-(b). In contrast, if we conduct deduplication with small chunks (1 block), we will achieve a much higher deduplication ratio, but we will lose sequential-read latency, as depicted in Fig. 2-(c). In this case, to read File 4, we must retrieve block D from File 2, blocks F and A from File 1, block B from File 3, and block C from File 1 again. Though the read request was sequential, this was obviously handled as a random read request. Figure 2-(d) shows the goal of this paper: selective deduplication. It deduplicates sequential-read-oriented files (Files 3 and 4) with a big chunk size to achieve good sequential-read performance; the other files are deduplicated using a small chunk size to achieve a high deduplication ratio.

Trade-off between deduplication ratio and I/O latency: Achieving a high deduplication ratio is essential for data deduplication. However, the problem here is that there is a trade-off between the deduplication ratio and the I/O latency. Figure 3 shows the trade-off for various chunk sizes.

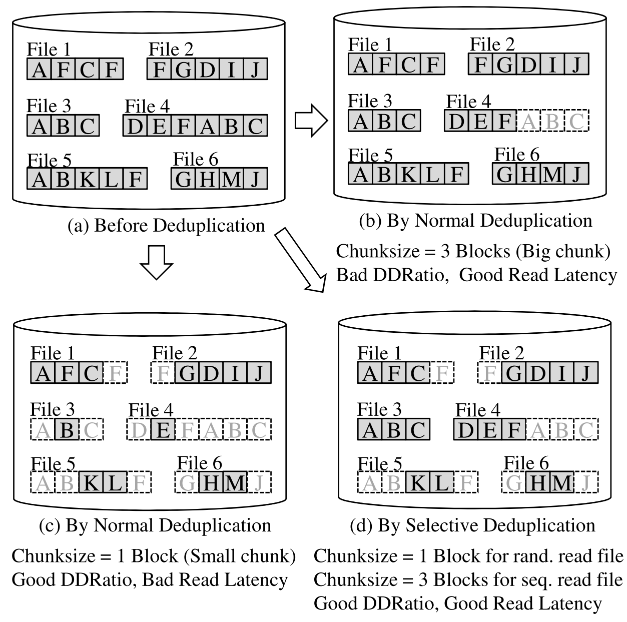

File 1 A F C F
File 2 F G D I J
File 3 A B C
File 4 D E F A B C
File 5 A B K L F
File 6 G H M J
(a) Before Deduplication
(b) By Normal Deduplication
Chunksize = 3 Blocks (Big chunk)
Bad DDRatio, Good Read Latency
(c) By Normal Deduplication
Chunksize = 1 Block (Small chunk)
Good DDRatio, Bad Read Latency
(d) By Selective Deduplication
Chunksize = 1 Block for rand. read file
Chunksize = 3 Blocks for seq. read file
Good DDRatio, Good Read Latency

Fig. 2 Illustration of selective deduplication. Files 3 and 4 are sequential-read-oriented files and the others are random-read oriented. (a) depicts the not-yet-deduplicated original file state. (b) and (c) depict deduplication results using big and small chunking methods, respectively. (d) depicts the results of selective deduplication. Since Files 3 and 4 are sequential-read oriented, these files are deduplicated with big chunks to achieve low read latency, and the others are deduplicated with small chunks to achieve a high deduplication ratio.

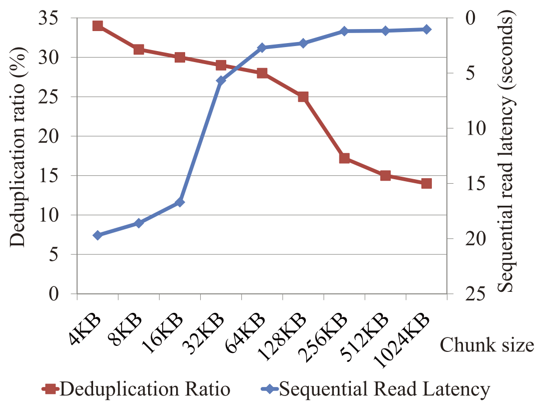


Fig. 3 Trade-off between data deduplication ratio and sequential-read latency. A bigger chunk size results in a worse data deduplication ratio, but a better sequential-read latency.

For the sequential-read latency experiment, we made a 100 MB file that was randomly stored in fixed-size chunks on the disk, but the data within each chunk was stored sequentially. For example, if the chunk size is 1,024 KB, then the 100 MB file consists of 100 chunks that are randomly located on the disk, but each chunk consists of sequential blocks. The sequential-read latency of a 1,024 KB-sized chunk is seven times faster than that of an 8 KB-sized chunk. In this experiment, we use Ubuntu 12.04 LTS with Ext4 file system [15]on Intel Xeon E5620 2.4 GHz CPU and 4 GB of RAM. Note that the Linux default read-ahead mechanism is enabled.

It appears that the I/O latency problem can be solved by using a bigger chunk size; however, it causes another problem: it reduces the deduplication ratio. For the deduplication-ratio experiment, we used the developer's workload described in Sect. 5.2. The deduplication ratio difference between a 1,024 KB-sized chunk and an 4 KB-sized chunk was more than double. In other words, a bigger chunk size resulted in a lower data deduplication ratio. Thus, we need a solution to achieve a low I/O latency while simultaneously achieving a high data deduplication ratio.

Offline write operation support: Special care should be taken for the write operation in offline data deduplication. For example, let us assume that block #N is already deduplicated and a write request is issued to it. In this situation, the deduplicated block #N cannot be updated immediately because other files may have shared links to it. Moreover, if block #N is updated, then the hash table should be updated to maintain consistency. This situation should be considered for offline data deduplication.

4. Design

Gibson and Miller [16] analyzed long-term file-activity patterns in a UNIX workstation environment at their university. They showed that more than 50% of files were not accessed at all for about 300 days, only 1.2% of files were accessed daily, and most modifications occurred on the day the file was created. The lifetime of more than 90% of deleted files was a single day [17]. Previous research on file access analysis has shown similar results [16], [18], [19]. That is, most files are not accessed for a long time. Only a few files are accessed for a short period.

According to observations on file accesses, only a few files are frequently accessed; the others are rarely used. We call them hot and cold files, respectively. For cold files, we concentrate on maximizing the deduplication ratio rather than I/O latency, since they are rarely accessed. In contrast, for hot files, we concentrate on minimizing the I/O latency for the sequential read, since their access frequency is higher.

In order to solve the sequential-read latency issue, we monitor each file to gather access patterns and conduct deduplication using big chunking for hot and sequential-read-intensive files. To solve the trade-off between deduplication ratio and I/O latency, we conduct deduplication using small chunking for random-read-intensive files or cold files.

To support this scheme efficiently, we assign a rank to each file. The rank represents each file's access patterns and its activeness. The deduplication policy is then selected by the rank of each file. For offline write operation support, we use a lazy update scheme that naturally solves the write latency.

