IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 25, 2020, accepted March 17, 2020, date of publication March 23, 2020, date of current version April 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2982688

AvaTar: Zero-Copy Archiving With New Kernel-Level Operations

HYUNCHAN PARK[1], YOUNGPIL KIM[2], (Member, IEEE), AND SEEHWAN YOO[3], (Member, IEEE)

[1]Division of Computer Science and Engineering, Jeonbuk National University, Jeonju 54896, South Korea
[2]School of Computer Science, Semyung University, Jecheon 27136, South Korea
[3]Department of Mobile Systems Engineering, Dankook University, Yongin 16890, South Korea

Corresponding authors: Youngpil Kim (ypkim@semyung.ac.kr) and Seehwan Yoo (seehwan.yoo@dankook.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) under Grant NRF-2015R1C1A1A02037330, Grant NRF-2017R1C1B5016000, and Grant NRF-2020R1G1A1005544, and in part by the "Research Base Construction Fund Support Program" through the Jeonbuk National University in 2017.

ABSTRACT The problem associated with current file archiving systems is a slow processing time owing to unnecessary data copying. To address this problem, a novel archiving system with zero-copy merging and splitting operations, referred to as AvaTar, is presented herein. For the operations, instead of copying the data, the block allocation information of the files is manipulated at the kernel level. We implemented kernel-level archiving primitives in a Linux kernel, called AvaTar-FS, and a user-level archiving tool, called AvaTar agent. Our evaluation results indicated that AvaTar required only 0.48 s to extract 1,024 files from a 4 GB archive file, which is 132-times faster when compared with traditional GNU Tar archiving. AvaTar affords practical benefits in uploading files to a real-world cloud storage system, and successfully completes the transfer of 1,024 files to Amazon Web Service cloud storage within 60.55% of the processing time required through a traditional approach.

INDEX TERMS Merging and splitting, archiving and extraction, zero-copy, file system, cloud storage system.

I. INTRODUCTION

Sharing files over cloud storage is easy. Once the setup is complete, numerous files placed in resources such as Dropbox or Amazon S3 directories can be automatically synchronised and immediately accessed by any computer [16]. However, installing a new cloud storage client for these commercial resources is a tedious task because all files have to be synchronised in the cloud on first use [23]. Small and numerous file transfers are prevalent on the cloud [6], [17], and they hinder the best-utilising network bandwidth. It is widely known that short-lived TCP sessions suffer from a low network utilisation during the slow-start phase, and small files transfer lead to slower start-up sessions. If we consider the use of UDP, files can be sent at maximum speed, although with decreased reliability of the communication channel. Therefore, transmitting small files to a cloud is inefficient.

A popular and convenient solution to the problem of an inefficient file transmission is the archiving of small files. We can efficiently utilise the network bandwidth if we merge small files into a single large file and send them all at once. After a single TCP slow-start, the session can fully utilise the network bandwidth. The receiver can then extract the archived file so that it has the original files.

Archiving and extraction are widely used operations for managing large numbers of files. For example, we create an archived file as a backup, and send it as an e-mail attachment or keep it in cloud storage such as Amazon S3 Glacier. However, the current archiving technique has a performance problem. Existing archiving applications, such as GNU Tar and GNU Ar, are extremely slow because the existing archiving utilities require a large number of data copies. To archive the files, we should read the data from the source files and then write the data into the destination file. To extract files from the archive, we should read and write data in a reverse manner. These readings and writings involve numerous accesses for the storage device, which is normally the slowest operation in a computer system. The utilities also copy the data between the user and kernel memory spaces. In fact, such accesses and copies may be unnecessary because all data are already in

The associate editor coordinating the review of this manuscript and approving it for publication was Lo'ai A Tawalbeh.

the storage device. Current archiving and extraction utilities simply create identical data blocks in the storage device through excessive readings and writings, wasting precious system resources.

To resolve the problem of slow and inefficient archiving, this paper presents a novel archiving system with kernel-level supports, referred to as AvaTar. AvaTar makes two main contributions to the current file archiving system. First, AvaTar is the first zero-copy archiving system that executes archiving and extraction extremely quickly. Because AvaTar does not incur any data copies in either memory or storage, it is faster and more efficient than traditional archiving systems. To remove unnecessary data copies, AvaTar uses zero-copy merging and splitting file operations, which are newly implemented in the OS kernel. These operations relocate data blocks between in-kernel data structures of the source and destination files. Because the operations do not incur data copies in the storage devices or between the user and kernel memory space, AvaTar is extremely fast and efficient. Thus, AvaTar uses zero-copy file merging and splitting as primitive operations for archiving and extraction.

Second, AvaTar provides high-level compatibility by using a file system image format as its archive file format. Thus, a system without AvaTar can mount an archived file without data copies. This is a novel idea that enables a system without new operations to benefit from a zero-copy extraction. This cannot be easily accomplished through a simple implementation using kernel-level merging and splitting proposed in prior studies. AvaTar is the only system providing such benefits.

We implemented and evaluated AvaTar using the Ext4 file system and GNU Tar on Linux. Our evaluation results indicated that AvaTar required only 0.48 s to extract 1,024 files from a 4 GB archive file, which is 132-times faster when compared with traditional GNU Tar archiving. We also conducted several real-world experiments including transferring the files over the commercial cloud system, Amazon Web Services (AWS). Our experimental results demonstrate that AvaTar transfers files to the cloud storage on AWS 1.65-times faster compared to a traditional method when we transfer an 1 GB file containing 1,024 files.

The remaining sections are organised as follows: section II describes the background and related studies conducted on merging and splitting file operations; section III describes the design of AvaTar and its components in detail; section IV details its implementation and evaluation on a Linux system; section V details our future considerations regarding AvaTar; and section VI provides some concluding remarks regarding this research.

II. BACKGROUND AND RELATED WORK

A. DATA BLOCK MANAGEMENT IN I-NODE

In modern file systems, there are two methods for managing data blocks in an I-node: using block numbers and using extents. When using block numbers, the numbers of data blocks are directly written into each I-node, and when using extents, the metadata of extents indicating physically adjacent data blocks are written [10].

I-nodes with block numbers are used in numerous UNIX-like traditional file systems such as Unix file system (UFS) [19], Ext2 [7], and Ext3 [26]. In these file systems, data blocks consist of direct and indirect blocks. A direct block stores a pointer (block number) to a data block. An indirect block contains references to direct or other indirect blocks. Depending on the level of indirect reference, indirect blocks are classified into three types: single, double, and triple indirect blocks. Traditional file systems efficiently trade performance with capacity, leveraging the depth of both direct and indirect blocks. For small files, direct blocks are used for quick and easy access, whereas indirect blocks are used for storing a number of data blocks for large-size files.

