Electronic Preprint for Journal of Information Processing Vol.24 No.5

Regular Paper

Design of Object Storage Using OpenNVM for High-performance Distributed File System

Fuyumasa Takatsu[1,a)] Kohei Hiraga[1,b)] Osamu Tatebe[2,c)]

Received: November 6, 2015, Accepted: March 8, 2016

Abstract: The current trend for high-performance distributed file systems is object-based architecture that uses local object storage to store the file data. The IO performance of such systems depends on the local object storage that manages the underlying low-level storage, such as Fusion IO ioDrive, a flash device connected through PCI express. It provides OpenNVM flash primitives, such as atomic batch write and sparse addressing. We designed an object storage using OpenNVM whose goal is to maximize IOPS/bandwidth performance. Using the sparse address space, it is possible to design object storage as an array of fixed-size regions. Using atomic batch write, the object storage supports the ACID properties in each write. Our prototype implementation achieves 740,000 IOPS for object creations using 16 threads, which is 12 times better than DirectFS. The write performance achieves 97.7% of the physical peak performance on average.

Keywords: object storage, OpenNVM, distributed file system

1. Introduction

The current trend in high-performance distributed file systems is object-based architecture. The file system metadata is managed by metadata servers, and the file data is managed by object storage servers. The file metadata and file data are stored in a local file system or a local object storage. The IO performance of a distributed file system depends on the local file system or the local object storage that manages the underlying low-level storage.

Flash devices and storage class memory can improve local file system performance; however, the performance improvement is limited because the design of the file system such as ext4 [1] and ZFS [2] is based on the hard-disk drive (HDD) property. To improve HDD performance, serial and sequential access is important. However, flash device and storage class memory require different access patterns in order to improve the IO performance. Given that there is no mechanical mechanism for disk head seek, this is not necessary to consider. In addition, parallel access provides better performance than the serial and sequential access. Moreover, further functionality compared with traditional block devices, such as sparse address space, persistent trim and atomic batch write, are proposed using Flash Translation Layers (FTLs) [3], [4], [5], [6], [7]. Therefore, it is quite crucial to re-design object storage specifically for flash devices in order to improve the IO performance of high-performance distributed file systems.

[1] Graduate School of System and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[2] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] takatsu@hpcs.cs.tsukuba.ac.jp
[b)] hiraga@hpcs.cs.tsukuba.ac.jp
[c)] tatebe@cs.tsukuba.ac.jp

In the big data analysis field, data is stored in storage devices because the data is too large to store in memory. Therefore, there is a growing performance requirement to storage. Flash devices and storage class memory have high physical performance. However, current storage system is not used effectively. Because many-core systems will be used in such a field, the thread concurrency will be more important. Hence, the big data analysis would allow applications to be written to multi objects in parallel. Furthermore, in HPC the file-per-process checkpointing is a create-intensive workload, which creates hundreds of thousands of files at the same time. PLFS [8] introduces the parallel log file system to solve this problem. Besides the checkpointing, there are several create-intensive applications including gene sequencing, image processing, and phone and video logs, that require the metadata operation performance such as file creations. There are several works, e.g., GIGA+ [9] and IndexFS [10], to attack this problem. Current storage systems have a problem that the file creation performance is not improved by parallel accesses. The more the cores are increased, the more important this problem becomes. We solved this problem by taking advantage of the special features of OpenNVM flash primitives [11] that is one of the proposed standards of the new FTL functionality.

This paper describes the object storage design for flash devices and storage class memory using OpenNVM flash primitives, which is assumed to be used for high-performance distributed file systems. Namespace and access control in distributed file systems are provided by metadata servers, and the underlying object storage does not require it. Using the sparse address space, it is possible to design object storage as an array of fixed-size regions. The objects can be addressed directly by the region number. Non-blocking design improves the IO performance by parallel accesses. It shows 740,000 Input/Output Operations Per

Second (IOPS) for object creations that use 16 threads, which is 12 times better than DirectFS. With regard to the write performance, the prototype implementation achieves 97.7% of the physical peak performance on average.

The contributions of this paper are as follows:

- a non-blocking object storage design that uses OpenNVM flash primitives for high-performance distributed file systems
- two object layouts in a region for high-performance read and write, and for snapshot
- optimization techniques to improve object creation performance using atomic batch write operation
- the prototype implementation achieves 740,000 IOPS for object creations, and 97.7% of the physical peak performance on average.

The remainder of this paper is organized as follows. Section 2 provides a brief description on the Fusion IO ioDrive, and related work for object storage. Section 3 introduces our approach, whose implementation is described in Section 4. The prototype implementation is evaluated in Section 5, and we conclude our work in Section 6.

2. Background

In this section, we provide a brief description on object storage and Fusion IO ioDrive.

2.1 Object Storage

High-performance distributed file systems use a local low-level storage device to store the file metadata and file data. The local file system provides a hierarchical namespace for these devices and the data managed as a file in the local file system. Such system also provides the features to create and remove files. Object storage servers use these local file systems; for example, Ceph [12] uses Btrfs [13] and Lustre [14] uses ext4 [1]/ZFS [2].

On the other hand, local file systems have richer functionality than required by high-performance distributed file systems, which may cause unnecessary overhead. One such functionality is the hierarchical namespace. Traditional UNIX file systems manage it through directory entries that manage the file name and inode number. When two or more files are created in the same directory, lock contention occurs for the same directory entry. **Figure 1** shows the performance of file creation in the same directory by multiple threads. In Fig. 1, the horizontal axis shows the number of threads, and the vertical axis shows the number of operations per second. The file creation performance is not improved by parallel accesses because of lock contention.

Object storage [15] does not have a hierarchical namespace. It only has a flat namespace that can avoid this lock contention problem for the directory entry.

2.2 Fusion IO ioDrive

Fusion IO ioDrive is a NAND flash memory connected with Peripheral Component Interconnect (PCI) express. IoDrive supports additional functionality provided by OpenNVM flash primitives. These primitives include sparse address space and atomic batch write. This paper mainly utilizes these two functionalities.

With OpenNVM version 0.7, we can use 144 PB sparse address space regardless of the physical ioDrive capacity. Mapping between the sparse and physical block addresses is managed by OpenNVM. All virtual blocks are available, but only written blocks are physically assigned in a low-level storage device.