4.1 Architecture

Figure 4 depicts the overall architecture of the proposed sys-

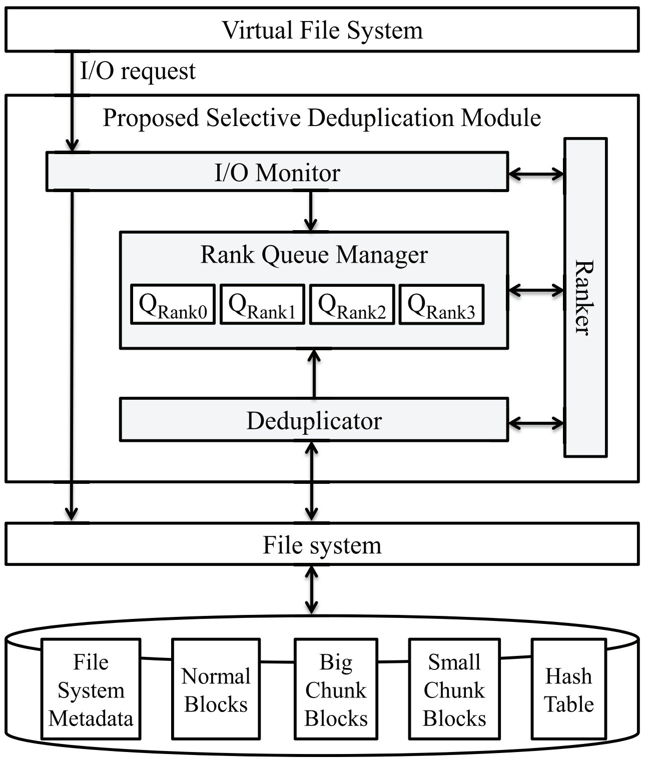


Fig. 4 Architecture of the proposed system. The proposed system monitors every I/O request from the VFS layer, and the Rank Queue Manager updates the rank queue according to the current I/O operation. When the system is idle, the Ranker and the Deduplicator run. The Ranker ranks all files in Q_{Rank0} and the Deduplicator conducts deduplication for newly ranked files.

tem. It consists of an I/O Monitor, Rank Queue Manager, Ranker, and Deduplicator. The I/O monitor captures file operations in the virtual file system (VFS) interface and updates the file's sequential or random access counter and time stamps. Then, the Rank Queue Manager updates the rank queues according to the file operations. When a new file is created, the Ranker assigns it a rank of 0 and inserts it into Q_{Rank0}.

The Rank Queue Manager maintains four rank queues, from Q_{Rank0} to Q_{Rank3}. Note that each queue is managed by a least-recently-used (LRU) algorithm. Newly created files or recently modified files are located in Q_{Rank0}. Thus, update-intensive files are always located in the front of Q_{Rank0}. The Rank Queue Manager locates the hot and sequential-read-intensive files in Q_{Rank1}. Hot and random-read-intensive files are located in Q_{Rank2}, and cold files are located in Q_{Rank3}. Q_{Rank3} exists only conceptually; the cold files are not directly managed by a queue.

When the system is idle, the Ranker and the Deduplicator run. The Ranker assigns a new rank to the files in Q_{Rank0}. When the Ranker assigns a new rank to a file, using the algorithm described in Fig. 5, the file is deduplicated by the Deduplicator according to its assigned rank. If the file is Rank 1, then the Deduplicator uses a big chunking method to preserve its data blocks' sequentiality on the disk. For

Table 1 Rank assignment and the associated deduplication policy

Rank	Characteristics	Deduplication mode
Rank 0	Need to be evaluated (Newly created or recently modified files)	None
Rank 1	Hot and sequential-read-intensive access pattern	big chunking
Rank 2	Hot and random-read-intensive access pattern	Small chunking
Rank 3	Cold	Small chunking

Rank 2 and Rank 3 files, it uses a small chunking method to maximize the deduplication ratio, because those files do not incur the sequential-read latency problem. To do this, the Deduplicator queries the file system to retrieve the target file blocks and hash tables, and then updates the hash table blocks and underlying file system's metadata to free redundant blocks. When the deduplication for a file is completed, the file is removed from Q_{Rank0} and inserted into its appropriate rank queue.

4.2 Rank Assignment by File Access Characteristics

4.2.1 File Ranking

We define four ranks according to each file's access pattern. Table 1 describes each rank, its characteristics, and its deduplication mode.

Rank 0 is assigned to files that need to be evaluated for a new rank. That is, the rank of newly created or recently modified files is 0. Note that update-intensive files will be naturally assigned to Rank 0.

Rank 1 is assigned to files that are hot and sequential-read intensive. For these files, we apply big chunking for data deduplication. Although the deduplication ratio likely decreases with big chunking, we can achieve a high performance sequential read.

Rank 2 is assigned to files that are hot and random-read intensive. Small chunking is applied to these files to achieve a high data deduplication ratio.

Rank 3 is assigned to files that are cold. A file is defined as cold if it has not been accessed for a given period (MAX_HOT_INT in Table 2), regardless of its access characteristics. Small chunking is applied to Rank 3 files for a high deduplication ratio.

To prevent unnecessary ranking operations, we maintain three parameters (see Table 2). The parameter MAX_HOT_INT determines the file's hot/cold state. When a file is accessed, it is marked as hot. However, if a file is not accessed for the interval MAX_HOT_INT, it is marked as cold.

To be assigned a new rank, a file needs sufficient evaluation time. This evaluation not only collects the file's access characteristics but also prevents unnecessary deduplication. The parameter MIN_EVALUATION_INT is the minimum evaluation interval needed to be ranked; the parameter MIN_EVALUATION_CNT is the minimum access count to be ranked. The default values were determined empirically by analyzing real-world workloads. The parameters are not

Table 2 Parameters

Type	Description	Default value
MAX_HOT_INT	Maximum interval from last access to retain hot state	2 weeks
MIN_EVALUATION_INT	Minimum evaluation interval. (Minimum written (modified) interval to be ranked.)	2 days
MIN_EVALUATION_CNT	Minimum evaluation count. (Minimum accessed count to be ranked)	100 times

Table 3 Information used to determine the rank of a file

Type	Field	Description
time	atime	Accessed time
time	mtime	Modified time
integer	seqCnt	Sequential access counter
integer	rndCnt	Random access counter

sensitive. As long as the values are not too small, the method will show similar results.

Table 3 shows the information used to determine the rank of a file. atime and mtime represent the access time and the modified time, respectively. In addition, seqCnt and rndCnt maintain the number of sequential accesses and random accesses of a file, respectively. These values are updated by the I/O Monitor module on every read/write access. The field atime is updated at every I/O operation. The field mtime is updated when the file is modified by a write operation.

For every read operation in a file, the I/O Monitor updates the seqCnt or rndCnt field according to the file-access pattern. The file-access pattern is determined by the file position. If the requested offset is the current file position $+1$ or -1, then the request is regarded as a sequential request and seqCnt is increased. Otherwise, the request is regarded as a random request and rndCnt is increased.When a file is ranked by the Ranker, the seqCnt amd rndCnt are reset to zero.

The ranking algorithm in Fig. 5 determines the rank of a file. The Ranker executes this algorithm when the system is idle. First, it checks the modified time to see whether a file has been recently updated. The recently updated files are assigned to Rank 0 to avoid deduplication. If not, the Ranker checks the recently accessed time to determine the hot/cold state. If the file is cold, then the file is assigned to Rank 3. If not, the Ranker checks the total monitored counts to see whether the file has enough evaluations to be ranked. If it has a high enough evaluation count, then it is assigned Rank 1 or Rank 2, based on the sequential access counter and the random access counter.