I-nodes with extents have been used in some recent file systems including Ext4 [18], BtrFS [24], NTFS [20], and XFS [25]. An extent is a group of contiguous data blocks, and is represented by a range of blocks. Two metadata are used for the range, namely the number of the first block and the number of contiguous blocks in the extent. The typical data structure used for writing an extent is a tree. Extents require a smaller storage space for metadata than writing blocks, and thus the number of I/O operations for the metadata is also reduced. Eventually, the performance of the file access to the user data is also enhanced. Although the fragmentation occurring on an extent-based file system degrades the I/O performance, we can mitigate this through a pre-allocation or lazy allocation of contiguous data blocks. Owing to these advantages, most modern file systems generally adopt an extent-based i-node. Therefore, we implemented our AvaTar prototype on an extent-based file system, Ext4.

B. RELATED WORK

Y. Kim *et al.* [13] suggested merging and splitting operations for large-size multimedia files. In their study, the splitting operation divides a single file into two parts for a fast modification of a portion of a movie file, and a merging operation is used to form them into a single file. These operations run on the OS kernel, and are implemented on Ext4, FAT, and exFAT file systems. The authors focused on in-kernel operations, and not on an end-user abstraction. To provide an archiving facility for the end-user, AvaTar includes a user-level component that is responsible for creating metadata for the archived files and for an extraction of the files from an archive. In addition, the authors consider only two file cases, which limits the functionality as a general archiving tool.

Y. Yoo *et al.* [29] suggested a remove operation for the middle parts of a file by manipulating an i-node. When removing the middle parts of a file in conventional file systems, the remaining data after the removal should be read and written back as a new file. Thus, the performance of the file systems can be largely degraded because it takes much time to copy the remaining data. The authors' scheme changes the mapping information between the logical file address and

physical block address when removing middle parts of a file. AvaTar provides more general abstractions for the merging and splitting file operations. With AvaTar's two abstractions, we can manipulate any locations of a file, regardless of its removal or attachment.

S.W. Jung *et al.* [12] attempted to represent an arbitrary file by connecting multiple blocks. The authors built a prototype system that manages a linked list of allocated blocks in an i-node such that the list can be recognised as a single file. Although the structure allows a flexible relocation of blocks, the authors did not propose specific operations, such as splitting or merging operations. In addition, they did not provide any performance evaluations. Compared to their approach, AvaTar presents a complete design, and its efficacy was validated through extensive experiments conducted on commercial cloud applications.

Several studies have addressed the issues of archiving systems used for storage purposes [8], [11], [21], [28] and the archiving method for several domains [14], [27]. Y. Diao *et al.* suggested StonesDB which is the database for archiving the data from a sensor network on the flash-based storage device [8]. Their main goal is to provide energy-efficient data storage in a specific environment that has a hierarchical node structure. T. F. Gosnell suggested a secure data archiving system with various functionality for management such as deleting the expired information without leakage or falsification [11]. R. Ohran *et al.* suggested a method for archiving and mirroring mass storage. Their main idea is the careful synchronization of the two mass storage to preserve the data integrity [21]. T. Yang *et al.* suggested DEBAR, the backup and archiving system using the deduplication [28]. The goal of DEBAR is to present a space-efficient archiving system while providing a scalable performance. V. Kobla *et al.* suggested several features including a extraction, key-frame indexing, and retrieval for a compressed video storage system [14]. J.M. Vau *et al.* suggested a communication technique for the multimedia messages between two servers [27]. They focused on avoiding data loss while transferring.

However, in terms of feasibility, previous approaches have commonly suffered from a significantly poor performance of the user-level archiving and extraction. In this regard, AvaTar provides a differentiated archiving and extraction performance using a zero-copy technique.

III. DESIGN

In this section, we present the design of AvaTar. First, we describe the goals and main ideas required to satisfy them. Second, we illustrate the overall architecture and main components of AvaTar, including the zero-copy file merging and splitting techniques, which are the key mechanisms.

A. GOALS AND MAIN IDEAS

The primary goal of AvaTar is providing an extremely fast and efficient archiving system. The root cause of a slow archiving is the numerous data copies required in both memory and storage devices. Data copying can be completely eliminated through new kernel-level operations, namely, zero-copy merging and splitting. Instead of copying identical data from the source files to the destination file, the operations relocate the indexes of the data blocks from the source i-nodes to the destination i-node. The manipulation of i-nodes can be achieved extremely quickly in the kernel memory and does not involve any data copies. This also reduces the storage space considerably. The AvaTar file system (AvaTar-FS) is a component of AvaTar that provides the zero-copy operations. We describe such operations in detail when we discuss AvaTar-FS.

Another goal of AvaTar is providing compatibility with existing systems that do not use AvaTar-FS. AvaTar-FS introduces new system calls, that requires modifications to the existing kernel. Despite the several advantages of introducing new system calls, it is impractical to assume that all existing cloud nodes adopt AvaTar-FS. Fortunately, we do not this assumption at least for the extraction. To provide the benefit of a zero-copy extraction even in a system without AvaTar-FS, we define an archived file format that is compatible with a conventional file system image. For example, if AvaTar generates an ISO 9660 image, which is widely used and equipped in several different operating systems (OSs), we can mount the image and easily access the files without data copies. We call this proposed archived file format as an AvaTar image, which we will describe in further detail in following subsection III-C.

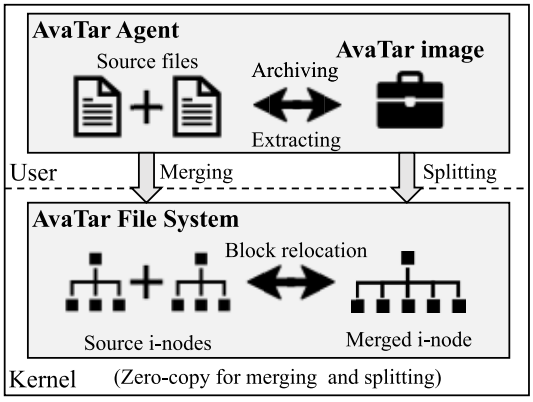


FIGURE 1. AvaTar architecture.

B. ARCHITECTURE

As shown in Figure 1, AvaTar consists of three components, namely AvaTar image, AvaTar file system (AvaTar-FS), and AvaTar agent. AvaTar image is a format for archived files. AvaTar-FS provides the zero-copy operations, and AvaTar agent is a user-level application that directly utilises AvaTar-FS for file archiving and extraction. These components are described in detail in the following subsections.