The functionality of atomic batch write is to write multiple blocks atomically. Using atomic batch write, double write is not required to maintain consistency. To ensure the updates of several blocks, double write technique is often used to avoid partial write, while it requires to write data twice. The atomic batch write ensures the updates of multiple blocks, which means all blocks are updated or unchanged. That is why we do not need double write to maintain consistency when using the atomic batch write operation. To use atomic batch write, an application uses the function named nvm_atomic_batch_operations. nvm_atomic_batch_operations writes data through an IO vector. In the current version of SDK, this function can write 128 IOs at most.

Figure 2 shows the results of the performance evaluation for ioDrive nvm_atomic_batch_operations. In this evaluation, nvm_atomic_batch_operations write 2 GiB data by changing the number of IOs from 1 to 128. The block size is 512 bytes. In Fig. 2, the horizontal axis shows the number of IOs in an nvm_atomic_batch_operations, and the vertical axis shows the number of written IOs per second. In addition, we changed the number of worker thread from 1 to 32. Each line corresponds to a different number of threads.

Figure 2 shows two interesting facts. One is that better performance is shown using many worker threads. Another is that bet-

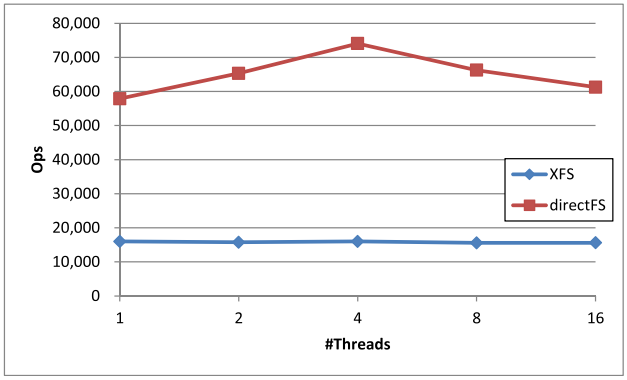


Fig. 1 File creation performance to same directory by multiple threads.

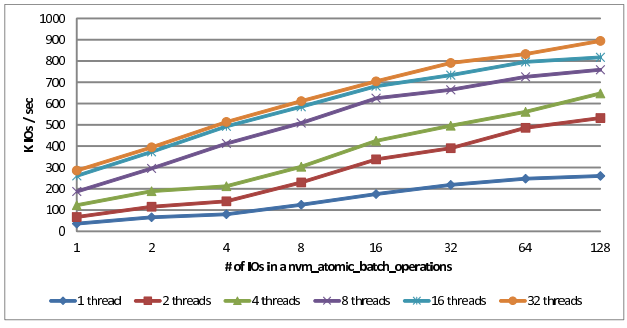


Fig. 2 Performance evaluation results for ioDrive nvm_atomic_batch_operations.

ter performance is shown when writing many IOs. For example, when using 32 worker threads, over 894 K IOs/sec is achieved when writing 128 IOs in one nvm_atomic_batch_operations. This represents over three times better performance compared with writing one IO in one operation.

The file system that takes advantage of the special features of ioDrive already exists. It is Direct File System (DFS) [16]. However, DFS provides a hierarchical namespace to occur lock contention because DFS is a POSIX-compliant file system. Therefore, the object storage taking advantage of the special features of ioDrive is quite crucial.

3. Our Approach

We designed high-performance object storage for a storage device by utilizing a new interface provided by OpenNVM flash primitives. Our target device has the following features:

(1) high parallel access performance
(2) a sparse address space that goes beyond the physical capacity of the device
(3) atomic-batch-write to multiple addresses.

Our proposed object storage is also able to apply our approach to other devices that satisfy these requirements, such as ANViL [17].

We assume these features to design object storage for a distributed storage system.

In a high-performance distributed storage system, the file system metadata, including namespace, time and access control information, are managed by metadata servers. The underlying object storage is not necessary for managing them. It is sufficient to address an object through a unique identifier.

The designed object storage provides the following features:

(1) creates an object and returns the object ID
(2) writes and reads data at the specified offset for the specified object

In this section, we provide a brief description on our approach that achieves these features.

3.1 Region

We assume that the sparse address space is 64 bits or larger. Only written blocks are physically assigned in a low-level storage device. Using the sparse address space, it is possible to design object storage simply. Our approach uses this space as an array of fixed-size regions as depicted in **Fig. 3**. Region size is specified as being sufficiently large to store the largest object in a storage system. Because the sparse address space is 64 bit or larger, even if the region size is large, the object storage can store a sufficient number of objects. However, this is not necessary. In fact, other object storage systems, such as OpenStack Swift, have the limitation of object size. If the Region size is specified small, many objects are supported. As shown in Fig. 3, the objects can be addressed directly by the region number, as the object ID.

The first region, which we call 'super region', stores meta-information of the object storage. Other regions manage one object. Dividing the address space to fixed-size large regions is similar to OBFS [18]. However, OBFS stores multiple small objects in a region in order to use the limited address space efficiently,

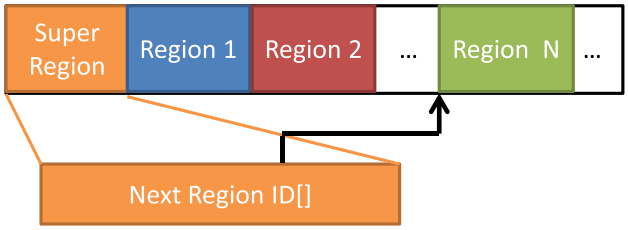


Fig. 3 Dividing sparse address space into regions.

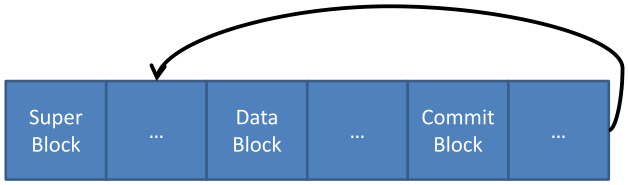


Fig. 4 Data structure of the region in Version Mode.

which requires a hash table to look up an object. This requires not only additional memory space, but also expensive linear search when the number of objects increases compared with the hash table size. On the other hand, our approach stores only one object in each region, which allows looking up an object by region number, i.e., object ID.

An object ID is assigned when the object is created. To avoid lock contention when assign the object ID, lock-free management is preferred, which is described in Section 4.