4.2.2 Queue Based Ranking Management

In order to assign or update a rank, we need an effective data structure to traverse all data files. We do not need to evaluate to rank already deduplicated, recently accessed, or recently modified files. To manage each file's rank effectively, we propose a multi-queue-based rank management scheme. In this system, three LRU queues named Q_{Rank0}, Q_{Rank1}, and Q_{Rank2} are proposed. The files are represented by the inode number in the queue. Q_{Rank0} maintains files that need to be evaluated. Newly created or recently modified files reside in Q_{Rank0}. Q_{Rank1} and Q_{Rank2} maintain files whose ranks are Rank 1 and Rank 2, respectively.

```
int rank(file_t file)
{
  if ( (NOW - file.mtime) < MIN_EVALUATION_INT)
    then return RANK0;
  else if ( (NOW - file.atime) > MAX_HOT_INT)
    then return RANK3;
  else if ( (file.seqCnt + file.rndCnt) < MIN_EVALUATION_CNT)
    then return RANK0;
  else if (file.seqCnt > file.rndCnt)
    then return RANK1;
  return RANK2;
}
```

Fig. 5 Ranking algorithm

Whenever a write operation is issued to a file in Q_{Rank0}, Q_{Rank1} and Q_{Rank2}, the file is moved to the top of $_{\text{Rank1}}$ to wait for assigning a new rank. For already ranked files, they still preserve their deduplicated state when they are moved to Q_{Rank0} since some chunks of the file may be referenced by other files.

The file in Q_{Rank0} will not be deduplicated if it is accessed again within the interval of MIN_EVALUATION_INT (in Table 2). For instance, there is a file that is updated once a day. This file moves to the top of Q_{Rank0} whenever it is modified. Thus, it will not be deduplicated since the Ranker checks Q_{Rank0} from the last entry. Even if it reaches the last entry, the Ranker does not assign a rank, owing to the ranking algorithm. Thus, update-intensive files are effectively filtered. On every file read operation, the file is moved to the top of its resident queue. Thus, the last entry is always the least-recently-used file, which makes it easy to find cold files.

Detailed queue management is described in Fig. 6. When a new file is created, it is inserted into Q_{Rank0}. If a file is modified by a write operation, then it needs to be reevaluated. Therefore, it moves to the top of Q_{Rank0} when it is modified. This operation not only gives a re-ranking opportunity to a file that was already assigned a rank, but also effectively prevents update-intensive files from being deduplicated.

In Fig. 6, Q_{Rank3} exists virtually. It conceptually maintains Rank 3 files, which are cold. If a file does not reside in Q_{Rank0}, Q_{Rank1}, or Q_{Rank2} then the file is Rank 3. If a cold file is accessed, it moves to the top of Q_{Rank0} to be evaluated again, because it is not cold anymore.

When the system is idle, the Ranker begins to look for cold files in Q_{Rank1} and Q_{Rank2}. Since these queues are ordered in an LRU manner, checking only the last entry is enough to find cold files. If the Ranker finds a cold file, the

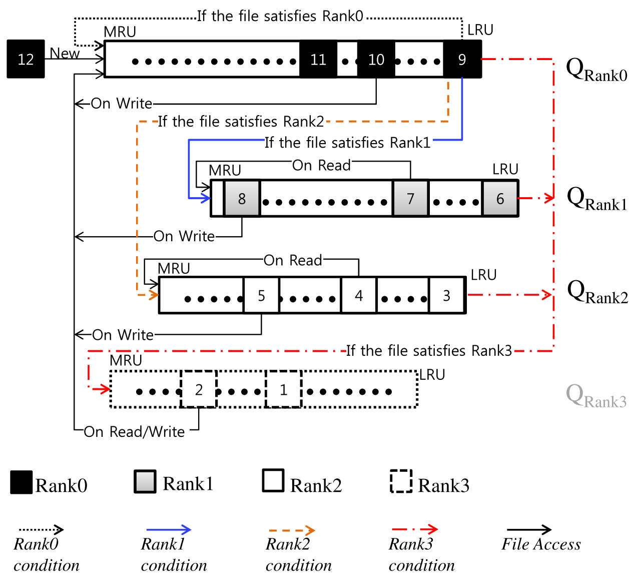


Fig. 6 Rank queue management. The purpose of the LRU queue is to efficiently find cold files and filter update-intensive files. The ranking conditions are determined by the ranking algorithm depicted in Fig. 5. Q_{Rank3} exists virtually because all cold files belong to Q_{Rank3}. There is no need to maintain them in an LRU queue.

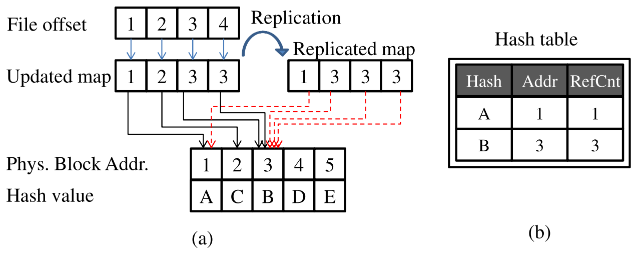
file is assigned to Rank 3 and the Deduplicator deduplicates the file using small chunking. The Ranker repeats this job until there are no cold files.

After that, the Ranker begins ranking from the last entry to the first entry of Q_{Rank0}. If the target file is again assigned to Rank 0, it is moved to the top of Q_{Rank0}. If the target file is assigned to Ranks 1, 2, or 3, the Deduplicator conducts deduplication according to its ranking, and the file is evicted from Q_{Rank0} and inserted to its corresponding queue. Note that the deduplication mode depends on the ranking. A big chunking method is used for Rank 1 files and a small chunking method is used for Rank 2 or Rank 3 files. After the deduplication operation is completed, the Ranker checks Q_{Rank0} again until there are no files that need to be assigned a new rank.

In Q_{Rank0}, already ranked (deduplicated) files can be inserted. Those are also re-deduplicated according to their new rank. In this case, some chunks of the file may be referenced by other files. The Deduplicator checks those chunks by the reference counter in the hash table. During a file is re-deduplicated, if a chunk of the file is not referenced, the reference counter of the chunk is decreased. If a reference counter of a chunk is zero, the chunk is removed from the hash table. Otherwise, the chunk is preserved in the hash table.

4.3 Offline Write Operation Support

In an offline system, the write operation requires special handling. A write operation with newly allocated blocks is not a problem. However, a write operation with already deduplicated blocks must be handled carefully, because some blocks may be shared by other files. If we update the entire data structure at every write operation, it will incur significant overhead. Actually, this kind of handling is an inline system, not an offline system.



Hash	Addr	RefCnt
A	1	1
B	3	3

Fig. 7 Updated map replication (a) and hash table (b) after file offset #2 is updated. When a file is updated, the original map is replicated and the replicated map preserves the original mapping information. After map replication is finished, the current map is updated. This makes the hash table and file mapping information consistent since the replicated map maintains the original mapping.

In order to support write operations for an offline system, we propose to update the map replication. In Fig. 7, if a write operation for an already-deduplicated block is issued, then the original file mapping information is replicated to a new block so that the original mapping is preserved. After the copy is complete, the file's mapping information updates according to the write operation.

The original map is preserved without any modification. Thus, the reference counter of the hash table is consistently maintained because the hash table is referenced by the original map. By doing this, the file update operation is performed without hash table update overhead.

