IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 September 2022, accepted 4 November 2022, date of publication 18 November 2022,
date of current version 28 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3223107

RESEARCH ARTICLE

IPFSz: An Efficient Data Compression Scheme in InterPlanetary File System

MANSUB SONG[1], JONGBEEN HAN[1], HYEONSANG EOM[1], AND YONGSEOK SON[2]
[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea
[2]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, Republic of Korea

Corresponding author: Hyeonsang Eom (hseom@cse.snu.ac.kr)

This work was supported in part by the Brain Korea 21 (BK21) FOUR Intelligence Computing [(Department of Computer Science and Engineering, Seoul National University (SNU)] funded by the National Research Foundation of Korea (NRF) under Grant 4199990214639, in part by the National Research Foundation of Korea (Optimizing GPGPU scheduling and processing in multi-GPU environments based on data locality) under Grant NRF-2021R1F1A1063438, in part by Reappay Payments Corporation, in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government through MSIT under Grant NRF-2021R1C1C1010861 and Grant NRF-2022R1A4A5034130, and in part by the Korea Institute for Advancement of Technology (KIAT) Grant funded by the Korean Government through MOTIE under Grant KIAT-P0012724.

ABSTRACT InterPlanetary File System (IPFS) is a peer-to-peer (P2P) distributed file system that allows data to be shared and exchanged between connected nodes. Recently, IPFS aims to replace existing protocols used in a centralized system (e.g., HTTP) by overcoming their weaknesses such as a single point of failure and arbitrary control. However, we observe that the performance of IPFS is worse as the data size increases since it demands large numbers of storage and network I/O operations. In this paper, we present an efficient data transmission scheme in IPFS called IPFSz to improve the I/O performance of IPFS. To do this, we extend IPFS by enabling real-time compression functionality while maintaining its existing operations in IPFS. In IPFSz, we compress/decompress the data during I/O operations by using a compression algorithm, manage the states of data (e.g., compressed or decompressed), and provide a compression/decompression interface to applications. Thus, IPFSz can reduce the number of storage and network I/O operations and storage space without sacrificing the integrity of the existing IPFS. We have implemented and evaluated IPFSz on five Amazon EC2 nodes. The experimental results show that IPFSz provides higher performance by up to 7.9× and saves the storage space by up to 6.3× compared with existing IPFS.

INDEX TERMS InterPlanetary file system, data compression, decentralized system, P2P network, distributed file system.

I. INTRODUCTION

InterPlanetary File System (IPFS) is a peer-to-peer (P2P) distributed file system for storing and sharing data in a global namespace connecting all computing devices by using content-addressing to uniquely identify each file [1], [2], [3], [4]. IPFS aims to replace existing protocols in a centralized system by overcoming their weaknesses such as a single point of failure and arbitrary control by a service provider. For example, HTTP is mainly used in a centralized system by the communication between clients and a central server. This centralized system with HTTP provides an efficient file access control by managing the files in a single server. However, it can be shut down by an attack to a single server or controlled by an arbitrary service provider. To handle these issues, IPFS provides a decentralized system by the communication between nodes in a P2P network. Every peer in the IPFS network exchanges data directly by P2P communication, which does not require a single server. In other words, each peer plays roles as both client and server so that the IPFS network has a much lower probability of incurring single point of failures and arbitrary control.

Although IPFS alleviates the issues of the centralized systems, there can be performance issues in IPFS when the data size is large [5], [6]. For example, when the data size is small (i.e., kilobytes), the performance of IPFS is similar to that

The associate editor coordinating the review of this manuscript and approving it for publication was Derek Abbott.

of other communication protocols (e.g., HTTP and FTP), however, the performance of IPFS is greatly reduced when the data size is large (i.e., megabytes). One of the main reasons is that the large data size makes directed acyclic graph (DAG) which manages the data in IPFS complex and deeper. As a result, the data access time using DAG increases as the data size increases.

In previous studies [3], [4], the P2P distributed file systems and IPFS were investigated to handle this performance issue. Chen et al. [4] provided a zigzag-based storage model to improve the performance of block storage in IPFS by combining three replication schemes and an erasure codes storage scheme. Le et al. [3] improved the performance of IPFS by using key-value memory caching which reduces the time of finding a data provider and makes it faster to retrieve data from IPFS. Our study is in line with these studies in terms of improving the performance in the P2P distributed file systems and IPFS. In contrast, we focus on reducing the number of storage and network I/O operations by using a data compression scheme.

In this paper, we propose an efficient data transmission scheme in IPFS, called IPFSz, for improving I/O performance. To do this, we extend IPFS by enabling real-time compression functionality while maintaining the fundamental features in IPFS such as decentralization and P2P communication. In IPFSz, we first compress and decompress data during I/O operations by using a compression algorithm. For example, we compress the data and write the compressed data in storage when uploading, while we decompress the compressed data into the original data and transfer the data to applications when downloading. Second, we manage states of data (e.g., compressed or decompressed) by using the metadata (i.e., content identifier) per data. For example, we use an unreserved area of the content identifier as a compression flag. Thus, according to the compression flag, we detect or decide whether the data is compressed or needs to be decompressed. Finally, we provide a compression interface to applications. This interface enables to use of the compression functionality with minor modifications to applications. As a result, IPFSz can reduce end-to-end response time by decreasing storage/network I/O operations and storage space.

We have implemented and evaluated IPFSz on five Amazon EC2 nodes. We have applied and evaluated various compression algorithms (e.g., Gzip, LZ4, LZO, Snappy, and Zstandard) with IPFSz to show which algorithm is most efficient in terms of performance and storage space. The experimental results show that IPFSz provides higher performance by up to 2.4× and 7.9× in the case of upload and download operations, respectively, and saves the storage space by up to 6.3× compared with existing IPFS.

The contributions of our work are as follows:

- We have observed the uploading and downloading performance bottleneck in the existing IPFS according to data size.
- We have designed and implemented an efficient data transmission scheme in IPFS, called IPFSz, to improve the I/O performance while maintaining the existing operations of IPFS.
- We have applied various compression algorithms to show which algorithm is most efficient in IPFSz.
- The experimental results show that IPFSz shows higher uploading/downloading performance by up to 7.9× and saves the storage space by up to 6.3× compared with existing IPFS.

The rest of this paper is organized as follows: Section II describes the background and motivation. Section III presents the design and implementation of IPFSz. Section IV shows the experimental results. Section V discusses related work. Section VII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. AN OVERVIEW OF EXISTING IPFS

IPFS is a distributed P2P file system for retrieving and sharing data. Unlike the location addressing used by HTTP, IPFS provides content addressing by a content identifier called CID to provide data integrity and trusted P2P communication. The CID has a multi-hash structure which consists of the type of hash function (SHA2, SHA3, etc.), the hash length, and the hash value of the data contents itself. When storing data, IPFS performs the hash function based on the data contents and stores their hash values in CID. By matching the CID of the requested data with the CID of transmitted data, IPFS provides data integrity and trusted communication. In terms of uploading and downloading operations in IPFS, the internal processing of operations is as follows.