3.2 Data Management Technique in a Region

Each region, with the exception of the super region, manages object data. In this section, we provide a brief description on the data management technique for the region.

Our object storage is assumed to be used by distributed storage systems. Some such systems create replicas to multiple storage nodes. In this case, collision detection for simultaneous writes to different object storages is extremely important in order to keep consistency among replicas. Vector Clock [19] is one of the methods used for detecting collisions. Using log-structured data layout, all versions in Vector Clock can be stored, but it may cause poor read performance. On the other hand, there is a case where applications write data once and read mostly. In this case, we do not need to maintain each version.

Thus, there are various requirements depending on the purpose. In order to satisfy each request we propose two object layouts, Version and Direct in each region. Version Layout stores all the versions. The log to change data is appended with the commit block that have the version as the log-structured data format. Direct Layout stores only the latest version.

3.2.1 Version Mode

This layout stores all object versions that can help to be migrated or snapshot.

The data structure is shown in **Fig. 4**. In this figure, the object store maintains a circular log that includes the commit and data blocks. The log meta-information is stored in the super block. By managing the data as a circular log, writing data is efficient because it only appends the data and the commit blocks. To append the data and commit blocks at every write, the version can be managed at every write. Because this data layout stores all

versions, it is possible to roll back to any version.

3.2.2 Direct Mode

This layout stores data directly without any version management.

The data structure is shown in **Fig. 5**. In this figure, there are two layouts, one with size and one without the size. Direct Mode with size has the super block to manage the object metadata, such as object size as shown in Fig. 5 (a). Direct Mode without size doesn't have the super block as shown in Fig. 5 (b).

If the application requires the object size, it should use the layout shown in Fig. 5 (a). When the metadata server of a distributed storage system manages the object size, the data layout shown in Fig. 5 (b) is used. Writing or reading data is done directly at the specified offset, and the highest performance of read and write is assumed in this mode.

4. Implementation

We use the OpenNVM version 0.7 to implement a prototype system. Note that our proposed design does not depend on the OpenNVM. It can be implemented by a standard interface that supports the features described in Section 3. The prototype provides API libraries to create, read, and write an object for a programming interface. Therefore, the library is used to build client applications. Using OpenNVM, we can use 144 PB sparse address space. Our implementation divides this sparse address space into regions and maps to the objects. Region size is a parameter in this implementation. When users specify 256 GB for a region size, which may be larger than the physical capacity, the object storage supports approximately 590 K objects.

In this section, we provide a brief description on our implementation and optimization to achieve high IO performance.

4.1 Version Mode

Data structure in a region is shown in **Fig. 6**. In this figure, the super block manages the address of the last commit block and the head and tail block address of the circular log in a region. The commit block manages the address of the previous (parent in the figure) commit block, write timestamp, and the address of data block that stores the data. Commit block is generated by every write. The version is managed by managing the commit block as a linked list.

When creating an object, a new object ID is assigned within the reserved range for each worker thread without acquiring any lock. The information required to assign a new object ID is managed in the super region and memory. The super block in a new region is initialized to store data in a log structured format.

When writing data, a new set of parent commit block, timestamp and data block is appended to the circular log list.

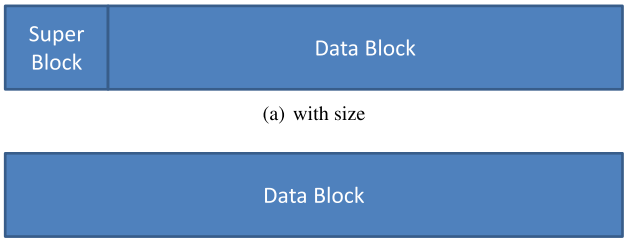


Fig. 5 Data structure for region in Direct Mode.

4.2 Direct Mode

When creating an object, a newly assigned region is initialized. In our implementation, we initialize the first block by zero in order to verify that it exists.

When reading or writing data, the data are accessed directly at the specified offset.

4.3 Optimization for Updating Super Region

When creating an object, the object storage initializes the super block and updates the super region. The super region content is also stored in the memory. Therefore, a new object ID is generated from the data on memory. When creating an object, the object storage updates the data on memory and the super region. However, there is an overhead for updating the super region at every creation process.

To reduce this overhead, the super region is updated with every N creations rather than every creation. This parameter (N) is called 'Skip-number'. When one object is created, the super block is updated as if N objects are created. After that, the super block is not updated until N objects are created. That is why the number of updates of the super region is reduced to one Nth. When failure occurs, there may be some objects that are not to be fully created. Even in this case, no physical space is wasted, since only allocated physical space is used due to the sparse address space.

4.4 Optimization for Initializing Super Block

When creating an object, the object storage also initializes the super block in a new region. There is an overhead to update the super block at every creation process. As indicated in

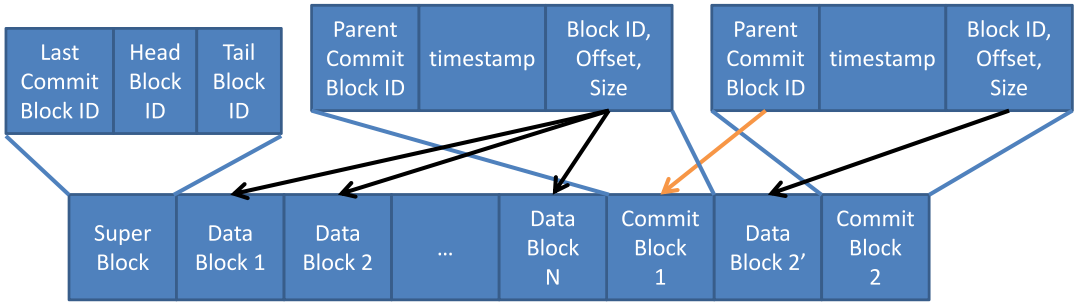


Fig. 6 Detailed data structure of Version Mode.

Section 2.2, OpenNVM shows better performance in the case of multiple writes. To improve initialization performance at object creation, we utilize the atomic batch write feature in OpenNVM. In this optimization, the object storage initializes not only the super block of the new object but also the super blocks of the objects that will be created in the near future with one nvm_atomic_batch_operations. In other words, the object storage initializes super blocks of multiple objects every N time. This may waste the initialized block space when failures occur, but the space is a few kilobytes at most.