C. AvaTar IMAGE

AvaTar image is a file format for archival and extraction. A detailed file format is not specified in our design, except

that the format should comply with that of conventional file systems. The benefit of using a mountable image format is that the image file can be integrated as a part of the existing file system, and thus does not inherently require data copies, which is a bottleneck of a user-level splitting operation. The mounting of the file system is simply achieved by reading the file system's metadata into the in-memory data structures of OS kernel. Subsequently, the files in the AvaTar image can be directly accessed through a conventional file system interface. Although mounting a file system requires more system resources than a kernel-level split operation, such resources are still extremely small and can be applied in a shorter time than data copying. Thus, a system without AvaTar-FS can also obtain the benefit of a zero-copy extraction.

In this paper, AvaTar image is formatted as a GNU Tar, which is a widely used file archiving format owing to its simple image formatting and its ability to be mounted as a file system using open-source software such as Ratarmount, Archivemount, and Tarindexer [1], [4], [5]. We only modify the block size of the Tar format from 512 B to 4 KB because 512 B is smaller than a storage block size, and is incompatible with conventional file systems. Note that an AvaTar image can comply with any other file system formats without any modification to the current design because there are no restrictions regarding the AvaTar image format. Moreover, AvaTar image works well even it is stored in not-extent based file system such as Ext2. Because the mount system does not rely on the file system containing the AvaTar image file, the data block management method of file system does not related with the behavior of AvaTar.

D. AvaTar FILE SYSTEM

AvaTar-FS is a kernel-level component providing zero-copy merging and splitting operations. In this study, AvaTar-FS is based on an extent-based file system and implemented on Ext4-FS. We first describe its interfaces and behaviours, followed by an explanation regarding the internal technique used in its operations.[1]

1) MERGE AND SPLIT: NEW FILE OPERATION INTERFACES

The operations are provided through new system calls, i.e. merge() and split(). A merge() operation takes two arguments as input, namely the descriptors of the destination and the source files. Subsequently, the operation merges a source file into the tail of the destination file. Such merging is achieved by relocating the data blocks from the source file to the destination file. To merge several files into a single AvaTar image, a user application should invoke the merge() system call several times. For each merge call, the user puts the merging source file name such that the file is attached after the same destination file. As a result of the merging, the file offset of the destination file is moved to the end of the file, and the source file is removed to protect the file system integrity from multiple allocations of a single block.

The split() operation takes three arguments as input, namely the descriptors of an AvaTar image, a file name to be created, and the length of the split. Initially, an empty file is created with the file name. The AvaTar image is then split into two parts: the first part is from the beginning to the split length, and the second part is the rest of the AvaTar image. The data blocks belonging to the first part are relocated to a new file, and the other data blocks remain in the AvaTar file. As a result of the splitting, the AvaTar file is decreased in size and the offset is moved to the beginning of the file. The size of the created file is set to the third argument, the length of the split. Similar to a merge() operation, the split() operations should be repeatedly called to split every file in the AvaTar image. Note that these two operations do not consider the metadata of the original files for archiving, such as their names, lengths, or timestamps. The metadata should be maintained by the user application applying the operations. We provide an example of a user application for the archiving and extraction, an AvaTar agent, which demonstrates how to use the operations while maintaining the metadata.

2) ZERO-COPY OPERATIONS: ENABLING TECHNIQUE

The enabling technique for a merge() and split() is applying zero-copy operations, which means merging and splitting the files without data copying. To realise zero-copy operations, AvaTar-FS relocates the allocated blocks from the source files to a destination file. When we merge two files, for example, we detach all of the allocated blocks from the i-node of the source file. The blocks are then attached to the i-node of the destination file at the end of the file. Although the concept of the block relocation technique is simple, its implementation depends on the block management scheme of the file system applied. In this paper, we present a block-relocation technique for an extent-based file system, Ext4.

Figure 2(a) shows example block relocations for a merge() operation. A user application first creates the metadata block of file #1, and then calls a merge() operation for merging file #1 to the AvaTar image. AvaTar-FS traverses the extent tree of file #1 and detaches the first extent from the tree. The extent is then attached after the metadata block, which is also an extent. Because the metadata extent and the attaching extent are physically adjacent, they are merged into a single extent. Another metadata block for file #2 is then allocated, and is also merged into the previous extent for the same reason. Finally, the extents of file #2 are relocated to the AvaTar image.

Figure 2(b) shows an example of block relocations for a split() operation. In the figure, the AvaTar image consists of 3 extents and 11 blocks including two metadata blocks. We split the AvaTar image into four files. First, the very first block should be split after the metadata block is read

[1]Our prior work presents the main mechanism of zero-copy operations in [22]. As our prior work was published in a Korean domestic journal, we introduce its detailed mechanism in this section. The prior work provides the prototype of the kernel-level splitting and merging operations only, even its interfaces and behaviors are different from AvaTar-FS. This paper suggests a whole new system to provide the archiving and extraction functionalities based on primitive operations.

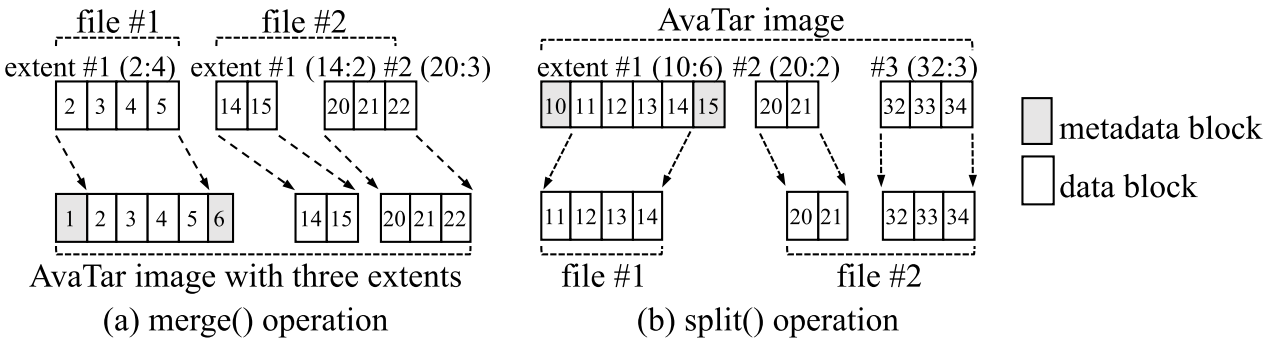


FIGURE 2. Example of block relocations for zero-copy merging (a) and splitting (b).

by an extraction application. The file containing the metadata block is removed immediately after the splitting. The next four blocks are then split and relocated to file #1, which is newly created by AvaTar-FS. In this case, the extent should be divided into two extents, one with four data blocks and another with one remaining block. The user application then reads the metadata for the next file in block #15 and calls a split() to remove the block. The metadata block will be removed as same as the metadata block for file #1. Consequently, the AvaTar image only contains two extents: #2 (20:2) and #3 (32:3), which are data blocks of file #2. Thus, the user application changes the AvaTar image as file #2 by manipulating the metadata.