1) UPLOADING

When an application uploads data, IPFS divides the data into small chunks (e.g., 256KiB per chunk) for efficient P2P transmission. After then, IPFS creates the content identifier (CID) per chunk and comprises a Merkle directed acyclic graph (Merkle DAG) by using chunks and their CIDs. Each node in Merkle DAG creates a block including a chunk and its CID. Then, IPFS creates a root CID to represent the blocks. After then, IPFS stores the block in local IPFS storage. Finally, IPFS updates and notifies the location information of the uploaded data to all the nodes connected in the P2P network. As mentioned above, during this I/O process, Merkle DAG manages the blocks in IPFS. However, if the uploaded data size is large, the number of blocks and metadata managed by Merkle DAG increases. As a result, the large-sized data increases the number of I/O operations and decreases the I/O performance [5], [6]. Thus, it is important to reduce the number of I/O operations by reducing the data size for achieving higher I/O performance.

2) DOWNLOADING

When an application downloads data, IPFS searches the data in its local storage. If the data is in the local storage, IPFS

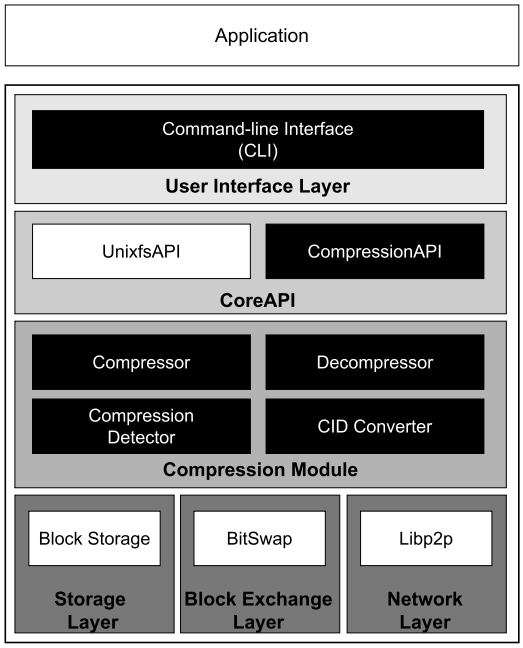


FIGURE 1. Overall architecture of IPFSz.

transfers the data to the application. Otherwise, IPFS searches the remote nodes which store the blocks of the data and receives blocks from the remote nodes. After then, IPFS creates Merkle DAG and reconstructs the data using transmitted blocks and transfers the whole data to the application. As mentioned above, during this communication process, IPFS in the local node receives blocks from remote nodes. However, if the downloaded data size is large, the number of blocks to be transferred increases. As a result, the large-sized data increases the number of network I/O operations and traffic. In addition, similar to the upload operation, the number of storage I/O operations increases because the number of blocks and metadata managed by Merkle DAG increases.

During uploading and downloading data, UnixFS [7] is used to create and manage the Merkle DAGs. UnixFS is a protocol-buffers [8] based format, a free and open-source cross-platform data format used to serialize structured data, for describing files, directories, and metadata in IPFS. For example, it has data type (e.g., file, directory, metadata), the content of data, data size, and hash type (e.g., SHA2-256) to represent data recorded in UnixFS.

B. DATA COMPRESSION ALGORITHMS

Data compression is used to reduce the size of data by eliminating certain redundant data. One of the types of data compression is lossless compression allows the original data to be perfectly reconstructed from the compressed data. Among the lossless data compression algorithms, there are high compression ratio algorithms (e.g., Gzip) and high-speed compression algorithms (e.g., LZ4, LZO, Snappy, and Zstandard). Gzip [9] is used on large scale for lossless data compression. It compresses data to create a smaller and less demanding representation for storing and transferring data. Although Gzip has

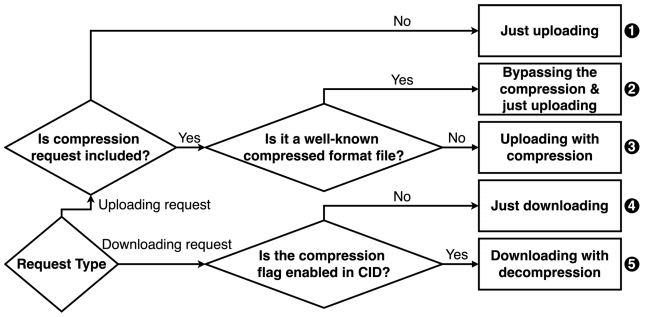


FIGURE 2. Process overview on IPFSz.

a slow compression/decompression speed, it provides a high compression rate.

Meanwhile, LZ4 [10], LZO [11], Snappy [12], and Zstandard [13] are representative lossless compression that focuses on compression speed rather than compression ratio. LZ4 is a byte-oriented compression scheme and aims to provide a good trade-off between speed and compression ratio. LZO supports overlapping compression and in-place decompression for highly redundant data. LZ4 and LZO generally have almost the same level of compression ratio and compression speed, but LZ4 has significantly faster decompression performance than LZO. Snappy is developed by Google and compresses/decompresses petabytes of data in Google's production environment. It aims for very high speeds and reasonable compression. Zstandard is developed by Facebook and combines a dictionary-matching stage with a large search window and a fast entropy coding stage. Zstandard provides a compression ratio comparable to Gzip while it also provides much faster performance for compression and decompression [14], [15]. In our scheme, we use these lossless algorithms since IPFS should guarantee data integrity and transfer data completely to applications. In addition, in this study, we perform a comparative study of which algorithms are most efficient in terms of performance and storage space.

III. DESIGN AND IMPLEMENTATION

Our primary goal is that the compression effectiveness in a p2p distributed system such as IPFS can provide higher uploading and downloading performance without depending on compression from network protocols. In this section, we present the design and implementation of IPFSz which is a variant of IPFS enabling data compression functionality to reduce the number of storage and network I/O operations and storage space.

A. OVERALL ARCHITECTURE

Figure 1 describes the overall architecture of IPFSz. White boxes indicate existing components in the existing IPFS, and black boxes indicate components that we have modified or created for our scheme. As shown in the figure, a client retrieves and stores the data by executing IPFS commands (e.g., downloading and uploading) via the command-line interface (CLI) in the client interface layer. We modified the

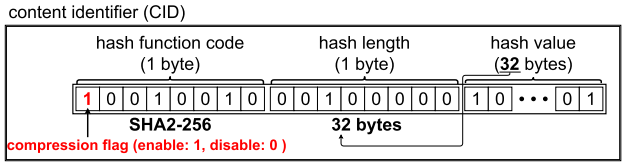


FIGURE 3. Managing compressed or decompressed data using content identifier (CID) in IPFSz.

TABLE 1. Hash function number used in IPFS.