5. Evaluation

We evaluate the prototype implementation of the object storage designed in this paper. This section provides the methods and results of these evaluations.

5.1 Evaluation environment

The environment for the node we evaluate in our approach is indicated in **Table 1**.

5.2 Access Performance Evaluation

This evaluation measures the read and write performance in each block size.

5.2.1 Read Performance Evaluation

This evaluation measures the read performance in each block size. Before evaluating the read performance, we prepare the write object to be the same block size as the read object. This means that the object has many versions in Version Mode because such mode creates a version every write operations. For the read performance evaluation, we read 1 GiB data from the object randomly or sequentially. We evaluate ten times and calculate the average in each block size. Furthermore, we evaluate the read performance of directFS, which is the local file system for ioDrive, for comparison. Before evaluating the read performance, we prepare the write file to be the same block size as the read file.

Table 1 Node specification.

CPU	Intel(R) Xeon(TM) E5620 CPU 2.40 GHz (4 cores 8threads) x2
RAM	24 GB
OS	CentOS 6
Storage Device	Fusion-io ioDrive 160 GB
SDK	OpenNVM (Version 0.7)

And, we use 1 thread for this evaluation.

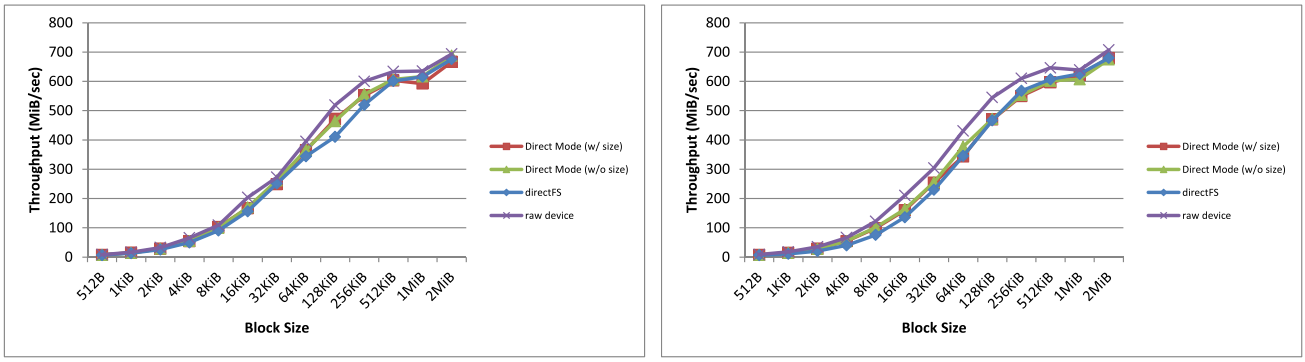
The results of the read performance evaluation are shown in **Fig. 7**. In each graph, the horizontal axis shows the block size, and the vertical axis shows the number of read bytes per second. In the read access performance evaluation, Direct Mode shows proportional performance to the block size. It achieves 687 MB/sec on sequential read using Direct Mode without size. In the read performance, Direct Mode with size shows almost the same performance as Direct Mode without size. That is because it does not require any extra access to the super block unlike the write operation. In Figs. 7 (b) and 7 (a), there is no graph in Version Mode. This is because we do not assume that the application read the file with many versions. Comparing with the directFS, Direct Mode without size achieves 94.0% of the sequential read performance on average and 90.0% of the random read performance on average. Furthermore, comparing the result of Direct Mode without size and raw device, Direct Mode without size achieves 91.2% of the sequential read performance of the physical peak performance on average and 86.2% of the random read performance of the physical peak performance on average.

5.2.2 Write Performance Evaluation

In this evaluation, we assess the writing performance in each block size by changing the number of worker threads. Before evaluating the write performance, we prepare the same number of objects as worker threads. For the write performance evaluation, we write 1 GiB data to the objects randomly or sequentially. We evaluate ten times and calculate the average in each block size.

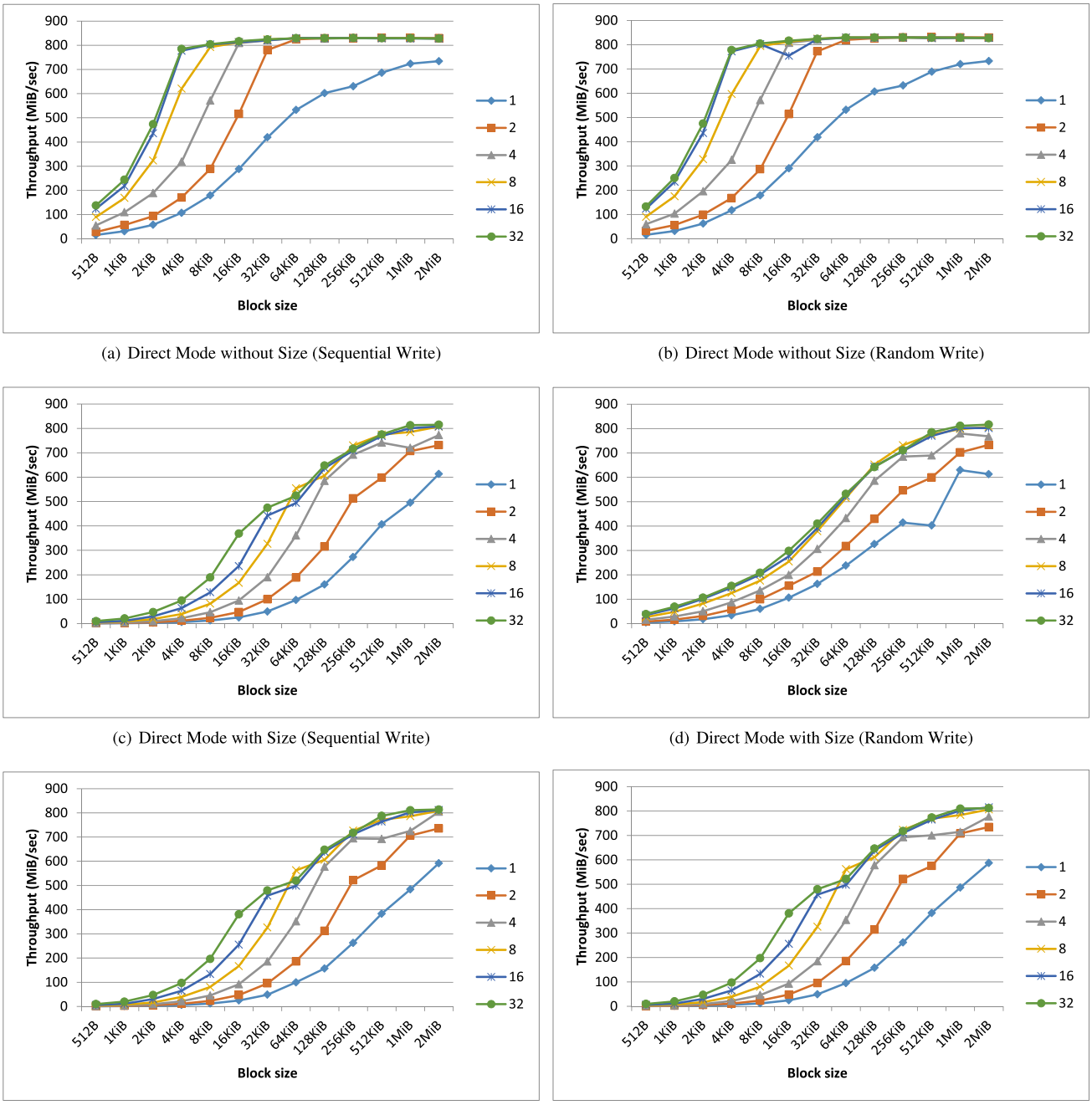
The results of the write performance evaluation are shown in **Fig. 8**. In each graph, the horizontal axis shows the block size, and the vertical axis shows the number of write bytes per second. Each line in Fig. 8 shows a different number of worker threads. It achieves 830 MB/sec on sequential and random write using Direct Mode without size. Comparing with Direct Mode without size, Direct Mode with size achieves 41.22% of the sequential write performance, and 52.12% of the random write performance. In write operation, Direct Mode with size reads a super block. If the object size is changed, Direct Mode with size also updates the super block. These two extra operations cause this performance degradation.