The overhead for an update operation is a one-time map copying and a one-time new block allocation for the modified contents. Once a new block is allocated and the map replication is complete, later update operations for that block do not require any further overhead.

4.4 Space Overhead Analysis

To support the proposed system, three additional structures are required: a hash table, additional inode information, and the rank queues. Although the proposed method requires additional memory space for the hash table, it is only needed when the offline deduplication works. That is, there is no additional memory space overhead for the hash table while the deduplication is stopped. However, the system requires a small additional space for file information in the inode structure. As mentioned in Table 3, a file requires 16 additional bytes for the information. However, this value is small enough to fit in memory and only currently used files' inode entries are loaded into memory. Most of them are stored on the disk. Lastly, the sizes of Q_{Rank0}, Q_{Rank1}, and Q_{Rank2} depend on the number of hot files in the workload. If the total workload size is 1 TB, and the average file size is 100 KB, and 10% of the files are hot, then the total queue size is 4 MB. Note that the proposed system does not maintain Q_{Rank3} for the cold files.

5. Evaluation

For the evaluation, we used Ubuntu 12.04 LTS on an Intel Xeon E5620 2.4 GHz CPU with 4 GB of RAM. We set the big chunking size to 256 KB, and the small chunking size to 4 KB. In order to see the relationship between the data deduplication ratio and the sequential-read latency, we experimented on file system data sets, including synthetic and real-world data sets. In this experiment, we can observe the runtime overhead and the performance interference of the offline operation of the deduplication thread.

5.1 Prototype

We built a prototype on top of a FUSE-based [20] Ext2 file system [21]. FUSE stands for File system in User Space. Many file systems are implemented using FUSE owing to its convenience and simplicity. Although it works in the user space, it directly accesses block devices and manages block de/allocations without any help from the underlying file system. This is important for the prototyping of the proposed system because, if the block management is performed by the underlying file system that serves the FUSE binary, our experiments may show abnormal results.

The prototype consists of the I/O monitor layer, rank queues, ranker thread, and the deduplication engine. When the system is idle, the ranker thread fetches a target file from Q_{Rank0}. The ranking algorithm in Fig. 5 determines the rank for the fetched file, and the deduplication engine selectively deduplicates the file according to its rank. Big and small chunks are stored in the chunk store. Though they are classified into two different sizes, they are essentially a set of blocks that are managed by the file system, because we use the file system's block-mapping structure as the deduplication mapping structure. The hash table resides in the deduplication engine. The hash table index consists of a SHA-1 [22] hash value, the address of the chunk, and the reference counter. The size of a hash table entry is 28 bytes (20 bytes for the SHA-1 hash, 4 bytes for the reference counter, and 4 bytes for the physical block number of the first block of a chunk). The file information is stored in the inode structure. The file information size is 16 bytes, as depicted in Table 3.

5.2 Sequential Latency vs. Deduplication Ratio

In order to evaluate how the proposed selective deduplication method works, two different workloads were considered; a synthetic data set and a real-world data set. From these results, we can see the trade-off between the deduplication ratio and the sequential-read latency. In this paper, deduplication ratio is calculated by whole size after deduplication over whole size before deduplication.

5.2.1 Synthetic Data Set

In order to see the detailed behavior of the selective deduplication method, we evaluated the proposed system based on a synthetic workload. We generated two file sets with low and high deduplication ratios, because the sequential-read latency depends on the deduplication ratio. We generated the file sets with various file size range to see practical effect. Table 4 shows detailed description of the workloads including their deduplication ratio for big-sized chunking and ssmall-sized chunking method and Fig. 8 shows their file size distribution.

In order to evaluate sequential read latency, we make the workloads deduplicated state. For big-sized and small-sized chunking methods, we conducted offline data deduplication by their chunking size to the workloads since these methods do not require runtime read characteristics. For selective deduplication, we create the workloads on the proposed system and we read whole files according to the sequential read ratio from 90% to 10%. Then the proposed method assigns a rank for each file and deduplicates it. After each workload is deduplicated, we evaluate read latency by reading all files in the workload according to its sequential read ratio from 90% to 10%. In this experiment, 10% of sequential read ratio means, 10% of randomly chosen files in the workload are read sequentially and 90% of the other files are read randomly.

By this experiments, we can observe the accuracy of the file-read-pattern detection method of the ranking algorithm and compare the sequential-read latency and the deduplication ratio. Table 4 shows a detailed description of the workload.

Figures 9 and 10 show the response time cumulative

Table 4 Synthetic workload description

Workload	Synthetic workload #1	Synthetic workload #2
Characteristics	High deduplication ratio	Low deduplication ratio
Total workload size	2.5GB	2.5GB
File size distribution	4KB-32MB	4KB-32MB
Number of files	About 5,000 files	About 5,000 files
Deduplication ratio with big chunking	20%	5.1%
Deduplication ratio with small chunking	70.3%	20.5%

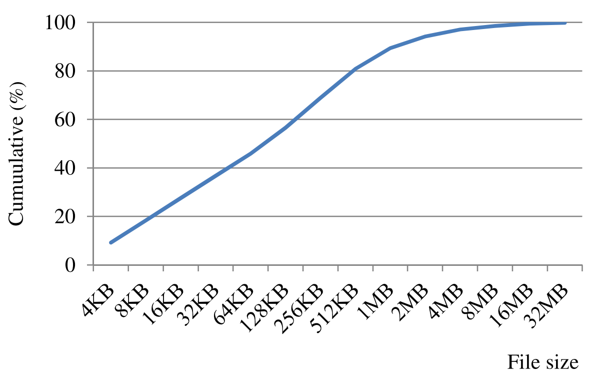


Fig. 8 File size distribution for the synthetic workloads described in Table 4. The two synthetic workloads have the same file size distribution. We generate a set of files with various range (4KB - 32MB).