Hash function name	Hash function code	Binary
sha1	17	0001 0001
sha2-256	18	0001 0010
sha3-512	19	0001 0011
...	...	...
keccack-512	29	0001 1101

CLI to allow the client to request the data compression. For example, the client can use an additional command option (e.g., *–compression*). With this option, the CLI detects the data compression request and calls the CompressionAPI in the CoreAPI [16]. In addition, instead of CLI, an application can directly use the CompressionAPI to utilize the compression functionality.

CoreAPI provides the core and control commands in IPFS. Also, coreAPI consists of UnixFS API and compression API. For example, UnixFS API in the coreAPI provides the commands to upload, download, and inquiry data (e.g., file, directory). We added compression API to coreAPI to provide compress and decompress the data.

Our proposed compression module has four sub-modules (i.e., compressor, decompressor, compression detector, and CID converter) for compressing or decompressing requested data. When an application requests to compress data (e.g., uploading), the compression detector identifies whether the operation is compression or decompression. The compressor compresses data using a compression algorithm to reduce the data size. In this case, the CID converter updates and sets a compression flag in CID. Meanwhile, the decompressor decompresses the compressed data when the compression flag is already set to transfer the original data to the application. In this case, the CID converter updates and unsets the flag. Note that since already compressed files have little compression effect, the compression detector also determines whether to compress or not by checking the well-known compressed file format such as zip, jpeg, mp3, mp4.

Our compression module cooperates with the storage layer, block exchange layer, and network layer in IPFS to store and transfer the data. In the storage layer, the block storage stores the uploaded and downloaded data and their DAG metadata to local storage. In the block exchange layer, the bitswap is the module to request and send blocks to other peers in the P2P network. In the network layer, the libp2p is a modular system of libraries that enable the development of peer-to-peer network applications such as transport, security, peer routing, and content discovery.

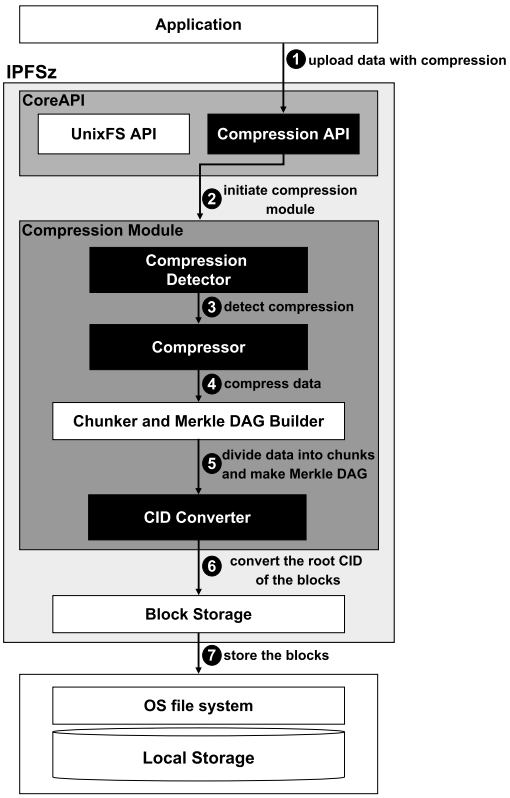


FIGURE 4. A procedure of compression when uploading in IPFSz.

Figure 2 describes the process overview on IPFSz. While processing an uploading request, IPFSz detects the request if it contains a compression command option. If the option is not included, the original data is uploaded without compression (①), otherwise the file format is parsed. If the file format is a well-known compression format such as zip, jpeg, mp3, and mp4, the original data is uploaded without compression (②), otherwise it is uploaded using compression (③). While processing a downloading request, IPFSz detects the CID that the compression flag is enabled (detailed in Section III-D). If the flag is disabled, the data can be downloaded immediately (④), otherwise decompression is performed during the downloading process (⑤).

B. ENABLING DATA COMPRESSION IN IPFSz

This section explains how to enable data compression functionality in IPFSz. To enable the functionality, we support managing the data states (e.g., compressed or decompressed). To do this, we utilize the content identifier (CID) in IPFSz whether performing the compression or decompression since all the files have their unique CID. Figure 3 shows how to manage the data states by using the CID in IPFSz. As shown in the figure, to support diverse hash function codes in IPFS, the CID has a multi-hash structure that consists of hash function code (1 byte), hash length (1 byte), and hash value (32 bytes). With this CID, IPFS provides a total of 12 hash functions, starting with the code number 17-29 as shown in Table 1. This means that the first 3 bits within 1 byte which

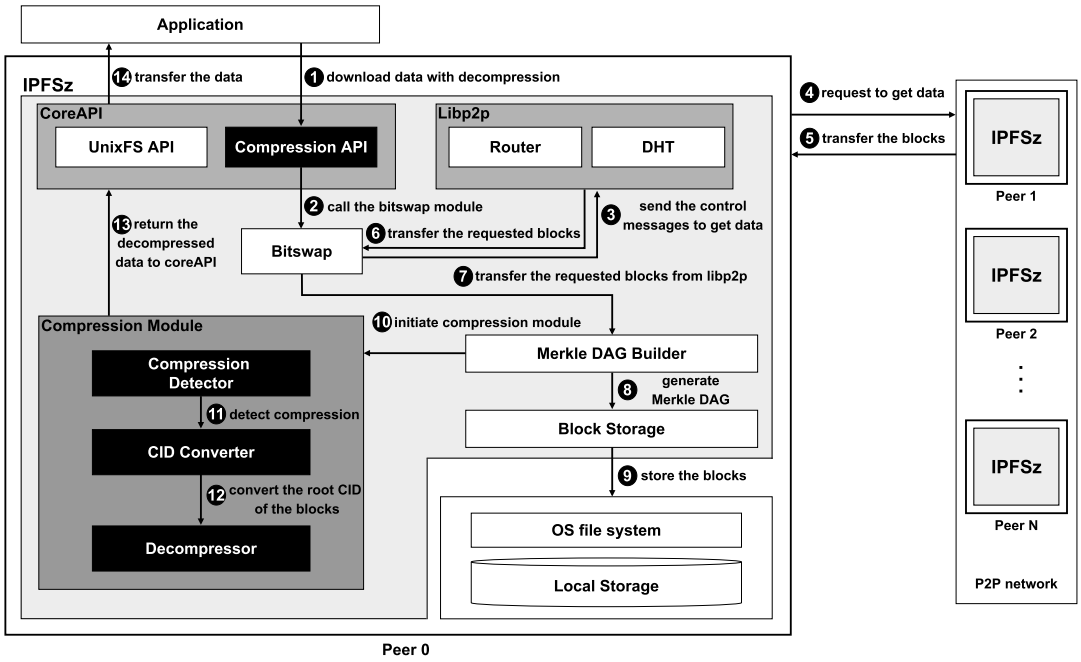


FIGURE 5. A procedure of decompression when downloading in IPFSz.

present the hash function code are unreserved areas on all the cases. Thus, we can employ a leftmost bit (i.e., 1 bit) as a compression flag by preserving the hash function code. For example, we set the compression flag in the CID to 1 when an application uploads the data. Meanwhile, when an application downloads data, we decompress the data if the flag is already set as 1. Then, we decompress the data and unset the flag (i.e., 0) and transfer the original data to the application. As a result, we can check whether the data states by using the CID and the IPFSz can determine whether performing compression or decompression.