We also evaluated the write performance of directFS and raw device. The results of the write performance evaluation of di-



(a) Sequential Read (b) Random Read

Fig. 7 Read performance evaluation results in each block size.



(a) Direct Mode without Size (Sequential Write)

(b) Direct Mode without Size (Random Write)

(c) Direct Mode with Size (Sequential Write)

(d) Direct Mode with Size (Random Write)

(e) Version Mode (Sequential Write)

(f) Version Mode (Random Write)

Fig. 8 Write performance evaluation results in each block size.

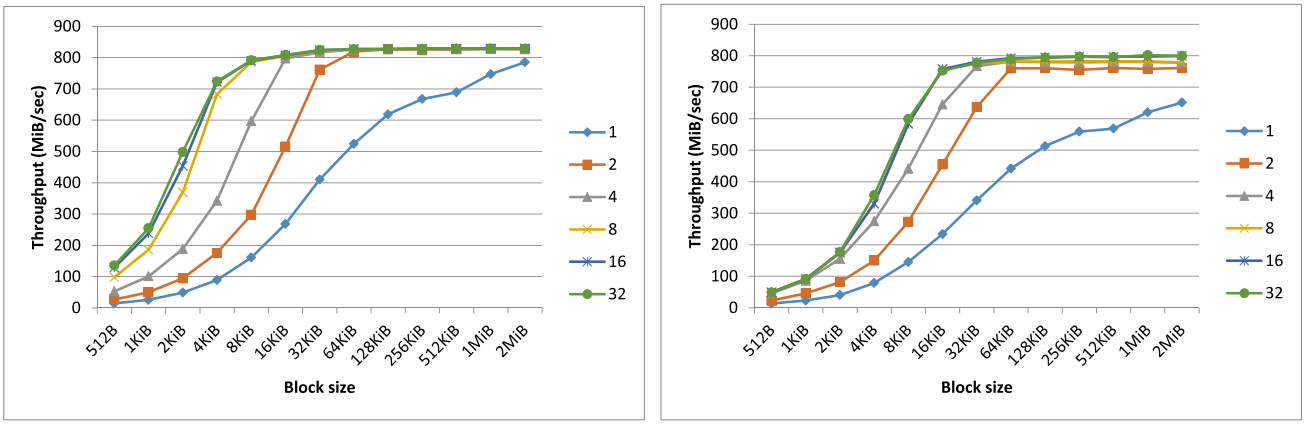
rectFS are shown in **Fig. 9**. Comparing with the directFS, Direct Mode without size improves 0.8% of sequential write performance, and 32.2% of the random write performance. The results of the write performance evaluation of raw device are shown in **Fig. 10**. Comparing the evaluation results of the raw device shown in Fig. 10 and the results of our approach shown in Fig. 8, Direct Mode without size achieves 97.7% of the physical peak performance on average.

5.3 Create Performance Evaluation

We evaluate the performance of creating objects. In the prototype implementation, there are two optimizations, and we evaluate each.

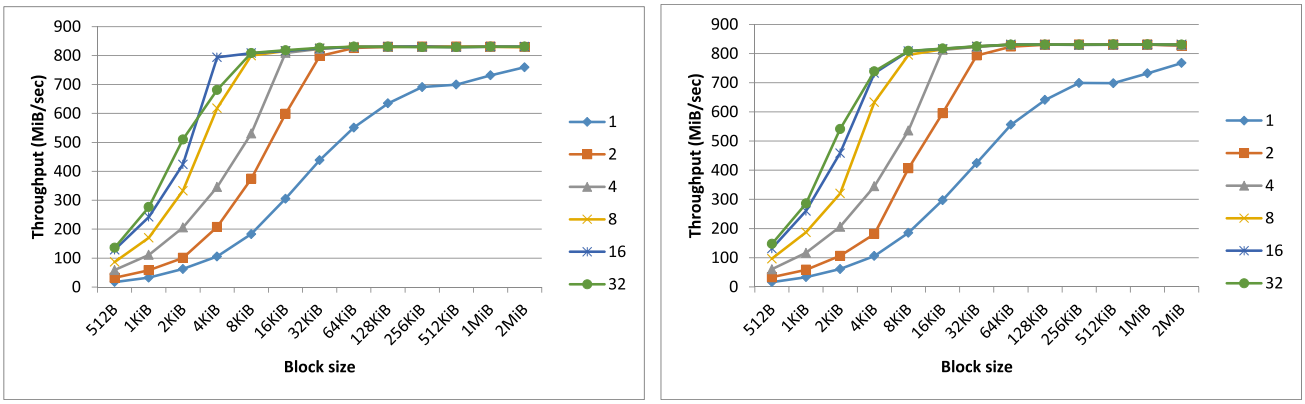
5.3.1 Key-value Stores Performance Evaluation