Based on our experience with the implementation of the technique used in the Ext4 file system, the block relocation technique can be easily implemented in a conventional file system because such a system already has numerous services and functions to manipulate an extent or a list of blocks. Thus, we believe that the merge() and split() operations can be supported by several file systems within an extremely short timeframe if the benefits of such operations are widely accepted.

E. AvaTar AGENT

AvaTar agent is a user-level component that is responsible for the archiving and extraction of an AvaTar image using merge() and split() operations. AvaTar agent has the same usage as the other archive utilities such as GNU Tar. As the only difference between AvaTar agent and other utilities, AvaTar agent uses AvaTar-FS for merge() and split(), whereas the others use a conventional read() and write().

For merging, AvaTar agent takes the target file name and the source file names as input. The agent then creates the metadata block and calls the merge() operation for each file. Regarding the metadata block, AvaTar agent stores the metadata, such as the name, size, timestamps, and access controls in the original file system, which has the same size as the data block. One metadata block is required for a source file. The data blocks of the corresponding source file are then relocated into the AvaTar image using a merge() operation of the AvaTar file system.

By splitting the image file, AvaTar agent can extract the source files from an AvaTar image. Firstly, AvaTar agent reads the metadata block and creates a source file according to the metadata. The agent then removes the metadata block from the AvaTar image using a split() operation. The metadata block is relocated into the temporary file, which the agent immediately removes. Next, the agent calls a split() operation again to relocate the data blocks corresponding to the source file size from the AvaTar image to the created file. As a result, the AvaTar file is decreased in size and the file offset points out the next metadata block. This procedure is repeated to split every file in the AvaTar image.

If a system does not have the AvaTar file system, and the AvaTar agent cannot use a zero-copy split, the splitting can be conducted using a mounting operation. Because an AvaTar image is formatted as a conventional file system image, it can be mounted without additional supports from the OS kernel. As described in Section III-C, the current AvaTar image is formatted as a modified Tar, which is not a standard file system format. Thus, AvaTar agent provides a mount functionality for a modified Tar file based on the file system in user space (FUSE). Because AvaTar agent is a user application, the installation overhead is relatively small. As a consequence, AvaTar can provide the benefit of zero-copy splitting for a system with or without AvaTar-FS.

IV. EVALUATION

In this section, we evaluate the performance of the AvaTar archiving system. For the experiments, we assume that a user composes and extracts an archive file from/to multiple files in the cloud storage.

First, we present the performance of the AvaTar file system, which takes advantage of a zero-copy merge and the split files. Second, comparing a traditional multiple files transfer against AvaTar, we show the practical benefits under the cloud archiving scenario.

A. EVALUATION METHOD

We implemented AvaTar agent and AvaTar-FS on Linux. The current AvaTar-FS runs on Linux kernel 4.4.0. In addition, we implemented the AvaTar mount utility based on a conventional user-level Tar-mount utility [4]. Because the mount utility is based on FUSE [2], it can be executed without an installation or modification of the system.

To evaluate AvaTar, we first generate the source files with random data extracted from */dev/urandom*. AvaTar then generates an AvaTar image, which is an archive file, from which AvaTar extracts the original files. To demonstrate the performance of a zero-copy merge and split, AvaTar agent uses either AvaTar-FS or a normal ext4 file system. Finally, we validate the consistency of the binary data in the target file by comparing the splitting results produced by Tar and AvaTar using the 'diff' utility.

B. PERFORMANCE OF AvaTar

This section presents the performance of AvaTar, comparing it with a traditional archiving solution, GNU Tar, on ext4-FS. To present the performance of AvaTar-FS, we conduct a merging and splitting of the files over a cloud storage. For the cloud storage, we use AWS, which consists of an instance type of t3a.medium for the Seoul region, two vCPUs, and 4 GB of memory. The target storage consists of 10 GB of a general purpose SSD (gp2) volume of Amazon's Elastic Block Store (EBS). During the experiments, the virtual machine (VM) was sustained in I/O burst mode, providing a maximum bandwidth of 3,000 IOPS or a rate of 250 MB/s. After 30 min of burst mode, the maximum I/O bandwidth was limited to 1/30 that of burst mode.[2] To obtain a consistent performance, we renewed the volume after a single run of all experiments such that the VM continued running in burst mode. For all experiments, we measured the processing time for five different runs and herein present the average only. The variations were sufficiently small to ignore.

We present the performance of archival and extraction with AvaTar, varying the number of files and their size. The labels in the graph, 'Tar: archive' and 'Tar: extract' indicate the performance results of archiving several source files and extraction an archive file using GNU Tar, respectively. Similarly, 'AvaTar: archive' and 'AvaTar: extract' indicate the archiving and extraction performances when using AvaTar. 'AvaTar: mount' indicates the results using the AvaTar-mount utility applied for an extraction of the AvaTar-image instead of an extraction with AvaTar.

In addition, we present the impact of a page cache, which is generally used to mitigate the slow file I/O performance when accessing a physical storage device. We measure the time of the archiving and extraction operations for two cases: when the target files are not cached and when they are. Usually, we cannot expect that all source files for archiving will be cached, or that the transferred archived file will be entirely cached owing to its large size.

1) UNCACHED ARCHIVING AND EXTRACTION

First, Figure 3a shows the performance results of non-cached file archiving for various file sizes, achieving a file archiving for 1,024 files with a change to the file size from 64 to 4.096 MB. In the figure, we can see that AvaTar shows a much shorter and consistent processing time than that of Tar. Both AvaTar and Tar show similar performance results for small sized files of 64 MB. However, the performance of Tar seriously decreases. Tar takes 63.71 s for extracting an archive file of 4,096 MB. By contrast, AvaTar takes only 0.48 s for extraction, which is only 0.75 % the processing time of Tar. Note that the performance of Tar degrades with the file size; however, AvaTar shows a consistent processing time of approximately 0.49 s for extraction and 1.84 s for archiving, regardless of the file size. AvaTar-mount also shows an extremely low but consistent performance, with an average speed of 1.03 s, but is twice that of the extraction using AvaTar. We believe that AvaTar-mount is still significantly useful when AvaTar-FS cannot be applied because its mounting is still much faster than extraction using Tar.