C. PROCEDURE IN IPFSz

In this section, we describe the overall procedure of uploading and downloading data in IPFSz. Figure 4 shows the procedure of uploading in IPFSz. As shown in the figure, in the case of an upload operation, an application try to store the data to its local node in IPFSz with compression (①). In this scenario, the IPFSz uses the CompressionAPI instead of UnixFSAPI because of compression requests. After then, the CompressionAPI initiates the compression module to compress the data (②). In the compression module, the compression detector checks whether the request type is compression or decompression (③). If the request type is compression, the detector calls compressor. The compressor compresses data (④) by using a compression algorithm. Then, the chunker divides an entire data to chunks as a unit of a certain size (e.g., 256KiB).

After then, the Merkle DAG builder comprises a Merkle DAG with the chunks and their CIDs (⑤). Each node in Merkle DAG creates a block including a chunk and its unique CID. The CID converter sets the compression flag in the root CID of the blocks to represent that the data is compressed (⑥). The block storage stores the blocks including compressed data as well as the modified DAG metadata (e.g., updated root CID) to local storage (⑦). Consequently, this uploading procedure with data compression reduces the number of storage I/O operations and space consumption.

Figure 5 shows the overall procedure of downloading in IPFSz. As shown in the figure, a client node (i.e., peer 0) tries to download the data from other nodes (i.e., peer 1-N) connecting a P2P network via the CompressionAPI (①). The CompressionAPI calls the bitswap module for exchanging blocks of data (②). The bitswap sends the control messages to the router to utilize the P2P data exchange network (③). The router in libp2p provides peer and content routing. Peer routing is the process of finding peer addresses by using the information of other peer nodes. Content routing is the process of discovering some specific piece of data to download. The DHT in libp2p is a distributed hash table that is used for the routing algorithm. After the router searches nodes that have contents of requested data, the router requests the nodes to get the requested data via P2P network (④). Each node transfers the blocks associated with the data to the client node (⑤). After then, libp2p transfers the requested blocks to bitswap (⑥). After the bitswap receives all the requested blocks, the bitswap transfers them to the Merkle DAG builder (⑦). Then, the DAG builder creates a DAG to create a relationship between blocks (⑧).

After then, the block storage stores the blocks including compressed data and DAG metadata in local storage (⑨). To decompress the compressed data, the Merkle DAG builder initiates the compression module (⑩). In the compression module, the compression detector identifies whether the request type is decompression or not (⑪). If it is

decompression, the CID converter invalidates the compression flag in the root CID of the blocks (⑫). The decompressor merges the blocks to an entire data by using the DAG metadata and decompresses the entire data which is already compressed in the upload operation. After then, the decompressed data is returned to CoreAPI (⑬) to transfer the requested data to the application (⑭). Consequently, this downloading operation with data compression reduces the network and storage I/O operations since the compressed data is transferred between peers. In addition, it reduces the storage usage consumption in the downloading node as well as the uploading node because the downloading node stores the compressed data in the local storage (as shown in ⑨).

D. COMPATIBILITY WITH IPFS

To be compatible with the existing IPFS, we devise a mapping table by using a hash table between the original/compressed data's CID and the compressed/original data's CID. When the application sends a request to link the original data's CID and the compressed data's CID to IPFSz, IPFSz creates two elements of the mapping table for the request. First element consists of the original data's CID as the key and the compressed data's CID as the value. Second element consists of the compressed data's CID as the key and the original data's CID as the value. By doing so, we search mapping information of the original/compressed data's CID or compressed/original data's CID faster. For example, when an application requests to download the original data via its CID, IPFSz checks whether the CID is in the mapping table or not. If the CID exists as a key, IPFSz gets the compressed data's CID corresponding to the key, decompresses the compressed data, and transfers it to the application. Otherwise, the original data is transferred from another node to the application.

When compressed data is removed from the IPFSz node, IPFSz uses the compressed data's CID as a key to get the original data's CID and deletes two elements using their CIDs. In summary, we can provide compatibility with existing IPFS by linking compressed data and original data by using the mapping table. Since the mapping table is an in-memory structure, we guarantee the consistency of the mapping table by logging it into a log file whenever an element of the map table is changed.

IV. EVALUATION

A. EXPERIMENTAL SETUP

For our evaluation, we use Amazon EC2 with a general-purpose instance called t3.2×large (machines with 8 virtual CPUs and 32GB of RAM, general-purpose SSD (gp2) 100GB, and up to 5 Gbps network bandwidth). Experiments span up to five EC2 regions, which we call sites: Seoul (ap-northeast-2), Sydney (ap-southeast-2), California (us-west-1), Virginia (us-east-1), and London (eu-west-2). All nodes are configured to be directly connected to each other through the IPFS network. Table 2 is the result of measuring the ping latency between each node. As shown in the table, the average ping latency between these nodes ranges from 61ms to 265ms.

TABLE 2. Ping latency (milliseconds) between nodes.

	Seoul	Sydney	California	Virginia	London
Seoul	-	140	135.6	173.8	237.5
Sydney	140	-	137.6	197.3	264.4
California	135.6	137.6	-	60.9	147.5
Virginia	173.8	197.3	60.9	-	75.4
London	237.5	264.4	147.5	75.4	-

In this node configuration, as for the dataset, we collect some of the most popular datasets from Kaggle. Kaggle is one of the most popular platforms with many data science and AI datasets [17]. Since IPFSz does not compress the already compressed files by the compression detector, we show the evaluation text data type that is generally effective for compression. Table 3 shows each name, brief description, and data size of seven datasets. Notice that we abbreviate the name of datasets as Netflix (Netflix Movies and TV Shows), DS4C (Data Science for COVID-19), GTD (Global Terrorism Database), BHD (Bitcoin Historical Data), MD (The Movies Dataset), SVHN (Street View House Numbers), ARXIV(ArXiv Dataset), EMNIST (Extended MNIST), ENFT (Ethereum NFTs) in our evaluation.