Firstly, we evaluate the performance of key-value stores. There is considerable difference between key-value store and our object storage. However, today, high performance distributed storage system supports key-value store as an alternative OSD backend. For example, Ceph [12] supports LevelDB [20] as an alternative OSD backend. Therefore, we evaluate the performance of two key-value stores, RocksDB [21] and NVMKV [4]. RocksDB is an embeddable persistent key-value store based on LevelDB. NVMKV is an key-value store using OpenNVM flash primitives. For this evaluation, we call PUT operations with 1 byte data because RocksDB and NVMKV do not support a create operation. We evaluate ten times and calculate the average in each number



(a) Sequential Write (b) Random Write

Fig. 9 Performance evaluation results for directFS with multiple threads.



(a) Sequential Write (b) Random Write

Fig. 10 Performance evaluation results for ioDrive with multiple threads.

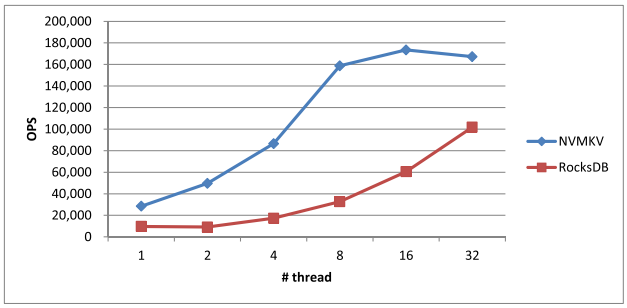


Fig. 11 Put performance of key-value stores by multiple threads.

of worker threads.

The results of the key-value store performance evaluation are shown in **Fig. 11**. In this graph, the horizontal axis shows the number of threads, and the vertical axis shows the number of operations per second. When the number of threads increases, the put performance increases up to the limit.

5.3.2 Optimization Evaluation of Updating Super Region

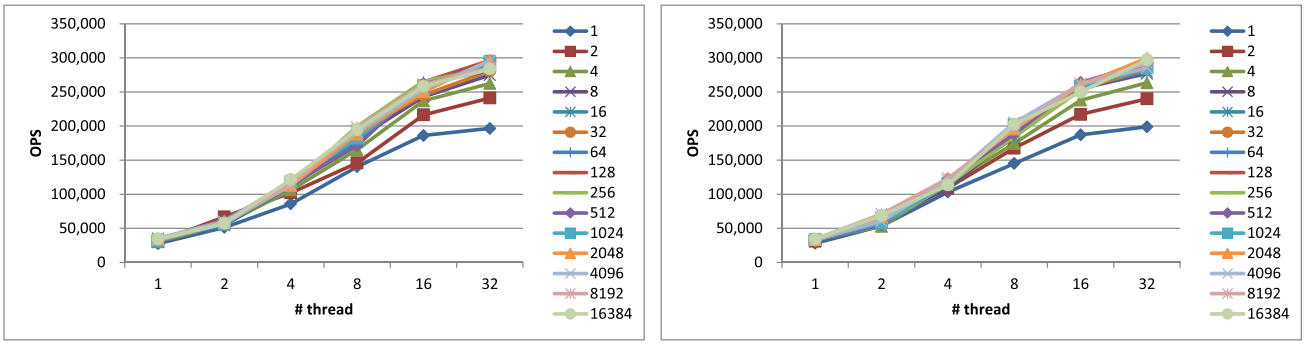
In this evaluation, we assess the create performance by changing the number of worker threads. We also evaluate the optimization effect of updating the super region. In this optimization, there is an update frequency parameter called skip-number. Therefore, we evaluate performance by changing this parameter.

The results are shown in **Fig. 12**. In each graph, the horizontal axis shows the number of threads, and the vertical axis shows the number of operations per second. The lines in Fig. 12 are the parameter called skip-number. When this parameter is 1, the optimization is not enabled. Therefore, the super region is updated at every create request. On the other hand, when this parameter is 1,024, this optimization is enabled, and the super region is updated every 1,024 creates. Comparing the evaluation results of the file system shown in Fig. 1 and the results of our approach shown in Fig. 12, each mode shows proportional performance to the number of worker threads regardless of optimization. This is because there is no lock among threads to create objects. In the case of 32 threads, the performance is improved up to 1.51 times by the optimization. This is because the written data size is reduced when creating an object.

5.3.3 Optimization Evaluation of Initializing Super Block

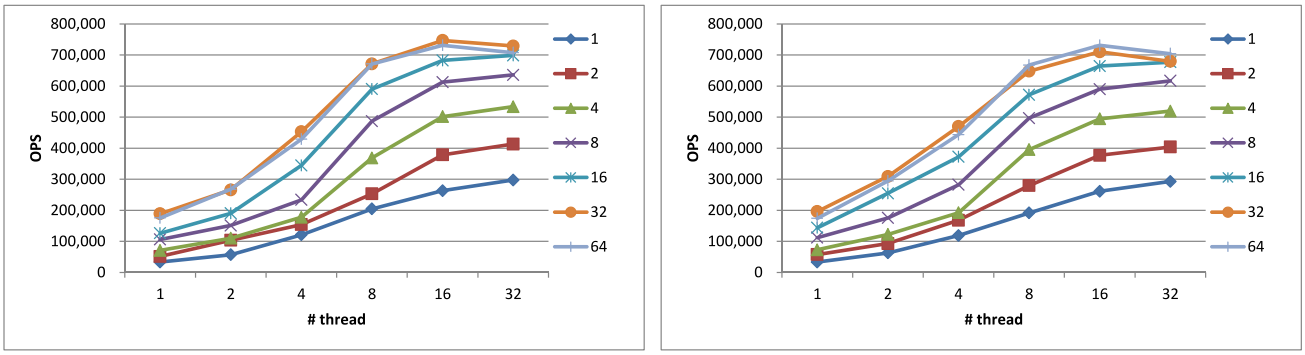
Next, we evaluate the optimization effects of initializing the super block. In addition, in this evaluation, we evaluate the create performance by changing the number of worker threads. In this evaluation, we changed the pre-creation count from 1 to 64, and we set the skip-number to 128.