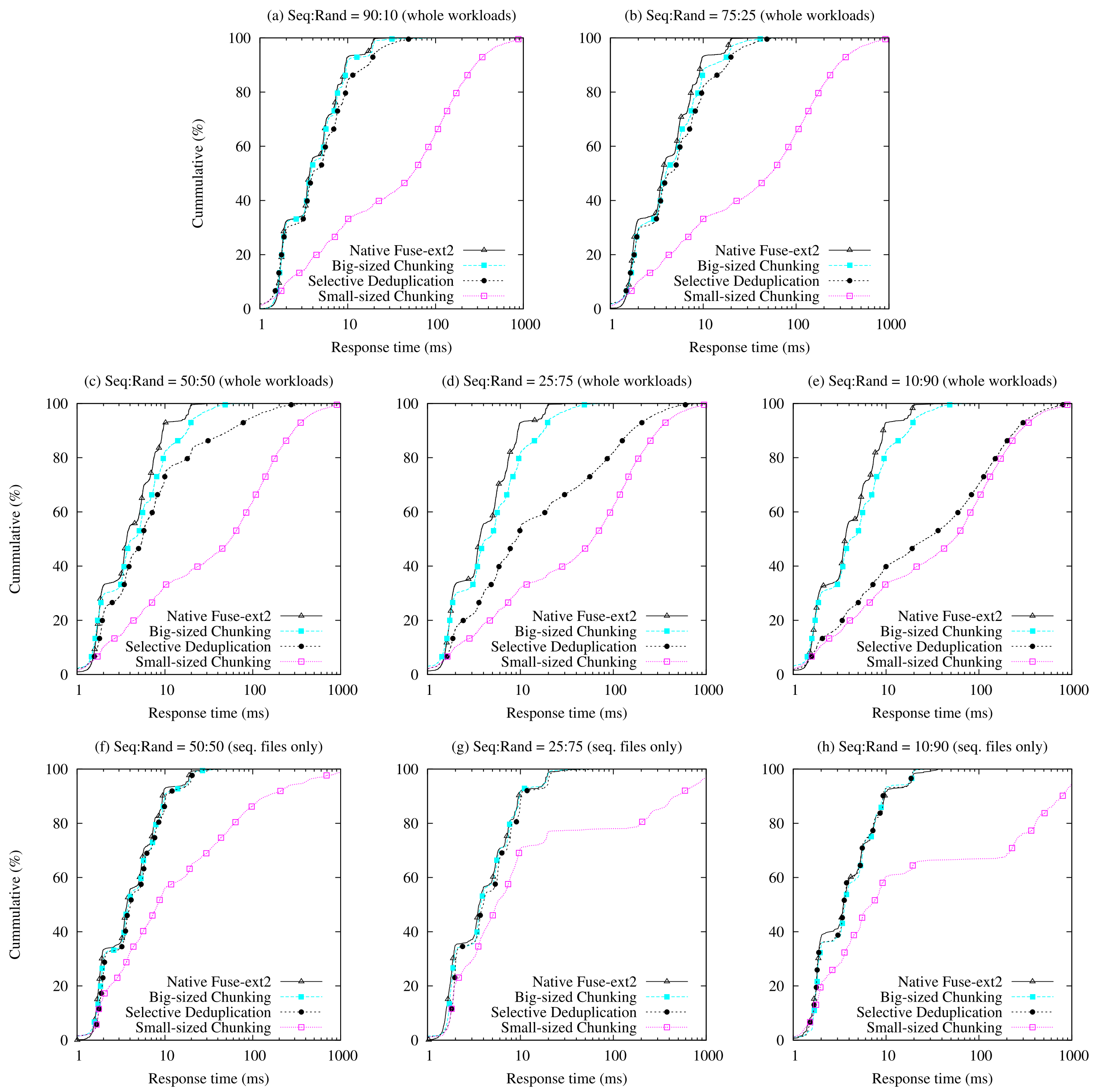


Fig. 9 Analysis of synthetic workload with high deduplication ratio. (a) - (e) show the response time CDF results of various read operations from 90% sequential to 10% sequential for complete workloads. (f) - (h) show the response time CDF results using a set of files accessed only by sequential read during the experiment to see the exact effect of the selective deduplication. Native Fuse-ext2 means the unmodified Fuse-ext2 file system. The other experiments were conducted under the proposed system based on the Fuse-ext2 file system.

distribution function (CDF) and the deduplication ratio results of the synthetic workloads with a high deduplication ratio and low deduplication ratio, respectively. In order to generate the CDF figures, we used response time per each file. Specifically, the figures from (a) to (e) are the results of reading whole files in the workloads, and the figures from (f) to (h) are the results of reading only sequentially read files, to see the exact effect of the selective deduplication. By comparing the two synthetic workloads, we can observe that the difference in read latency between big chunking and small chunking depends on the deduplication ratio.

We can observe the runtime I/O monitoring overhead of the proposed system by comparing the selective deduplication with the Native Fuse-ext2. Figures 9-(a) and 10-(a) which have little read latency in the random workload-have a very small read-latency difference between them. That is,

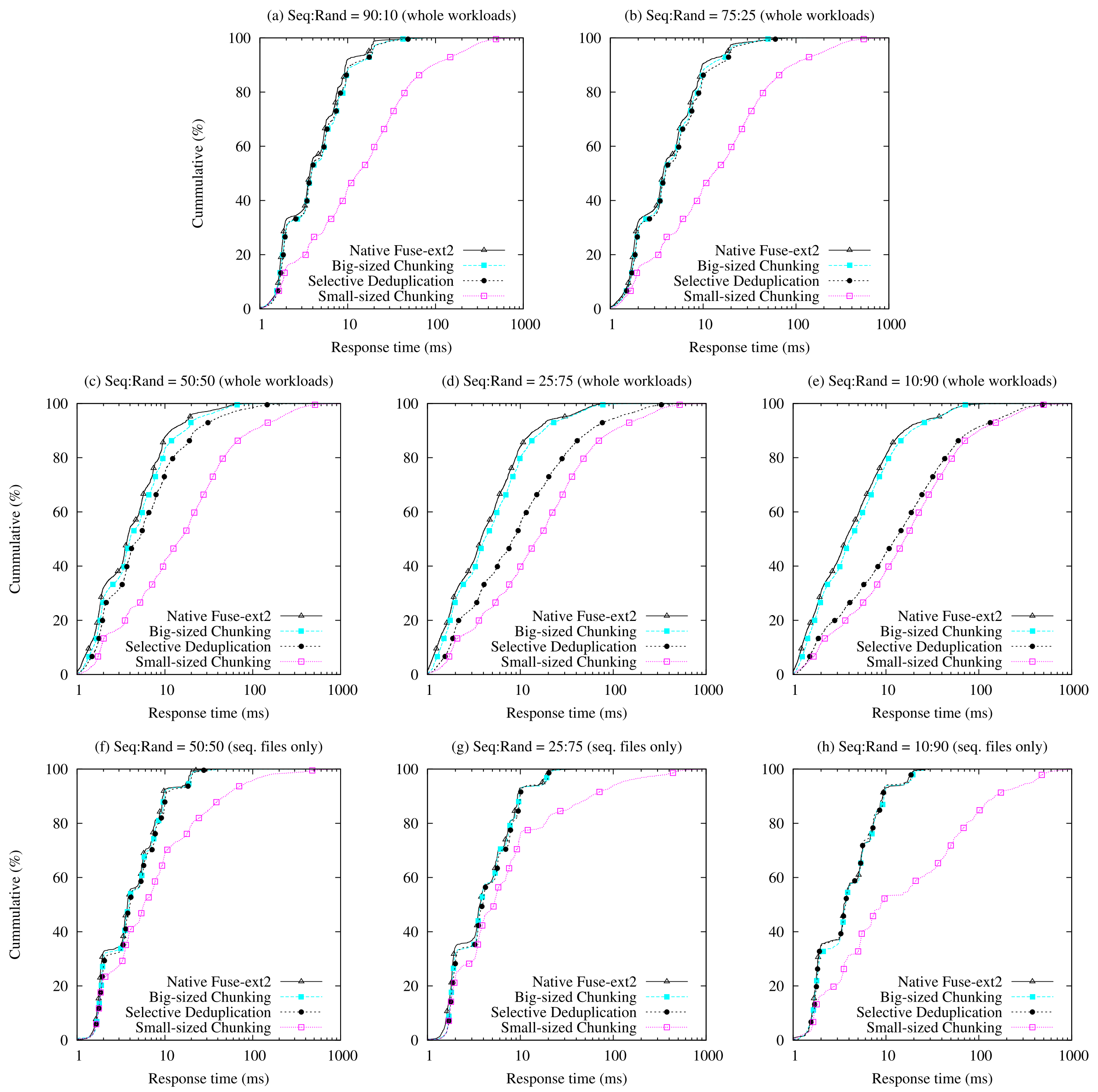


Fig. 10 Analysis of synthetic workload with low deduplication ratio. (a) - (e) show the response time CDF results of various read operations from 90% sequential to 10% sequential for complete workloads. (f) - (h) show the response time CDF results using a set of files accessed only by sequential read during the experiment to see the exact effects of selective deduplication. Native Fuse-ext2 means the unmodified Fuse-ext2 file system. The other experiments were conducted under the proposed system based on the Fuse-ext2 file system.

the I/O monitoring overhead of the proposed system is negligible from the perspective of read latency.