B. PERFORMANCE RESULTS

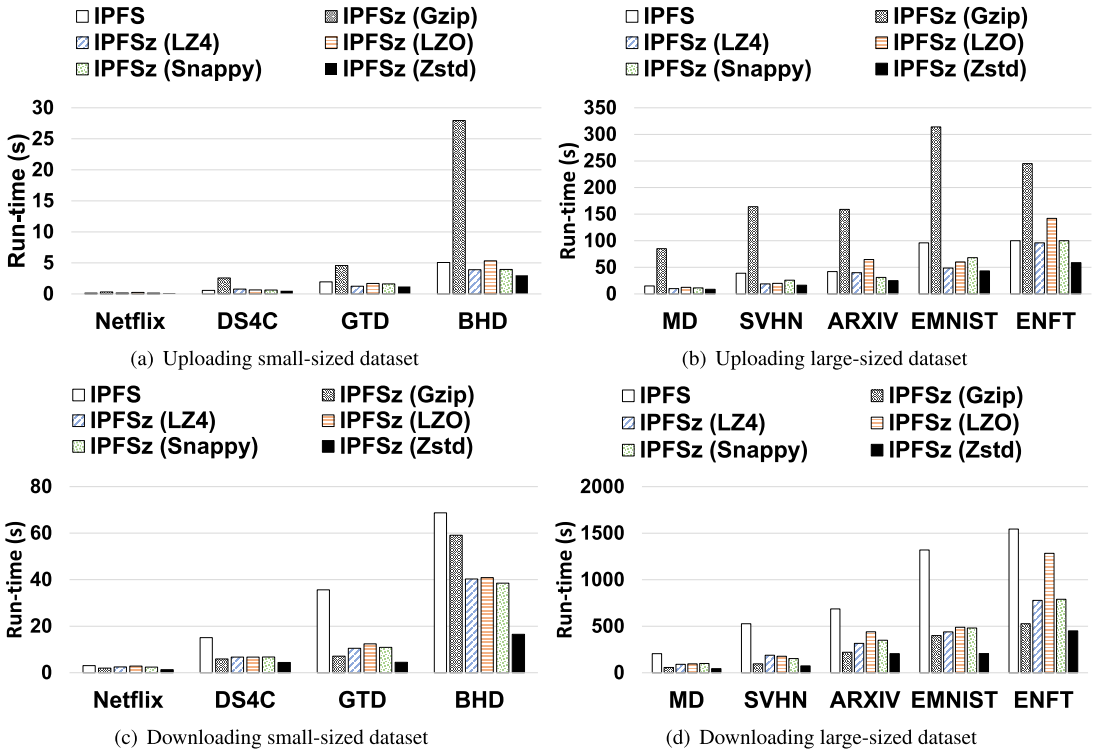
In this section, we show the performance results and space consumption with different datasets and compression algorithms. In the first subsection, we select Sydney node as an uploader and other sites as downloaders and measure the uploading and downloading time between the five nodes with all the compression algorithms and datasets. In the second subsection, we measure and show the storage space consumption of the Sydney node (i.e., uploader).

1) UPLOADING AND DOWNLOADING TIME

Figure 6 shows the experiment results of IPFS and IPFSz with different compression algorithms (i.e., Gzip, LZ4, LZO, Snappy, and Zstd) and different datasets. Figure 6(a) and 6(b) show the results of the uploading performance in the case of small and large-sized datasets, respectively. As shown in Figure 6(a), in the case of small-sized data, IPFSz with Zstd provides the highest performance and improves the performance by up to 1.05×, 1.12×, 1.61×, and 1.69× compared with IPFS in the case of Nexflix, DS4C, GTD, and BHD, respectively. Meanwhile, IPFSz with Gzip decreases the performance by up to 2.14×, 4.5×, 2.35×, and 5.51× than IPFS, respectively. It is because Gzip provides a good compression ratio but the compression speed is low. Especially, IPFSz with Gzip shows the lowest performance at BHD since the compression ratio of BHD itself is not efficient compared with the compression time. As shown in Figure 6(b) in the case of large-sized data, IPFSz with Zstd improves the performance by up to 1.72×, 2.37×, 1.68, 2.22× and 1.7× compared

TABLE 3. The datasets in our evaluation.

Name	Brief description	Size	Format
Netflix Movies and TV Shows	Listings of movies and TV shows on Netflix - Regularly Updated	3MB	.csv
Data Science for COVID-19	DS4C: Data Science for COVID-19 in South Korea (accept in NeurIPS 2020)	52MB	.csv
Global Terrorism Database	More than 180,000 terrorist attacks worldwide, 1970-2017	163MB	.csv
Bitcoin Historical Data	Bitcoin data at 1-min intervals from select exchanges, Jan 2012 to March 2021	318MB	.csv
The Movies Dataset	Metadata on over 45,000 movies. 26 million ratings from over 270,000 users	944MB	.csv
Street View House Numbers	Real-world image dataset for machine learning and object recognition	2.3GB	.png
ArXiv Dataset	arXiv dataset and metadata of 1.7M+ scholarly papers across STEM	3.16GB	.json
Extended MNIST	An extended variant of the full NIST dataset	5.78GB	.csv
Ethereum NFTs	On-chain activity from the Ethereum NFT market	6.92GB	.sqlite



(a) Uploading small-sized dataset

(b) Uploading large-sized dataset

(c) Downloading small-sized dataset

(d) Downloading large-sized dataset

FIGURE 6. The run-time of uploading and downloading datasets.

with IPFS in the case of MD, SVHN, ARXIV, EMNIST, and ENFT. Likewise, IPFSz with Gzip decreases the performance by up to 5.67×, 4.21×, 3.78×, 3.27×, and 2.45× compared with IPFS, respectively. It shows a similar result in the case of uploading small-sized datasets.

Figure 6(c) and 6(d) show the results of the downloading performance in the case of small and large-sized datasets, respectively. As shown in Figure 6(c), in the case of small-sized data, IPFSz with Zstd improves the performance by up to 2.34×, 3.43×, 7.92×, and 4.17× compared with IPFS in the case of Nexflix, DS4C, GTD, and BHD, respectively. Unlike the case of uploading datasets, the network traffic according to the number and size of blocks affects the overall performance (i.e., end-to-end data transmission time) since the blocks are transferred from remote nodes when downloading. For example, IPFS shows the worst performance since the increased number of blocks to be transmitted over the network affects more the performance than the low compression ratio and speed.

As for another example, in the case of all datasets except BHD, IPFSz with Gzip takes less downloading time than that of IPFSz with LZ4, LZO, and Snappy. It is because the higher compression ratio of Gzip is more effective compared with the faster decompression of LZ4, LZO, and Snappy. Meanwhile, IPFSz with Zstd provides the fastest performance on all datasets since Zstd has a compression algorithm with a reasonable compression ratio and fast decompression. As shown in Figure 6(b), in the case of large-sized data, IPFSz with Zstd/Gzip improves the performance by up to 4.63×/3.61×, 7.14×/5.48×, 3.35×/3.1×, 6.4×/3.3×, and 3.4×/2.9× compared with IPFS in the case of MD, SVHN, ARXIV, EMNIST, and ENFT respectively. As similar to the case of small-size data, Gzip with a high compression ratio shows better performance compared with IPFS, IPFSz with LZ4, LZO, Snappy.