The results are shown in **Fig. 13**. In each graph, the horizontal axis shows the number of threads, and the vertical axis shows the number of operations per second. Each line in Fig. 13 shows a different number of pre-creating objects. When this parameter is 1, the optimization is not enabled. Therefore, the super block is always initialized at every object-creating request. On the other



(a) Direct Mode (b) Version Mode

Fig. 12 Evaluation results for optimization of updating super region.



(a) Direct Mode (b) Version Mode

Fig. 13 Evaluation results for optimization of initializing super block.

hand, when this parameter is 64, the optimization is enabled, and the super block is initialized every 64 times.

In Fig. 13, the maximum performance is shown at 16 threads for both modes. This is the limit of CPU because the number of threads is 16 in the node. In the case of 16 threads, the performance improves up to 2.84 and 2.80 times in Direct and Version Mode, respectively. We used nvm_atomic_batch_operations to write to ioDrive. nvm_atomic_batch_operations shows better performance when writing many vectors as shown in Fig. 2. Comparing the evaluation results of the file system shown in Fig. 1 and the results of our approach shown in Fig. 13, our approach achieved 740,000 IOPS using 16 threads, which is 12 times better than the performance of DirectFS. Furthermore, comparing the evaluation results of the key-value store shown in Fig. 11 and the results of our approach shown in Fig. 13, our approach is 4.3 times better than NVMKV using 16 threads.

6. Related Work

6.1 File System

POSIX-compliant file systems, such as ext3 [22], ext4 [1], XFS [23], ZFS [2], and Btrfs [13], are often used as object storage for large-scale storage systems. For example, Ceph [12] has used the OSD-based Btrfs, and Lustre [14] has used the OSD based ext4 or ZFS. NILFS2 [24] is a POSIX-compliant log-structured file system (LFS) [25], and provides a continuous snapshot feature. The NILFS2 generates a checkpoint in each synchronous write, which can be saved as a snapshot using the LFS features. Each snapshot can be accessed by mounting a read-only file system without unmounting the NILFS2, which can be used for online backup. There is no restriction on the number of snapshots. The snapshot can be generated provided the capacity of the storage permits. NILFS2 has a clean-up daemon that manages expired checkpoints. The clean up requires complicated processes to re-use the blocks. Our implementation simplifies this process using the trim function from OpenNVM. There are also file systems for flash using LFS, such as F2FS [26], Yaffs [27], and JFFS2 [28]. However, they do not utilize the sparse address space. There are also storage studies that improve the performance using non-volatile memories (NVMs) such as flash, for example, SCMFS [29] and NVMFS [30]. SCMFS designs a file system using a memory and TLB. NVMFS is a file system that improves access performance through a combination NVM and SSD. Direct File System (DFS) [16] is a POSIX-compliant file system designed for Fusion IO ioDrive using the Virtual Storage Layer (VSL) functionalities, such as sparse addressing space and atomic writes. DFS provides a hierarchical namespace, but our approach provides a flat namespace. We can eliminate directory locking and namespace related operations in order to improve performance in multithreaded applications. Moreover, our approach includes the log-structured object layout to manage versions in each object. The Fusion IO DirectFS is a file system based on DFS. ANViL [17] is an advanced storage virtualization for modern non-volatile memory device like Fusion IO ioDrive. It proposes not only a new form of storage virtualization but also the ioctl extension to ext4 to allow file snapshots and deduplication. Because ext4 is one of general file systems, it still has over-head by a hierarchical namespace. Our work also uses new form of storage virtualization, but our work does not support a

hierarchical namespace.

6.2 Object Storage

Object storage is better to use as the backend storage system of distributed file systems because it can eliminate hierarchical namespace overhead. There are several object storages, such as BlobSeer [31] and OBFS [18]. BlobSeer is a distributed data storage, and therefore, it does not write to low-level storage directly. BlobSeer uses a local filesystem to store data. OBFS is local object storage that divides the address space into regions, and stores objects in a region. OBFS manages multiple objects in a region in order to use limited address space efficiently, which requires a hash table to look up an object. It requires not only additional memory space but also expensive linear search when the number of objects increases compared with the hash table size. Object-based SCM [32] provides a file system that uses the OSD interface [33]. Object-based SCM proposes three techniques to manage data in each object. However, each technique does not use the sparse address space, moreover, it requires many indirect references.

7. Conclusion

This paper shows a design of object storage that uses OpenNVM flash primitives for high-performance distributed file systems. Non-blocking design is a key feature for flash devices and storage class memory to improve the IO performance by parallel accesses. The sparse address space allow designs of an array of fixed-size regions that contain a single object, where the object can be addressed by region number, i.e., object ID. Atomic batch write plays an important role in supporting the ACID properties in each write, and several optimizations.

The prototype implementation showed a proportional performance improvement in terms of the number of threads in object creation. In this paper, we proposed two optimizations: one for updating the super region, and another for initializing super blocks. We also evaluated the effect of these optimizations. The optimizations for updating the super region improved creation performance up to 1.51 times using 32 threads. The optimization for initializing super blocks improved the performance up to 2.84 and 2.80 times in Direct and Version Mode, respectively, using 16 threads. The second optimization achieved 740,000 IOPS using 16 threads, which is 12 times better than DirectFS.

Regarding the read and write performance, the prototype implementation achieved 687 MB/s and 830 MB/s for sequential read and write, respectively. In addition, it achieved 97.7% of the physical peak performance on average.

Future work includes designing the delete operation and incorporating this object storage into a high-performance distributed file system.

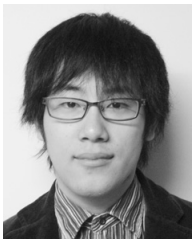
Acknowledgments This work is supported by JST CREST, "System Software for Post Petascale Data Intensive Science" and "EBD: Extreme Big Data — Convergence of Big Data and HPC for Yottabyte Processing".