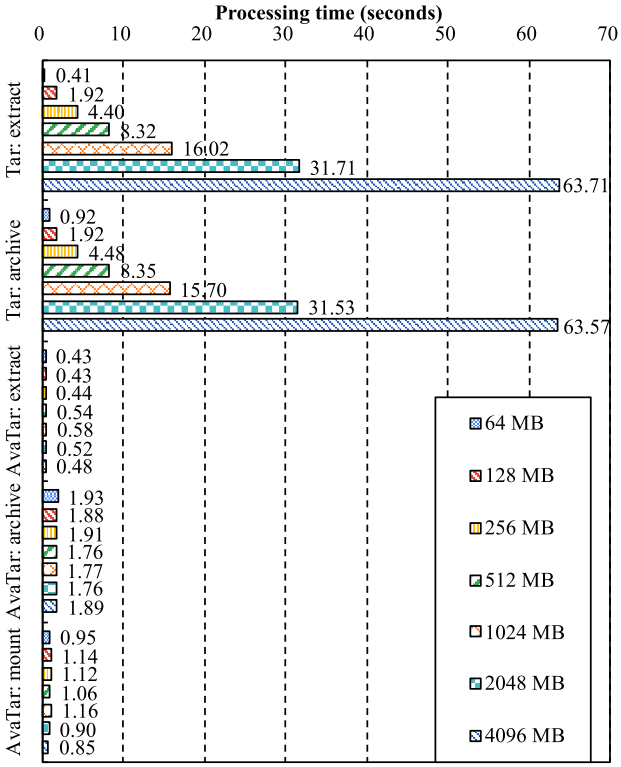
Next, Figure 3b shows the performance results of non-cached file archiving for fixed-size data. We conducted file archiving operations for 1 GB of data when changing the number of files from 64 to 4,096. In the figure, AvaTar consistently shows a better performance than Tar. In this experiment, Tar shows a relatively consistent processing time of approximately 16 s. However, the processing time of AvaTar increases with the number of files because AvaTar should manipulate the metadata of each file, and the number of system calls for zero-copy merging and splitting also increases. However, the worst result with AvaTar was 6.54 s in the case of archiving 4,096 files, which is only 39.49% of the result from Tar archiving. In the graph, the performance gap is at maximum 16.13 s when archiving 64 files. In this case, AvaTar consumes only 0.3 % of the Tar processing time. The results of AvaTar-mount show a similar trend as the results of AvaTar, but are slightly slower than the extraction with AvaTar.

2) CACHED ARCHIVING AND EXTRACTION

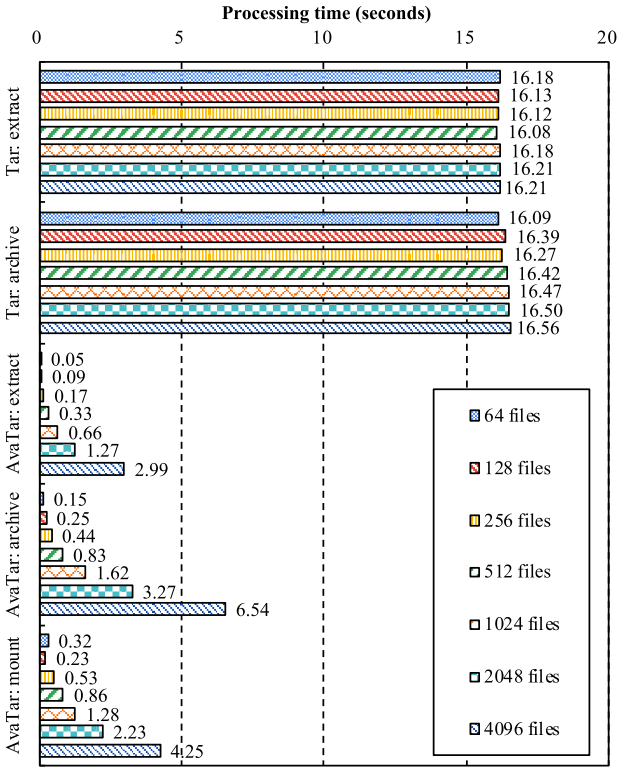
To present the performance of AvaTar and Tar using memory-cached data, we warm up the page cache by reading files before supplying the actual archiving and extraction workload. We then conduct the same experiments using the non-cached configurations. Figure 4 shows the performance results of cached file archiving for a) a fixed number of files and b) a fixed data size. When we compare Figures 4a(a) and 3b(a), Tar completes the archiving and extraction twice as fast when the data are cached except for data sizes of 2 and 4 GB. Because the size of the system memory is 4 GB, the 2 and 4 GB data cannot be fully cached. Note that Tar requires double the size of the cache memory because the data will be copied for writing to a new file. Thus, Tar incurs a cache wiping or pollution problem with necessarily duplicated data. By contrast, AvaTar show similar results regardless of the cached data because AvaTar does not deal with the data.

In Figure 3b(b), the results of Tar are exactly half the results shown in Figure 4a(b) because the data sizes are all 1 GB, and the data can be fully loaded into the 4 GB system memory. AvaTar still shows similar results as in the previous experiment. In our configurations, however, AvaTar

[2]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html



(a) Uncached file archiving performance for 1,024 files.



(b) Uncached file archiving performance for 1 GB of data.

FIGURE 3. Performance results of uncached file archiving.

is still faster than Tar for all cases applied. AvaTar extracts the 64 files within 0.043 s which is only 0.52% of the result of Tar. For the worst case, AvaTar archives the 4,096 files in 6.46 s, which is 75.39% the result of Tar.

An interesting result occurred with AvaTar-mount. The mount is achieved within 0.48 s for 4,096 files, which is 16.32% of the AvaTar extraction. However, we cannot conclude that AvaTar-mount will be more helpful than AvaTar-extraction because the data are not fully cached when the data size is extremely large or when the other workloads continuously compete for the cache.

TABLE 1. Memory footprint for 1,024 file archiving in MB.

Data size (MB)	64	128	256	512	1024	2048	4096
Tar-extract	136	265	524	1045	2085	3696	3693
Tar-archive	134	264	523	1045	2085	3695	3687
AvaTar-extract	23	23	23	27	29	25	29
AvaTar-archive	5	5	5	5	5	5	9