Figure 7 shows the results of IPFS, IPFSz, and Optimized IPFSz with multimedia and binary files that are already compressed and are difficult to compress. The optimized

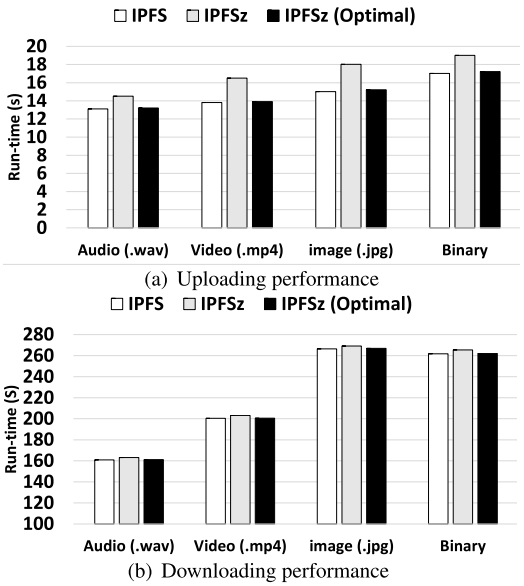


FIGURE 7. The run-time of uploading and downloading datasets (multimedia and binary file).

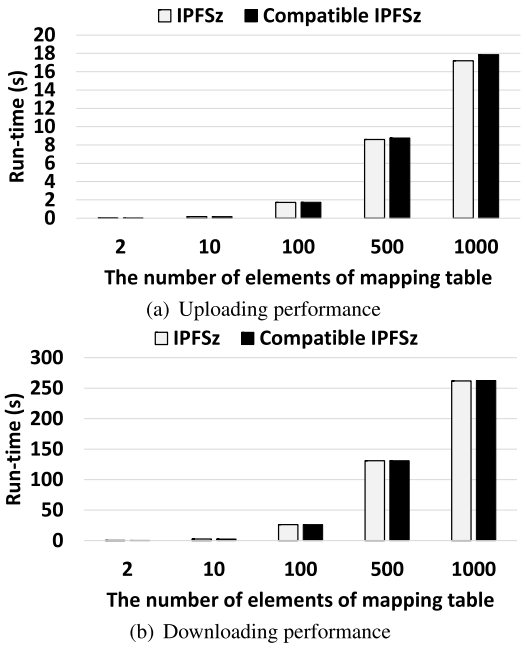


FIGURE 8. The run-time of uploading and downloading dataset (2MB binary files).

IPFSz adopts an approach of bypassing compression if the file format is the pre-compressed file format. The file types are audio (.wav), video (.mp4), image (.jpg), and binary files, and their sizes are 800MB, 900MB, 1GB, and 1GB. Also, the compression ratios of each data are 0.81, 0.96, 0.99, and 1, respectively. Figure 7(a) shows the uploading performance. The optimized IPFSz has almost the same performance as the existing IPFS and improves the performance by up to 1.06×, 1.18×, 1.17×, and 1.17×, compared with IPFSz in the case of audio, video, image, and binary file, respectively. The reason is that the optimized IPFSz does not compress through the bypassing compression approach. Whereas IPFSz degrades the performance due to the compression overhead. Figure 7(b) shows the downloading performance. The optimized IPFSz has almost the same performance as the existing IPFS and IPFSz. The reason for similar performance is that the decompression overhead is hidden by low downloading performance through network communication between geographically distant nodes.

2) MAPPING TABLE OVERHEAD

We evaluate the mapping table overhead effects when uploading and downloading in IPFSz. As shown in Figure 8, the compatible IPFSz denotes that IPFSz has compatible with the existing IPFS by using the mapping table. Otherwise, the IPFSz does not have compatible with existing IPFS without the mapping table. For evaluation, we use 2MB binary files, and the number of each request is 1, 5, 50, 250, and 500 times to create 2, 10, 100, 500, and 1000 elements of the mapping table, respectively. Figure 8(a) shows the uploading performance. The compatible IPFSz which manages the mapping table degrades the performance slightly by up to 1.02×, 1.02×, 1.01×, 1.02×, and 1.04×, compared with IPFSz without the update overhead of the mapping table. Figure 8(b) shows the downloading performance. The compatible IPFSz has almost the same performance as the IPFSz. The reason is that it takes about 650 milliseconds to update 1000 elements of the mapping table, thus the update overhead does not affect the performance of the uploading and downloading.

In addition, the recovery operation, which restores the mapping table when IPFSz node restarts via log file, does not affect the uploading and downloading performance since it affects only the restarting of the IPFSz node to join the network. Also, this operation time takes only 5 milliseconds to recover 1000 elements, which has much less overhead than the update overhead of the mapping table. Thus, the recovery operation does not affect the restarting time of the IPFSz node.

3) STORAGE SPACE CONSUMPTION

Table 4 shows the experiment results of the storage space consumption according to the datasets and compression algorithms. Each cell contains the storage space usage and compression ratio pair. The lower the compression ratio, the better the performance. The gray color cells in the table represent the largest storage space savings for each dataset. As shown in the table, IPFSz with Gzip provides the lowest space consumption and saves the storage space by up to 2.27×, 7.72×, 5.87×, 3.32×, 4.25×, 7.34×, 3.07×, 4.74×, and 3.74× compared with IPFS in the case of all datasets. IPFSz with Zstd provides the second lowest space consumption and saves the storage space by up to 2.09×, 6.37×, 5.93×, 3.28×, 3.76×, 6.34×, 2.85×, 4.44×, and 3.95× compared with IPFS. As shown in Figure 6, IPFSz with Zstd provides the highest performance while it provides similar storage space consumption to IPFSz with Gzip. Meanwhile, the Netflix dataset shows the least storage consumption among all the

TABLE 4. Storage space usage and compression ratio of datasets. First number and second number in each cell denote storage space usage and compression ratio, respectively. Grey cell shows the best storage space efficiency in each dataset.

	IPFS	IPFSz (Gzip)	IPFSz (LZ4)	IPFSz (LZO)	IPFSz (Snappy)	IPFSz (Zstd)
Netflix	3MB / 1	**1.32MB / 0.44**	1.92MB / 0.64	1.97MB / 0.66	2.14MB / 0.71	1.43MB / 0.48
DS4C	52MB / 1	**6.7MB / 0.13**	12.6MB / 0.24	13.6MB / 0.26	13.7MB / 0.26	8.2MB/ 0.16
GTD	163MB / 1	27.8MB / 0.17	40.9MB / 0.25	44.6MB / 0.27	49.3MB / 0.3	**27.5MB / 0.168**
BHD	318MB /1	**95.9MB / 0.3**	170.3MB / 0.54	160.9MB / 0.51	165.9MB / 0.52	97MB / 0.31
MD	944MB / 1	**221.9MB / 0.24**	377MB / 0.4	376.3MB / 0.4	386MB / 0.41	251.4MB / 0.27
SVHN	2.3GB / 1	**316.7MB / 0.14**	795.9MB / 0.34	699.2MB / 0.3	597.1MB / 0.26	366.5MB / 0.16
ARXIV	3.16GB / 1	**1.03GB / 0.33**	1.44GB / 0.46	2.14GB / 0.68	1.64GB / 0.52	1.11GB / 0.35
EMNIST	5.78GB /1	**1.22GB / 0.21**	1.74GB / 0.3	1.95GB / 0.34	1.98GB / 0.34	1.3GB / 0.26
ENFT	6.92GB / 1	1.85GB / 0.27	2.98GB / 0.43	5.86GB / 0.85	3.3GB / 0.48	**1.75GB / 0.25**

datasets in Table 4. It is because the Netflix dataset has less duplicated data to be compressed compared with other datasets.