References

[1] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L.: The New ext4 Filesystem: Current Status and Future Plans, *Proc. 2007 Linux Symposium*, pp.21–33 (2007).
[2] Bonwick, J. and Moore, B.: *ZFS*: *The last word in file systems* (2007).
[3] Lee, E.K. and Thekkath, C.A.: Petal: Distributed Virtual Disks, *SIGOPS Operating Systems Review*, Vol.30, No.5, pp.84–92 (1996).
[4] Marmol, L., Sundararaman, S., Talagala, N. and Rangaswami, R.: NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store, *Proc. 2015 USENIX Conference on Annual Technical Conference*, pp.207–219 (2015).
[5] Nellans, D., Zappe, M., Axboe, J. and Flynn, D.: ptrim ()+ exists (): Exposing new FTL primitives to applications, *2nd Annual Non-Volatile Memory Workshop* (2011).
[6] Ouyang, X., Nellans, D., Wipfel, R., Flynn, D. and Panda, D.K.: Beyond block I/O: Rethinking traditional storage primitives, *Proc. 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp.301–311 (2011).
[7] Saxena, M., Swift, M.M. and Zhang, Y.: Flashtier: a lightweight, consistent and durable storage cache, *Proc. 7th ACM european conference on Computer Systems*, pp.267–280 (2012).
[8] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M. and Wingate, M.: PLFS: A Checkpoint Filesystem for Parallel Applications, *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, pp.21:1–21:12 (2009).
[9] Patil, S. and Gibson, G.: Scale and Concurrency of GIGA+: File System Directories with Millions of Files, *Proc. 9th USENIX Conference on File and Stroage Technologies*, pp.177–190 (2011).
[10] Ren, K., Zheng, Q., Patil, S. and Gibson, G.: IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp.237–248 (2014).
[11] Fusion-io: NVM Primitives Library (2014), available from ⟨http://opennvm.github.io/nvm-primitives-documents/⟩.
[12] Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E. and Maltzahn, C.: Ceph: a scalable, high-performance distributed file system, *Proc. 7th symposium on Operating systems design and implementation*, OSDI '06, pp.307–320 (2006).
[13] Rodeh, O., Bacik, J. and Mason, C.: BTRFS: The Linux B-tree filesystem, *ACM Trans. Storage*, Vol.9, No.3, p.9 (2013).
[14] Koutoupis, P.: The lustre distributed filesystem, *Linux Journal*, Vol.2011, No.210 (2011).
[15] Mesnier, M., Ganger, G.R. and Riedel, E.: Object-based storage, *Communications Magazine, IEEE*, Vol.41, No.8, pp.84–90 (2003).
[16] Josephson, W.K., Bongo, L.A., Li, K. and Flynn, D.: DFS: A File System for Virtualized Flash Storage, *ACM Trans. Storage*, Vol.6, No.3, p.14 (2010).
[17] Weiss, Z., Subramanian, S., Sundararaman, S., Talagala, N., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: ANViL: advanced virtualization for modern non-volatile memory devices, *Proc. 13th USENIX Conference on File and Storage Technologies*, pp.111–118 (2015).
[18] Wang, F., Brandt, S.A., Miller, E.L. and Long, D.D.E.: OBFS: A File System for Object-based Storage Devices, *Proc. 21st IEEE/12th NASA Goddard Conference on Mass Storage systems and Technologies*, pp.283–300 (2004).
[19] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
[20] LevelDB, available from ⟨https://github.com/google/leveldb⟩.
[21] RocksDB, available from ⟨http://rocksdb.org/⟩.
[22] Ts'o, T.Y. and Tweedie, S.: Planned Extensions to the Linux Ext2/Ext3 Filesystem, *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference*, pp.235–243 (2002).
[23] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G.: Scalability in the XFS File System, *Proc. USENIX 1996 Annual Technical Conference*, pp.1–14 (1996).
[24] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S. and Moriai, S.: The Linux Implementation of a Log-structured File System, *SIGOPS Operating Systems Review*, Vol.40, No.3, pp.102–107 (2006).
[25] Rosenblum, M. and Ousterhout, J.K.: The Design and Implementation of a Log-structured File System, *ACM Trans. Computer Systems*, Vol.10, No.1, pp.26–52 (1992).
[26] Lee, C., Sim, D., Hwang, J. and Cho, S.: F2FS: A New File System for Flash Storage, *13th USENIX Conference on File and Storage Technologies*, pp.273–286 (2015).
[27] One, A.: YAFFS: Yet Another Flash File System (2002), available from ⟨http://www.yaffs.net/⟩.
[28] Woodhouse, D.: JFFS: The journalling flash file system, *Ottawa Linux Symposium*, Vol.2001 (2001).
[29] Wu, X. and Reddy, A.L.N.: SCMFS: A File System for Storage Class

Memory, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.39:1–39:11 (2011).
[30] Qiu, S. and Reddy, A.: NVMFS: A hybrid file system for improving random write in nand-flash SSD, *Proc. 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies*, pp.1–5 (2013).
[31] Nicolae, B., Antoniu, G., Bougé, L., Moise, D. and Carpen-Amarie, A.: BlobSeer: Next Generation Data Management for Large Scale Infrastructures, *Journal of Parallel and Distributed Computing*, Vol.71, No.2, pp.168–184 (2011).
[32] Kang, Y., Yang, J. and Miller, E.L.: Object-based SCM: An Efficient Interface for Storage Class Memories, *Proc. 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, pp.1–12 (2011).
[33] Weber, R.: SCSI Object-Based Storage Device Commands (2004).

Fuyumasa Takatsu is a Ph.D. student at University of Tsukuba. He received his M.E. from University of Tsukuba in 2014. His research interest is distributed file system.



Kohei Hiraga is a Ph.D. candidate at University of Tsukuba. He received his M.E. from University of Tsukuba in 2011. Main research interests are grid computing and distributed file system.

Osamu Tatebe received his Ph.D. in computer science from the University of Tokyo in 1997. He worked at Electrotechnical Laboratory (ETL), and National Institute of Advanced Industrial Science and Technology (AIST) until 2006. He is now a professor in Department of Computer Science at University of Tsukuba. He has been a co-chair of Grid File System WG of Open Grid Forum since 2004. His research area is high-performance computing, data-intensive computing, parallel and distributed system software. He is a member of ACM and Japan Society for Industrial and Applied Mathematics (JSIAM).