We also compared response time of the proposed selective deduplication with the big and small chunking methods. The selective deduplication successfully supports low response time for sequentially read files. The CDF of the big chunking and the small chunking methods from Figure 9-(a) to Figure 9-(e) show almost the same read latency, regardless of the workload. However, the selective deduplication shows various read latencies depending on the sequential-read ratio because this method deduplicates files based on the sequential-read pattern. In the case of the 90% sequential-read workload, the selective deduplication CDF is similar to the big chunking CDF (Fig. 9-(a)). In contrast, in the case of the 10% sequential-read workload (Fig. 9-(e)), the CDF of the selective deduplication is similar to the CDF of the small chunking method, because 90% of the workload is random-read files that are deduplicated with small chunking. However, for the sequential-read-workload files only, we can still achieve high performance(Fig. 9-(h)) owing to the characteristics of the selective deduplication. In the case of the workload with a low deduplication ratio (Fig. 10), the same trends were shown. This means the selective deduplication works well, regardless of the deduplication ratio.

Figure 11 shows that the selective deduplication method effectively controls the trade-off between the deduplication ratio and sequential-read performance. The result shows that the selective method achieves as high a read performance as the big chunking method; in addition, it achieves a higher deduplication ratio than the big chunking method. The deduplication ratio of the big and small chunking methods is a fixed value because they do not consider the workload characteristics. However, the selective method effectively supports high performance sequential reads. The workload of 10% sequential read ratio achieves a deduplication ratio almost equal to the small chunking method. The throughput for the selective method is similar to the small chunking method's results in Fig. 11-(a). This is because 90% of the files are deduplicated using small chunking. However, as shown in Fig. 11-(b), the selective method successfully achieves 97% of the sequential-read performance of a native file system, and 86% of an ideal deduplication ratio for the 10% sequential-read-oriented files.

In Fig. 11-(a), the throughput of the proposed method is quite low in the case of the random workload (i.e., S:R = 25:75 and 10:90). This is because the random-read performance depends on the physical distance of the blocks. That is, if a file is fragmented throughout the whole disk volume, the random-access performance is also affected. One possible solution is a container-based approach. If data deduplication is conducted inside a container, as opposed to the entire volume, then the block distance will be much closer.

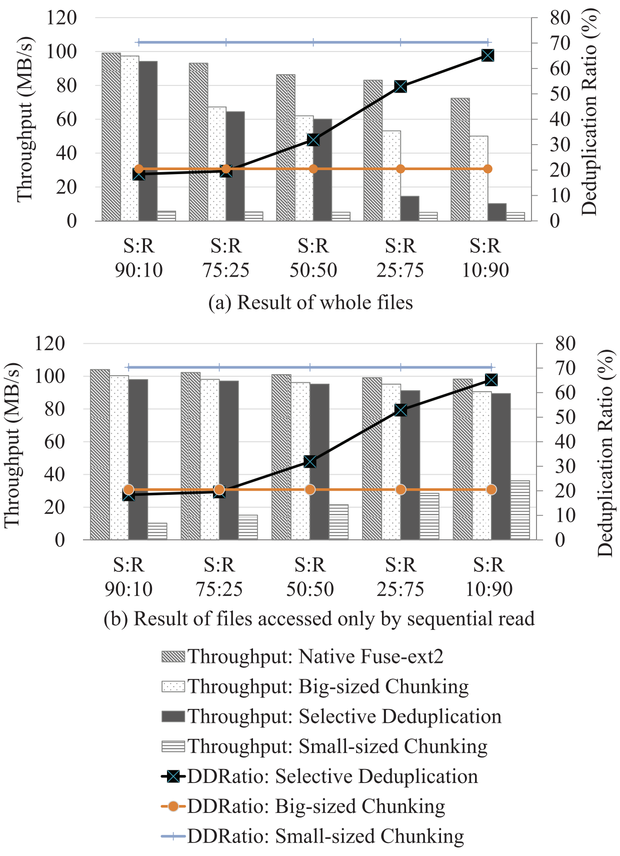


Fig. 11 Sequential-read throughput versus deduplication ratio for the sequentially read files versus deduplication ratio. The bars depict the throughput and the lines depict the deduplication ratio. In this experiment, we used the high deduplication ratio workload. The selective deduplication method achieves a high deduplication ratio and a high read throughput at the same time. The x-axis' S:R means Sequential:Random ratio.

5.2.2 Real-World Data Set

We chose two distinct users' workloads for the real-world data set: a normal desktop user's workload and a developer's workload. Table 5 describes the characteristics of each workload and Fig. 12 shows their file size distribution. For experiment, we copied the whole files of the workloads into the proposed system and we ran each workload for two weeks. The desktop user mainly uses web-browser and word-processor and the developer mainly uses vim editor and gcc compiler. Table 5 shows the overall results for the two workloads. The proposed system can achieve almost the same deduplication ratio as that using small chunking. Moreover, it also achieves a low sequential-read latency at

Table 5 Real-world workload description

Workload	Real-world workload #1	Real-world workload #2
Characteristics	Desktop User	Developer
Mainly used applications	Web-browser, word processor	Vim editor, gcc compiler
Total workload size	42GB	90GB
Operating System	Ubuntu 11.10	Ubuntu 10.10
File size distribution	0B-2.2GB	0B-1.6GB
Number of files	About 50,000 files	About 120,000 files
Files accessed within the last two weeks	About 5%	About 13%
Files unmodified for a year	About 50%	About 70%

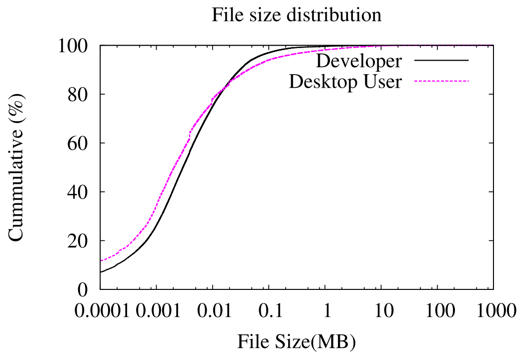


Fig. 12 CDF of file size distribution for the real-workloads

the same time.

Figure 13 shows the CDF results of the sequential-read response time for all files in the desktop user workload. We excluded the CDF results of the developer workload because it showed the same trends. The performance difference between Native Fuse-ext2 and Not Deduplicated is almost the same. Therefore, the runtime overhead for I/O monitoring is negligible in terms of read latency. Since the entire workload's sequentiality is low, the selective deduplication's CDF result is close to the small chunking method. However, if we observe the results for the sequential-read-oriented files, it achieves a low read latency similar to the big chunking method. Figure 14 depicts the results of sequentially read files (i.e., files that are assigned Rank1). It shows that the proposed system works well for the sequential-read-oriented file's read latency. It is almost the same as the big chunking method's read performance.