V. RELATED WORK

A. DATA COMPRESSION SCHEMES

Previous approaches [18], [19], [20] present different data compression schemes to improve performance and reduce the size of data. Mogul et al. [18] proposed specific extensions to the HTTP protocol for the combination of delta encoding and data compression. It can provide improvements in response size and response delay for an important subset of HTTP content types. Fei et al. [19] introduced a novel lossless compression approach called SEAL for causality analysis. SEAL offers 2.63× and 12.94× data size reduction on the two real-world datasets, respectively. Besides, 89% of the queries are faster on the compressed dataset than the uncompressed one, and SEAL has perfectly the same query results as the uncompressed data. Jia et al. [20] presented a dynamic on-line compression scheme, called SlimCache. This scheme improves the cache hit ratio by virtually expanding the usable cache space through data compression. Xiong [21] presented a building information model (BIM) big data storage framework in order to achieve smooth browsing and query of BIM data by using efficient data compression. This framework provides the efficient Huffman compression algorithm replacing a construction process based on a binary tree with heap sorting. Our study is inspired by these studies and extends IPFS by adopting the compression technique to improve its I/O performance.

B. EXPLOITING COMPRESSION IN FILE SYSTEMS

Many file systems [22], [23], [24] support data compression at the file system level. For example, Btrfs [22], ZFS [23], and F2FS [24] can enable the data compression feature with different compression algorithms such as Zlib, LZO, Zstd, etc. With this feature, the file system data will be compressed automatically as new data is written to storage. When accessing the compressed data, the data will be automatically decompressed. This feature saves storage space, reduces the amount of write traffic, and extends the SSD life cycle. Ji et al. [25] presented a file access pattern guided compression (FPC) for a log-structured file system to reduce write I/O without additional user-side latency by using a dual-mode compression technique. Our study is in line with these file systems in terms of providing data compression at a file system layer. In contrast, we focus on providing data compression in the P2P distributed file system instead of local file systems.

C. IMPROVING I/O PERFORMANCE IN DISTRIBUTED SYSTEMS

Previous studies [3], [4], [26], [27], [28] investigated the P2P distributed systems to improve the performance. Chen et al. [4] proposed a storage strategy in IPFS, which combines triple replication schemes and erasure codes storage scheme to solve the large data storage problem of service providers. Zhang et al. [26] proposed a scalable architecture of a cloud storage system for small files on a P2P distributed system. In this architecture, the central routing node stores the status and routing information of all data nodes and the data nodes store content and metadata of files. Zhang et al. [27] proposed a small file merging strategy (SFMS) on P2P distributed file system by adding a virtual node and the file index tree based on the hash value. This strategy stores the small files after merging them into a big file, which improves higher throughput for reading small files.

Kim et al. [29] presented an efficient SmartNIC offload and high performance of a distributed file system, called LineFS. LineFS takes advantage of spare SmartNIC processing capacity to perform the LZW algorithm which is one of the lossless data compression algorithms, saving network bandwidth for replication and alleviating write amplification by reducing the amount of data. Bharambe et al. [28] presented a simulation-based study of BitTorrent. In this study, they deconstruct the system and evaluate the impact of its core mechanisms by focusing on important metrics (e.g., peer link utilization, file download time, and fairness amongst peers). Le et al. [3] provided the high performance of IPFS by using key-value memory caching. It offers to reduce the time of finding data provider nodes and makes it faster to retrieve data from IPFS. Our study is in line with these studies in terms of investigating the performance of distributed systems. In contrast, we focus on reducing the storage and network I/O operations by using a data compression scheme in the P2P distributed file system.

VI. DISCUSSION AND FUTURE WORK

We could consider an alternative approach that adds a compression/decompression server connected between an

TABLE 5. Comparison of best I/O performance and space efficiency according to data type and size.

Data	Format	Original data size	Best algorithm for space efficiency / Compressed data size	Best algorithm for uploading	Best algorithm for downloading
Netflix	.csv	3MB	IPFSz (Gzip) / 1.32MB	IPFSz (Zstd)	IPFSz (Zstd)
DS4C	.csv	52MB	IPFSz (Gzip) / 6.7MB	IPFSz (Zstd)	IPFSz (Zstd)
GTD	.csv	163MB	IPFSz (Zstd) / 27.5MB	IPFSz (LZ4)	IPFSz (Zstd)
BHD	.csv	318MB	IPFSz (Gzip) / 95.9MB	IPFSz (Zstd)	IPFSz (Zstd)
MD	.csv	944MB	IPFSz (Gzip) / 221.9MB	IPFSz (Zstd)	IPFSz (Zstd)
SVHN	.png	2.3GB	IPFSz (Gzip) / 316.7MB	IPFSz (Zstd)	IPFSz (Zstd)
ARXIV	.json	3.16GB	IPFSz (Gzip) / 1.03GB	IPFSz (Zstd)	IPFSz (Zstd)
EMNIST	.csv	5.8GB	IPFSz (Gzip) / 1.22GB	IPFSz (Zstd)	IPFSz (Zstd)
ENFT	.sqlite	6.92GB	IPFSz (Zstd) / 1.75GB	IPFSz (Zstd)	IPFSz (Zstd)

application and IPFS nodes. The server compresses the data received from the client and sends it to the IPFS node. Also, the server decompresses the compressed data from the IPFS node and then sends it to the client. However, this approach can degrade the performance of uploading and downloading requests because the server is centralized and processes excessive compression/decompression for all connected IPFS nodes. Another alternative approach is that the application communicates with the IPFS node using compression in HTTP protocol. This approach is inspired by many studies using IPFS only at the storage level and with a separate service server [30], [31]. The problem of the approach is that we should modify the HTTP protocol to apply compression algorithms for better performance. One of our goals is that the compression scheme in a p2p distributed system such as IPFS can provide higher uploading and downloading performance without depending on compression from network protocols. Thus, we extend IPFS by enabling compression functionality while maintaining the fundamental features of IPFS.

IPFSz used general-purpose compression algorithms to compress data, and in most cases Zstandard has the best performance improvement. As shown in Table 5, Gzip shows the best space efficiency in the case of almost all the data type and size. Also, Zstd has a similar space efficiency compared with Gzip. In some cases (GTD and ENFT), Zstd shows better space efficiency than Gzip. The reason for the similar space efficiency is that both Gzip and Zstd are based on statistical compression algorithm (e.g., Huffman coding) using character frequencies. In terms of I/O performance, Zstd shows the best performance for almost all the file types. In the case of uploading the GTD, LZ4 shows a better performance than that of Zstd but shows almost the same performance. The reason for the similar performance is that the compression operation of LZ4 and Zstd is based on LZ77 which is widely used as the core of lossless compression algorithms. Also, Zstd shows the best performance when we use text data type compared with other types. However, in the case of multimedia content, the performance improvement is insignificant and sometimes performance degradation occurs because the multimedia data has already been encoded. Therefore, we have adopted an approach of bypassing compression if it is a well-known compression format. We will extend the scheme by considering specific compression schemes for the compression formats (e.g., image, video, audio). Finally, we will design and implement a dynamic compression approach to adapt the most efficient compression algorithm during run-time according to the characteristics of datasets and different configurations.