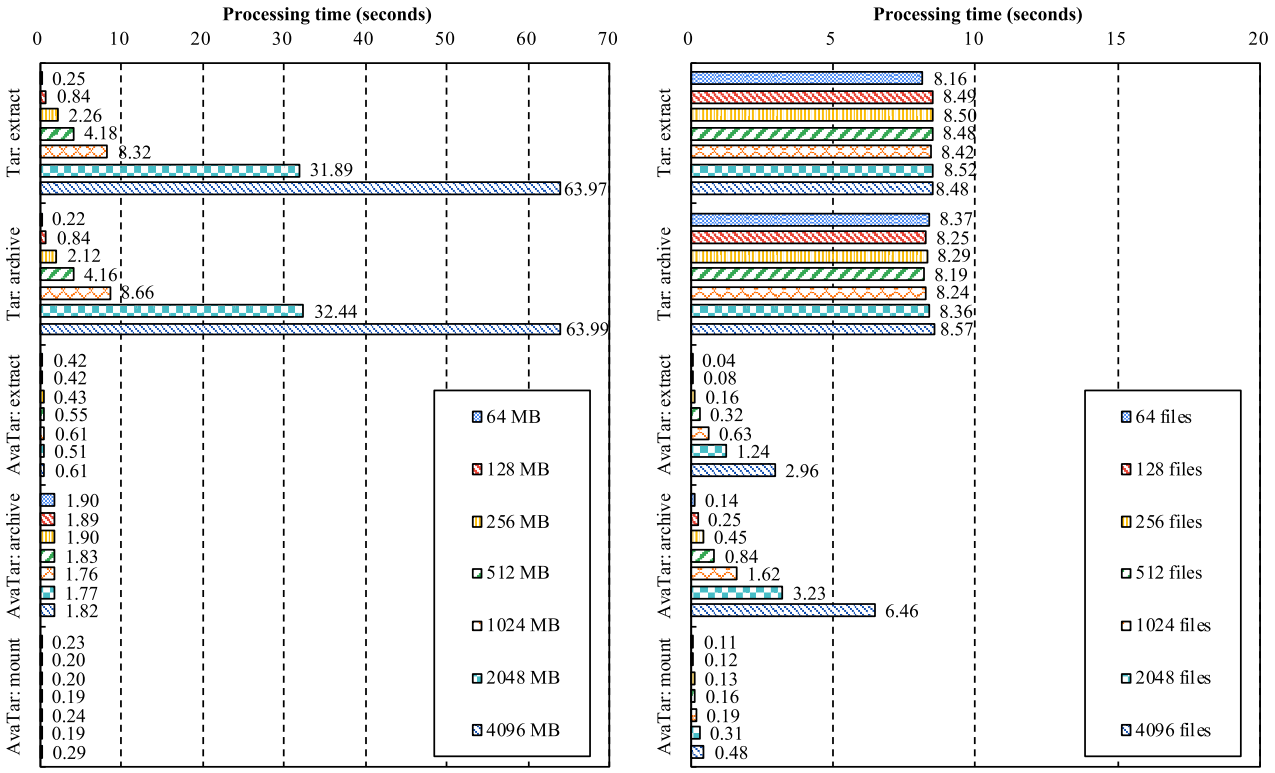
Table 1 compares the memory footprint when we use AvaTar and Tar. The measured footprint obtained when we conduct the archiving and extraction of 1,024 non-cached files while varying the size of the total working set. As the result indicate, the memory footprint of AvaTar is extremely small and almost constant. Owing to the zero-copy merge and split operations, AvaTar-FS only touches the metadata, instead of caching the user data.

By contrast, Tar consumes as much memory as the data size utilised. As the working set increases, Tar consumes more memory bandwidth and page cache; thus, memory-intensive cloud applications and services may be negatively affected. Concisely, AvaTar is more friendly to other cloud services because it minimises the page cache pollution.

In summary, the first experiments demonstrate that AvaTar archiving with zero copy operations provides a much better performance than a traditional user-level archiving system as expected. In our experiments, the best archiving time with AvaTar is only 0.48 s, whereas traditional archiving with Tar is 63.71 s. That shows AvaTar is 132 times faster than the traditional archiving system. In addition, the results show that the performance of AvaTar is not related with the data size, whereas the performance of a user-level archiving system is dependent upon this size. Although the performance of AvaTar is correlated with the number of files, AvaTar provides a much better performance than the user-level archiving system when dealing with a large number of files.

C. UPLOADING TO CLOUD STORAGE

In this section, we consider a practical use case of AvaTar, namely uploading data to cloud storage. Cloud storage is a popular service that provides an object-based storage, such as Amazon S3, Google Drive, OneDrive, or Dropbox. Such a service provides not only file storage but also several



(a) Cached file archiving performance for 1,024 files.

(b) Cached file archiving performance for 1 GB of data.

FIGURE 4. Performance results of cached file archiving.

functionalities for the file objects, such as sharing over the Internet, versioning, indexing, and searching. Users upload their files onto the cloud storage, and can access the files whenever and wherever they want. The upload client uploads the requested files one by one using the HTTP PUT method. The uploading with AvaTar is more efficient because AvaTar agent archives the source files into a single file, and then sends the file under the maximum network bandwidth. After the transmission, AvaTar agent should extract the file, and then register the extracted files using the PUT method locally.

To demonstrate the efficiency of AvaTar, we conducted a practical experiment under four uploading scenarios: uploading with a traditional method, uploading a Tar-archived file, uploading an AvaTar image, and uploading an AvaTar image onto a server without AvaTar-FS. In the last scenario, the AvaTar image should be mounted using AvaTar agent, which is compatible with another file system. For cloud storage, we generate an AWS EC2 instance using a t3a.medium type within the Seoul region. We then build the cloud storage service on the instance using ownCloud, which is an open-source based object storage system that provides a commercial-level performance [3]. For the client, we use an on-premise system located in Jeonju, which is approximately 200 km from the AWS Seoul region. We sent files from the client system to the ownCloud server. Thus, our experimental environment reflects real traffic over the Internet. To present the archiving performance for small files, we fix the total amount of data to transmit to 1,024 MB, and observe the performance by changing the number of files from 64 to 4,096.

The detailed procedures of the experiments are as follows: 1) A traditional upload: We upload each file to the cloud storage server through the HTTP PUT method. To simply implement the eventual synchronisation protocol in cloud storage, we sequentially send the target files one by one. 2) Tar-archived upload: The files are archived using Tar in a client system, and the archived file is sent to the ownCloud server through the secure copy protocol (SCP), which is a TCP-based file transfer protocol.[3] We then extract the file using Tar on the server, and register the files using HTTP PUT. 3) AvaTar-image upload: This is similar to a Tar-archived upload but uses AvaTar instead of Tar. 4) AvaTar-image upload and mount: This is the same as an AvaTar image upload, but only uses the AvaTar mount utility, instead of a file extraction with AvaTar.

The overall results are shown in Figure 5, which indicates the elapsed time for each upload scenario for cloud storage. For every archiving scenario, the size of the archived file is approximately 1,024 MB and the sending of the file through SCP takes 23.88 s. As shown in Figure 5, the shortest

[3]SCP is outdated and the rsync and sftp are recommended as its alternatives (https://www.openssh.com/txt/release-8.0). We have used SCP because it is well-known and easy-to-use. Note that any other method or protocol can be used for transferring AvaTar image.