5.3 Performance Interference over Normal I/O Operations

During offline operations, normal I/O operations may be issued. It is inevitable for performance interference to occur. In this experiment, we show how the proposed method interferes with the performance of normal I/O operations.

5.3.1 Runtime Overhead

The proposed system has runtime overheads caused by I/O monitoring and queue management. The well-known standard system utilities and benchmarks in Table 7 are considered to evaluate the runtime overhead of the proposed method.

Figure 15 shows the results of the normalized performance reduction for the workloads. One can observe that the disk-I/O-intensive workloads had less than 7% runtime overhead. Netperf had no performance degradation since it does not issue disk I/O commands.

5.3.2 Performance Interference by Offline Operation

Because offline operations utilize system idle time, we need to measure how much idle time is available in a given workload. The longer the system is idle, the less is the interference over normal I/O operations. Many prior studies on idle time and power saving have shown that there is sufficient idle time. Robertson et. al. showed that more than 600 desktops were always left on out of 700 total [25], and Agarwal et. al. observed that two-thirds of office PCs were left on after hours [26]. Many network-connected computers

Table 6 Result of deduplication for the real world workloads

Workload	Rank 0 (%)	Rank 1 (%)	Rank 2 (%)	Rank 3 (%)	Dedupe. ratio using proposed system (%)	Dedupe. ratio using small chunking (%)	Dedupe. ratio using big chunking (%)
Desktop User	2.8	5.9	4.1	87.2	14.3	16.1	11.2
Developer	0.9	2.2	1.2	95.7	31.1	32.8	17.2

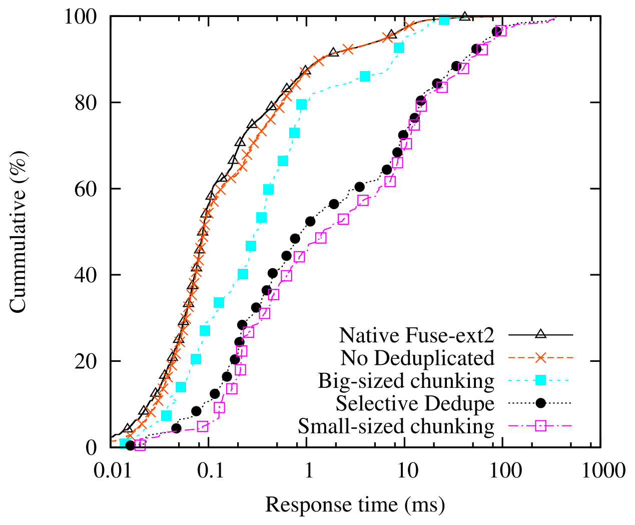


Fig. 13 Response time CDFs of sequential reads for all files of the desktop user workload.

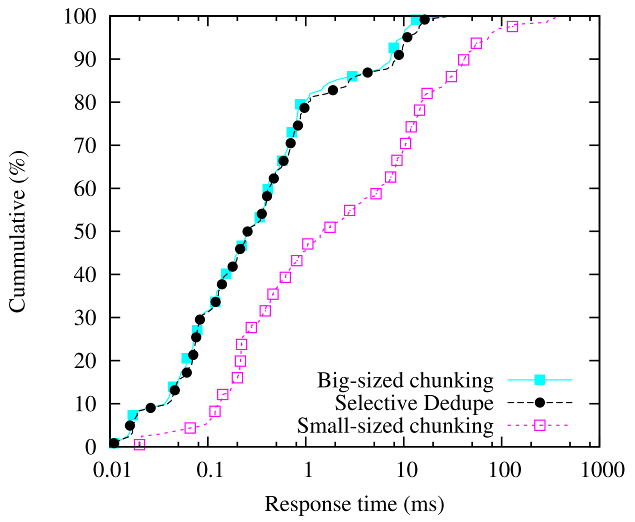


Fig. 14 The response time CDF of sequential read of the workload Desktop User only for sequentially read files (i.e., files that are assigned Rank 1)

Table 7 Standard system utilities and benchmarks

Name	Description (Characteristics)	Workload
tar	Archiving system utility (Disk I/O intensive)	Linux 2.6.24 source files
untar	Un-archiving system utility (Disk I/O intensive)	Linux 2.6.24 source files
grep	String searching system utility(Disk I/O intensive)	Linux 2.6.24 source files
cp	File copying system utility (Disk I/O intensive)	Linux 2.6.24 source files
Filebench [23]	A benchmark that simulates various type of file based app. (Disk I/O intensive)	Webserver with default parameters
Netperf [24]	A benchmark that measures network performance (Network I/O intensive)	Default parameters (Local-to-Local TCP)

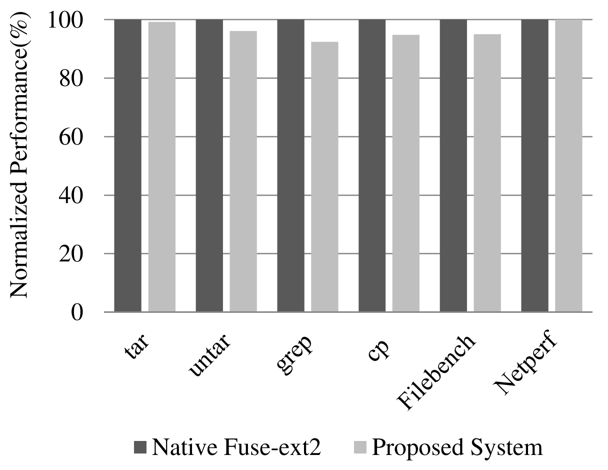


Fig. 15 Normalized performance reduction for various applications.

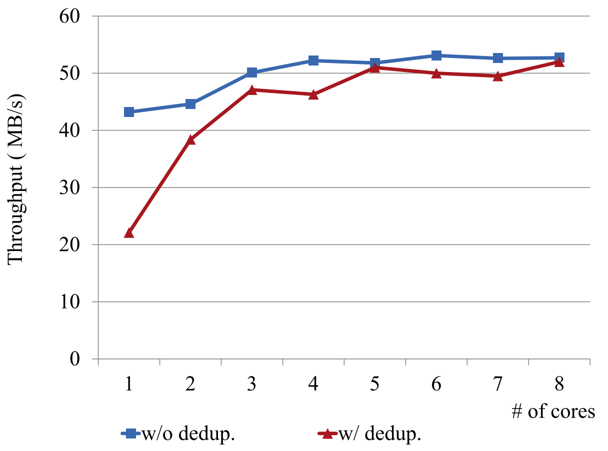


Fig. 16 Performance interference of the data deduplication thread. The webserver workload in the Filebench benchmark tool was used, which runs 100 simultaneous I/O threads.

are always on because users want their PCs to maintain the always-on status, and most of them remain idle [27], [28]. Especially in [27], enterprise desktops remained idle for an average of 12 hours per day.

In the case of our experiments shown in Table 5, only a few files were assigned to Rank 1 or Rank 2. That is, only a few files were targets for offline deduplication because most files had already been deduplicated by Rank 3 as time went by. Typically, when a user has 1 TB of storage, the deduplication targets will be almost 50 GB of files (5%). It takes less than an hour to deduplicate a 50 GB file set using the Rank 3 deduplication method.