VII. CONCLUSION

In this paper, we investigate the procedure of operations in IPFS. We find that the performance of IPFS is worse as the data size increases. To handle this issue, we present IPFSz which is a variant of IPFS to enable data compression functionality for better I/O performance and storage space consumption. The experiment results show higher uploading and downloading performance by up to 7.9× and save the storage space by up to 6.3× compared with existing IPFS.

REFERENCES

[1] J. Benet, "IPFS—Content addressed, versioned, P2P file system," 2014, *arXiv:1407.3561*.
[2] E. Daniel and F. Tschorsch, "IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks," *IEEE Commun. Surveys Tuts.*, vol. 24, no. 1, pp. 31–52, 1st Quart., 2022.
[3] V. Le, R. Moazeni, and M. Moh, "Improving security and performance of distributed IPFS-based web applications with blockchain," in *Proc. Int. Conf. Adv. Cyber Secur.* Cham, Switzerland: Springer, 2021, pp. 114–127.
[4] Y. Chen, H. Li, K. Li, and J. Zhang, "An improved P2P file system scheme based on IPFS and blockchain," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2652–2657.
[5] J. Shen, Y. Li, Y. Zhou, and X. Wang, "Understanding I/O performance of IPFS storage: A client's perspective," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Jun. 2019, pp. 1–10.
[6] O. A. Lajam and T. A. Helmy, "Performance evaluation of IPFS in private networks," in *Proc. 4th Int. Conf. Data Storage Data Eng.*, Feb. 2021, pp. 77–84.
[7] (2014). *UnixFS*. [Online]. Available: https://github.com/ipfs/go-unixfs
[8] (2001). *Protocol Buffers*. [Online]. Available: https://developers.google.com/protocol-buffers
[9] P. Deutsch. *Deflate Compressed Data Format Specification Version 1.3*, document RFC 1951, 1996.
[10] (2011). *LZ4—Extremely Fast Compression*. [Online]. Available: https://github.com/lz4/lz4
[11] (1996). *Lempel-Ziv-Oberhumer*. [Online]. Available: http://www.oberhumer.com/opensource/lzo/
[12] (2011). *Snappy*. [Online]. Available: https://github.com/google/snappy
[13] (2016). *Zstandard*. [Online]. Available: https://github.com/facebook/zstd
[14] M. Mahoney. (2016). *Large Text Compress. Benchmark*. [Online]. Available: http://mattmahoney.net/dc/text.html
[15] M. Mahoney. (2019). *Silesia Open Source Compression Benchmark*. [Online]. Available: http://mattmahoney.net/dc/silesia.html
[16] *CoreAPI*. Accessed: 2018. [Online]. Available: https://docs.ipfs.tech/reference/go/api/#go-coreapi
[17] (2010). *Kaggle*. [Online]. Available: https://www.kaggle.com/
[18] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 1997, pp. 181–194.

[19] P. Fei, Z. Li, Z. Wang, X. Yu, D. Li, and K. Jee, "SEAL: Storage-efficient causality analysis on enterprise logs with query-friendly compression," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 2987–3004.
[20] Y. Jia, Z. Shao, and F. Chen, "SlimCache: An efficient data compression scheme for flash-based key-value caching," *ACM Trans. Storage*, vol. 16, no. 2, pp. 1–34, Jun. 2020.
[21] G. Xiong, "Research on big data compression algorithm based on BIM," in *Proc. IEEE Int. Conf. Power, Intell. Comput. Syst. (ICPICS)*, Jul. 2021, pp. 97–100.
[22] 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.
[23] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The Zettabyte file system," in *Proc. 2nd Usenix Conf. File Storage Technol.*, vol. 215, 2003, pp. 1–13.
[24] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 273–286.
[25] C. Ji, L.-P. Chang, R. Pan, C. Wu, C. Gao, L. Shi, T.-W. Kuo, and C. J. Xue, "Pattern-guided file compression with user-experience enhancement for log-structured file system on mobile devices," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 127–140.
[26] Q.-F. Zhang, X.-Z. Pan, Y. Shen, and W.-J. Li, "A novel scalable architecture of cloud storage system for small files based on P2P," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Sep. 2012, pp. 41–47.
[27] Y. Zhang, Q. Zhang, Y. Chen, C. Tu, E. Liu, J. Ren, and Y. Xue, "A small file performance optimization algorithm on P2P distributed file system," in *Proc. IEEE 16th Int. Conf. Commun. Technol. (ICCT)*, Oct. 2015, pp. 487–492.
[28] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analyzing and improving a BitTorrent networks performance mechanisms," in *Proc. IEEE INFOCOM. 25TH IEEE Int. Conf. Comput. Commun.*, 2006, pp. 1–12.
[29] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ. CD-ROM*, Oct. 2021, pp. 756–771.
[30] J. Sun, X. Yao, S. Wang, and Y. Wu, "Blockchain-based secure storage and access scheme for electronic medical records in IPFS," *IEEE Access*, vol. 8, pp. 59389–59401, 2020.
[31] M. Zichichi, S. Ferretti, and G. D'Angelo, "On the efficiency of decentralized file storage for personal information management systems," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2020, pp. 1–6.

MANSUB SONG received the B.S. degree in computer engineering from Yeungnam University, in 2019. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. His research interests include blockchain and distributed systems.

JONGBEEN HAN received the B.S. degree in computer engineering from Hansung University, in 2015, and the M.S. degree from the Department of Computer Science and Engineering, Seoul National University, in 2019, where he is currently the Ph.D. degree in computer science and engineering. His research interests include blockchain, operating, and distributed systems.

HYEONSANG EOM received the B.S. degree in computer science and statistics from Seoul National University (SNU), Seoul, South Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of Maryland at College Park, MD, USA, in 1996 and 2003, respectively. He was an Intern at the Data Engineering Group, Sun Microsystems, CA, USA, in 1997, and a Senior Engineer at the Telecommunication Research and Development Center, Samsung Electronics, South Korea, from 2003 to 2004. He is currently a Professor with the Department of Computer Science and Engineering, SNU, where he has been a Faculty Member, since 2005. His research interests include high performance storage systems, operating systems, distributed systems, cloud computing, energy efficient systems, fault-tolerant systems, security, and information dynamics.

YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science, Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana-Champaign. Currently, he is an Assistant Professor with the Department of Computer Science and Engineering, Chung-Ang University. His research interests include operating, distributed, and database systems.