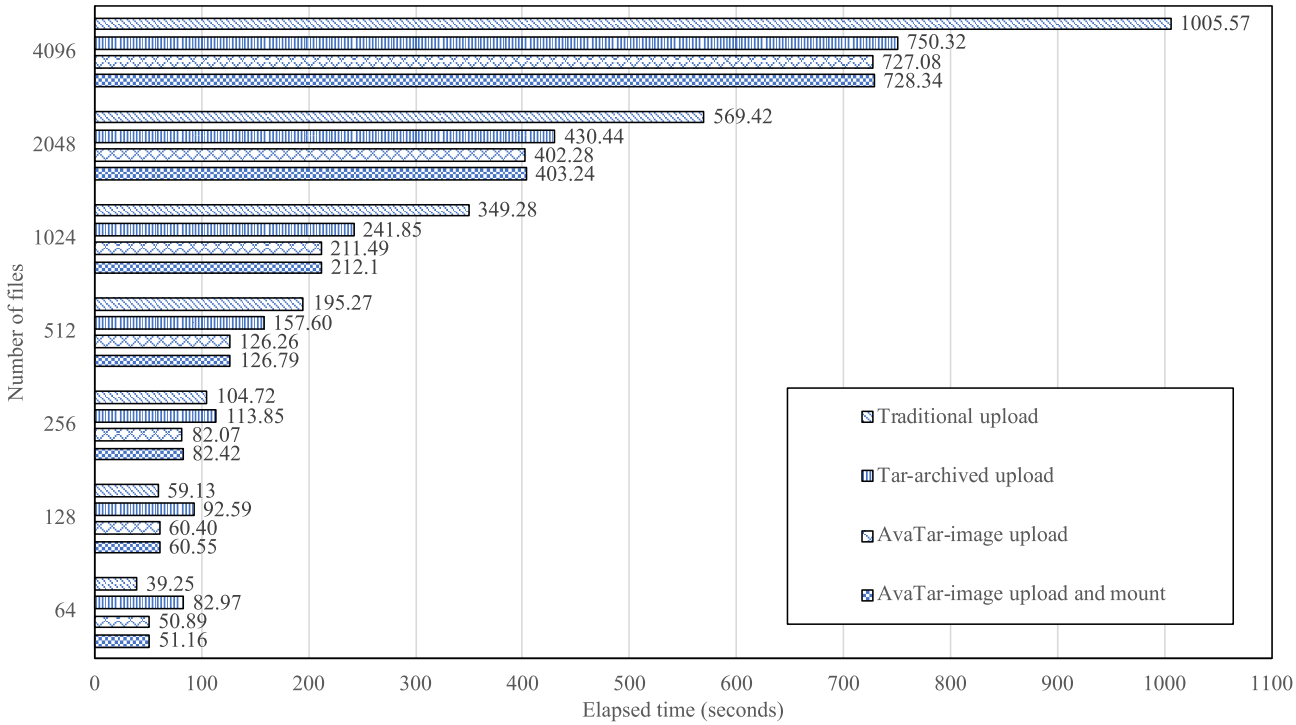


FIGURE 5. Performance results of four uploading scenarios.

uploading time is 39.25 s when we upload 64 files using a traditional upload. In this case, the size of each file is 16 MB, which is sufficiently large to fully utilise the network bandwidth. Although not presented in the graph, the registration of 64 files, after the reception of the files, takes 26.82 s. When uploading 128 files, as shown in Figure 5, the traditional upload takes 59.13 s, which is slightly shorter than that of an AvaTar-image upload. However, Tar-archived uploading takes 92.59 s, which is approximately 56% longer than a traditional upload.

Figure 5 shows the clear benefit of using AvaTar as the number of files increases. For 256 files, as shown in the figure, the AvaTar-image upload takes 82.07 s, which is 78.37% that of a traditional upload, at 104.72 s. At the same time, the Tar-archived upload is still slower than a traditional upload by approximately 8.72%.

As shown in Figure 5, an AvaTar-image upload achieves the best performance when the number of files is larger than 256. In the case of 1,024 files, the result with AvaTar is 211.49 s, which is only 60.55% of the result of a traditional upload at 349.28 s. The result is impressive because it includes all Internet traffic, and AvaTar achieves 1.65 times faster performance than traditional cloud files transfer.

Even with the AvaTar-mount utility, the result was 60.73% that of a traditional upload. In this case, the extraction and mounting of an AvaTar-image takes 0.66 and 1.28 s, respectively. Although the Tar-archived upload is also faster than that of the traditional method, the result of the AvaTar-image upload is 12.55% faster than the that of the former.

We observed similar results when uploading 2,048 and 4,096 files, in which only the registration time is increased. By contrast, the traditional upload shows that the uploading time increases with the number of files owing to an inefficient use of network bandwidth.

One reason for the excellent performance of an AvaTar upload is that AvaTar mitigates the inefficiency of traditional HTTP-based file transfer methods. In a traditional file transfer, every file transfer uses a separate TCP session. By contrast, AvaTar makes a single session, and transfers all files as an archive, maximising the network bandwidth.

The results of this experiment can be summarised as follows: 1) sending a large archived file is better than sending several small files, as expected; 2) AvaTar decreases the archiving and extraction times significantly, and thus its overall result is better than that of the existing user-level archive system; and 3) even when the new AvaTar-FS is unavailable, the benefit of AvaTar still remains with the user-level AvaTar-mount utility.

It should be noted that our contribution can be increased if an AvaTar-image upload is implemented inside the cloud storage system. In our experiment, the final registration procedure in the cloud storage server copies the extracted files from their original path to the data path of the cloud storage server, which is an unnecessary data copying. If we can remove the data copying from the cloud storage server, the benefit of AvaTar will become even clearer than in the experimental results presented herein.

V. DISCUSSION

A. CONSIDERATION FOR COMPRESSION

One considerable design choice is the compression. AvaTar does not include a component explicitly for compression because compression is controversial in terms of its gain and loss. Compression requires additional reading and writing for the storage device, particularly in the case of a large file that cannot be stored in the main memory. It also requires a large amount of computational power, and cannot guarantee a reduction of the data size for multimedia files that have already been highly compressed. Thus, compression remains a choice of users because there is no restriction on the compression for an AvaTar image, which is simply a regular file. Online compression techniques that compress the data stream for a transfer through a network are already provided, and such techniques can be used on AvaTar without any conflict [9], [15].

B. CONSIDERATION FOR ORIGINAL FILES

The other issue is how we manage the original files after merge() and split() operation. As the extents are moved to AvaTar image from the original files, the original files will become unavailable. To prevent this, we can just copy the extents instead of moving them. As a result, original files and AvaTar image share the data blocks. Such multiple allocations of data blocks are not allowed in conventional file systems. However, the concept that several files share the data blocks are already used by the hard link. If we provide the proper policy to manage the sharing of data blocks in AvaTar FS, the original files and AvaTar image can exist together without any problem. Because it requires a more intensive discussion about the related file system semantics and behaviors, we do not address the issue in this paper.