For the worst-case analysis in performance interference over normal I/O operations, we considered two distinct applications from Table 7; the Filebench benchmark to represent a disk-I/O-intensive workload and the Netperf benchmark to represent a non-disk-I/O workload.

Since the offline data deduplication uses a single thread, we experimented with various numbers of cores to see the overhead on a multi-core environment.

Figure 16 shows the results of the Filebench bench-

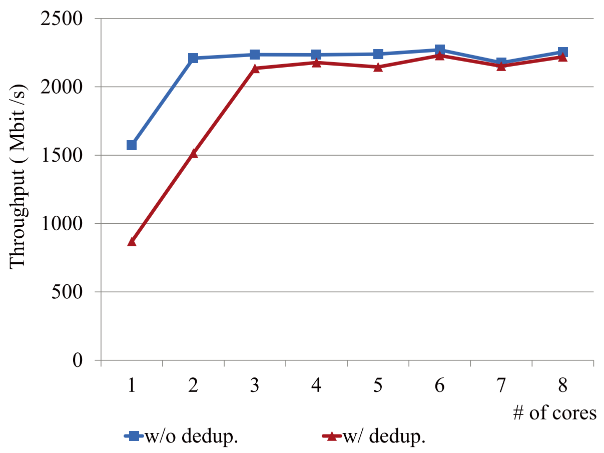


Fig. 17 Performance interference results for Netperf, which does not issue disk I/O operations. We ran the offline data deduplication thread while Netperf was being benchmarked.

mark. It shows that performance interference existed but was negligible in a multi-core environment, even for an I/O-intensive workload. Owing to the complexity of the hash value calculation and comparison, the data deduplication thread requires considerable CPU time rather than disk I/O. Thus, the performance interference in a multi-core environment is negligible, even if the data deduplication thread is running.

Figure 17 shows the results of the Netperf benchmark. In the previous experiment, Netperf was not affected by the runtime overhead of the proposed system since it does not issue disk I/O. However, its performance was affected by the proposed system when the offline deduplication thread was running. However, if we use three or more cores, the performance reduction is also negligible.

Therefore, whether an application is I/O intensive or not, the performance interference caused by offline operations will not be considered a problem in modern computer systems.

6. Conclusion

In this paper, we proposed a low-overhead selective offline data deduplication method for primary storage. It selectively deduplicated data according to the access patterns so that it supports a low I/O latency and a high data deduplication ratio at the same time. To accomplish this, we defined four ranks according to the characteristics of a file; the multi-queue-based rank management scheme effectively handled each rank. Through the experiments, we observed that it achieved a maximum of 86% of an ideal deduplication ratio and 97% of the sequential-read performance of a native file system. We also noted that the runtime overhead is less than 7%.

References

[1] L. Dubois and M. Amalsas, “Key considerations as deduplication evolves into primary storage,” White Paper, IDC, May 2010.

[2] B. Zhu, K. Li, and R.H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," FAST, pp.1–14, 2008.

[3] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," FAST, pp.197–210, 2009.

[4] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," FAST, pp.111–123, 2009.

[5] W. Dong, F. Douglis, K. Li, R.H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," FAST, pp.15–29, 2011.

[6] D.T. Meyer and W.J. Bolosky, "A study of practical deduplication," ACM Transactions on Storage (TOS), vol.7, no.4, p.14, 2012.

[7] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput," USENIX Annual Technical Conference, 2011.

[8] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," USENIX Annual Technical Conference, 2011.

[9] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," ACM SIGOPS Operating Systems Review, pp.174–187, ACM, 2001.

[10] K. Srinivasan, T. Bisson, G.R. Goodson, and K. Voruganti, "idedup: latency-aware, inline data deduplication for primary storage," FAST, pp.1–14, 2012.

[11] C. Alvarez, "Netapp deduplication for fas and v-series deployment and implementation guide," Technical Report, TR-3505, 2011.

[12] EMC, "Achieving storage efficiency through emc celerra data deduplication," White paper, March 2010.

[13] IBM, "IBM storage tank — a distributed storage system," White paper, Jan. 2002.

[14] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," USENIX Annual Technical Conference, pp.285–296, 2012.

[15] M. Cao, S. Bhattacharya, and T. Tso, "EXT4: The next generation of EXT2/3 filesystem," 2007 Linux Storage & Filesystem Workshop, 2007.

[16] T.J. Gibson and E.L. Miller, "Long-term file activity patterns in a unix workstation environment," Proc. 15th IEEE Symposium on Mass Storage Systems, pp.355–372, 1998.

[17] A.W. Leung, S. Pasupathy, G.R. Goodson, and E.L. Miller, "Measurement and analysis of large-scale network file system workloads," USENIX Annual Technical Conference, pp.5–2, 2008.

[18] K. Ramakrishnan, P. Biswas, and R. Karedla, "Analysis of file i/o traces in commercial computing environments," ACM SIGMETRICS Performance Evaluation Review, vol.20, no.1, pp.78–90, 1992.

[19] A.J. Smith, "Analysis of long term file reference patterns for application to file migration algorithms," Software Engineering, IEEE Trans., no.4, pp.403–417, 1981.

[20] M. Szeredi, "File system in userspace, fuse, http://fuse.sourceforge.net."

[21] T.Y.T. R. Card and S. Tweedie, "Design and implementation of the second extended filesystem," Amsterdam Linux Conference, 1994.

[22] FIPS Pub., "180-1. secure hash standard," National Institute of Standards and Technology, vol.17, 1995.

[23] R. McDougall, "Filebench: Application level file system benchmark,"

[24] R. Jones et al., "Netperf: a network performance benchmark," Information Networks Division, Hewlett-Packard Company, 1996.

[25] J.A. Roberson, C.A. Webber, M.C. McWhinney, R.E. Brown, M.J. Pinckard, and J.F. Busch, "After-hours power status of office equipment and inventory of miscellaneous plug-load equipment," Lawrence Berkeley National Laboratory, 2004.

[26] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta, "Somniloquy: Augmenting network interfaces to reduce pc energy usage," NSDI, pp.365–380, 2009.

[27] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft, "Skilled in the art of being idle: Reducing energy waste in networked systems," NSDI, pp.381–394, 2009.

[28] T. Das, P. Padala, V.N. Padmanabhan, R. Ramjee, and K.G. Shin, "Litegreen: Saving energy in networked desktops using virtualization," USENIX Annual Technical Conference, 2010.

Sejin Park received a B.S. degree in software engineering from Kumoh National University of Technology, Korea in 2007. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering, POSTECH, Korea. His research interests include virtualization technology, storage systems, and embedded systems

Chanik Park received a B.S. degree in 1983 from Seoul National University, Seoul, Korea, and an M.S. degree and Ph.D. in 1985 and 1988, respectively, from Korea Advanced Institute of Science and Technology. Since 1989, he has been working for POSTECH, where he is currently a professor in the Department of Computer Science and Engineering. He was a visiting scholar with the Parallel Systems group in the IBM Thomas J. Watson Research Center in 1991, and was a visiting professor with the Storage Systems group in the IBM Almaden Research Center in 1999. He has served at a number of international conferences as a program committee member. His research interests include storage systems, operating systems, and virtualization technology.