VI. CONCLUSION

This paper presented AvaTar, a novel kernel-level archiving system. Using zero-copy merging and splitting primitives, AvaTar significantly improves the performance of archiving and extraction. We implemented AvaTar on a recent version of Linux, and evaluated it against the conventional archiving tool GNU Tar. Further, we presented the practical performance of AvaTar when applied to the commercial cloud storage service AWS. Our results show that AvaTar accelerates the performance of traditional file archiving by up to 132 times, and shortens the file upload time to commercial cloud storage by up to 60.55%, when compared with a traditional method. In addition, the memory footprint of AvaTar is extremely small and mostly constant regardless of the archiving data size. In addition to performance improvements, AvaTar also supports compatible file archiving using AvaTar images, which are easily mountable on conventional file systems. We believe that AvaTar is beneficial for both cloud providers and users; cloud providers can reduce their overall provisioning costs, and cloud users can receive an improved quality of service.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers of the IEEE Access for their valuable comments and suggestions for improving the quality of this article.

REFERENCES

[1] *Archive Mount: A Fuse Filesystem For Mounting Archives in Formats Supported by Libarchive*. Accessed: Mar. 24, 2020. [Online]. Available: https://github.com/cybernoid/archivemount/
[2] *Linux Fuse (Filesystem in Userspace)*. Accessed: Mar. 24, 2020. [Online]. Available: https://github.com/libfuse/libfuse/
[3] *Owncloud: Open-Source Personal Cloud Collaboration Platform*. Accessed: Mar. 24, 2020. [Online]. Available: https://owncloud.org/
[4] *Ratar: Random Access Read-Only Tar Mount*. Accessed: Mar. 24, 2020. [Online]. Available: https://github.com/mxmlnkn/ratarmount/
[5] *Tarindexer: Python Module For Indexing Tar Files For Fast Access*. Accessed: Mar. 24, 2020. [Online]. Available: https://github.com/devsnd/tarindexer
[6] I. Bermudez, S. Traverso, M. Mellia, and M. Munafo, "Exploring the cloud from passive measurements: The Amazon AWS case," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 230–234.
[7] R. Card, "Design and implementation of the second extended filesystem," in *Proc. 1st Dutch Int. Symp. Linux*, 1995.
[8] Y. Diao, D. Ganesan, G. Mathur, and P. J. Shenoy, "Rethinking data management for storage-centric sensor networks," in *Proc. CIDR*, vol. 7, 2007, pp. 22–31.
[9] F. Fusco, M. P. Stoecklin, and M. Vlachos, "NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1382–1393, Sep. 2010.
[10] P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating System Concepts*. Hoboken, NJ, USA: Wiley, 2018.
[11] T. F. Gosnell, "Data archiving system," U.S. Patent 7 801 871 B2, Sep. 21 2010.
[12] S. W. Jung, S. Young Ko, Y. J. Nam, and D.-W. Seo, "Block link file system supporting fast editing/writing for large-sized multimedia files in multimedia devices," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2012, pp. 457–458.
[13] Y. Kim, Y. Woo, H. Lee, and E. Seo, "Design and implementation of split/merge operations for efficient multimedia file manipulation," *Comput. Standards Interfaces*, vol. 48, pp. 80–89, Nov. 2016.
[14] V. Kobla, D. S. Doermann, and K.-I. Lin, "Archiving, indexing, and retrieval of video in the compressed domain," *Proc. SPIE*, vol. 2916, pp. 78–89, Nov. 1996.
[15] C. Krintz and S. Sucu, "Adaptive on-the-fly compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 1, pp. 15–24, Jan. 2006.
[16] G. Lee, H. Ko, S. Pack, V. Pacifici, and G. Dan, "Fog-assisted aggregated synchronization scheme for mobile cloud storage applications," *IEEE Access*, vol. 7, pp. 56852–56863, 2019.
[17] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "Understanding and surpassing dropbox: Efficient incremental synchronization in cloud storage services," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2014, pp. 1–7.
[18] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new Ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, vol. 2, 2007, pp. 21–33.
[19] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Comput. Syst. (TOCS)*, vol. 2, no. 3, pp. 181–197, Aug. 1984.
[20] *NTFS Technical Reference: How NTFS Works*, Microsoft, Washington, DC, USA, 2009.
[21] R. S. Ohran, "Method and system for mirroring and archiving mass storage," U.S. Patent 6 397 307 B2, May 28 2002.
[22] H. Park, J.-H. Jang, and J. Lee, "Design and implementation of Kernel-level split and merge operations for efficient file transfer in cyber-physical system," *IEMEK J. Embedded Syst. Appl.*, vol. 14, no. 5, pp. 249–258, Oct. 2019.
[23] V. Persico, A. Montieri, and A. Pescape, "On the network performance of Amazon S3 cloud-storage service," in *Proc. 5th IEEE Int. Conf. Cloud Netw. (Cloudnet)*, Oct. 2016, pp. 113–118.
[24] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, Aug. 2013.

[25] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. USENIX Annu. Tech. Conf.*, vol. 15, 1996, pp. 1–15.
[26] S. Tweedie, "Ext3, journaling filesystem," in *Proc. Ottawa Linux Symp.*, 2000, pp. 24–29.
[27] J.-M. Vau, J. Moelle, O. Furon, and O. Rigault, "Method for archiving multimedia messages," U.S. Patent 10 528 981 B1, Jul. 27 2006.
[28] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan, "DEBAR: A scalable high-performance de-duplication storage system for backup and archiving," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, 2010, pp. 1–12.
[29] Y. Yoo, S. Sopharath, Y. Woo, J. Kim, and Y. Ko, "Design and implementation of the metadata modification concept minimizing file modification," in *Proc. TENCON-IEEE Region 10th Conf.*, Oct. 2018, pp. 1499–1503.

HYUNCHAN PARK received the B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, South Korea. He was a Research Professor with Korea University, from 2014 to 2016. He is currently an Assistant Professor with the Division of Computer Science and Engineering, Jeonbuk National University. His current research interests include storage systems, particularly with flash-based devices, as well as virtualization and operating systems.

YOUNGPIL KIM (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University, Seoul, South Korea, in 2002, 2004, and 2015, respectively. He was appointed as a Research Professor with the College of Informatics, Korea University. In 2019, he joined the School of Computer Science, Semyung University, Jecheon, South Korea, where he is currently an Assistant Professor. His research interests include system reliability, the kernel architecture used in operating systems, cloud computing, and system-level security.

SEEHWAN YOO (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, South Korea, in 2002, 2004, and 2013, respectively. From 2013 to 2014, he worked with the Software Platform Lab, LG Electronics. He is currently an Associate Professor with the Department of Mobile Systems Engineering, Dankook University, Yongin, South Korea. His current research interests include cloud computing systems, and system security for mobile applications and services.