Design of object storage using OpenNVM for high-performance distributed file system

Fuyumasa Takatsu*, Kohei Hiraga*, and Osamu Tatebe†
*Graduate school of System and Information Engineering,
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan
†Faculty of Engineering, Information and Systems,
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan
Email: takatsu@hpcs.cs.tsukuba.ac.jp, hiraga@hpcs.cs.tsukuba.ac.jp, tatebe@cs.tsukuba.ac.jp

***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 I/O 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. With regard to write performance, it achieves 97.7% of the physical peak performance on average.**

I. 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 I/O 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 [11] and ZFS [1] 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 I/O 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 on Flash Translation Layers (FTLs) [9] [10] [13] [16] [20]. Therefore, it is quite crucial to re-design object storage specifically for flash devices in order to improve the I/O performance of high-performance distributed file systems.

In 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 does not used effectively. Because many core systems will be used in such field and thread concurrency in a node will be more important, big data analysis applications write to multi objects in parallel. Current storage systems have a problem that the file creation performance is not improved by parallel accesses. The more cores are increased, the more important this problem is. We solved this problem by taking advantage of the special features of OpenNVM flash primitives [2] that are proposed standards of 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, thus the underlying object storage does not have to implement them. 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 I/O 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 use 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 create 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

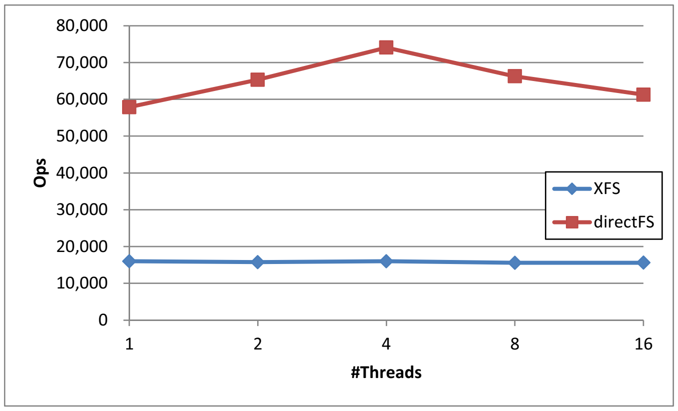


Fig. 1. File creation performance to same directory by multiple threads

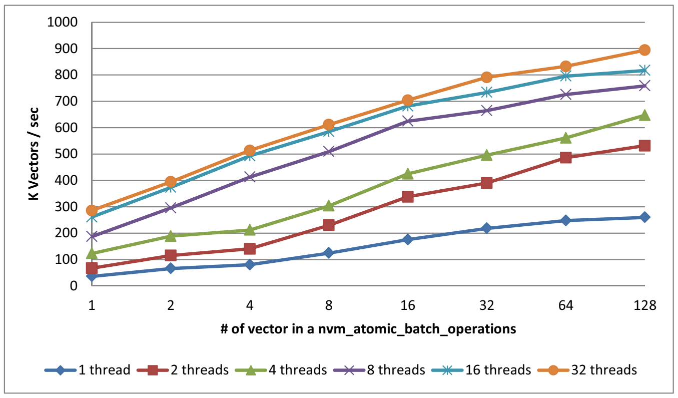


Fig. 2. Performance evaluation results for ioDrive nvm_atomic_batch_operations

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.

II. Background

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

A. 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 [25] uses Btrfs [18] and Lustre [6] uses ext4 [11]/ZFS [1].

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. The file creation performance is not improved by parallel accesses because of lock contention.

Object storage [12] does not have a hierarchical namespace. It only has a flat namespace that can avoid this lock contention problem for the directory entry.

B. 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 144PB 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. Therefore, with a request that writes to multiple blocks, all blocks are written successfully or no block is written. Using atomic batch write, double write is not required to maintain consistency. 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 vectors 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 2GiB data by changing the vector number from one to 128. The block size is 512 bytes. In Figure 2, the horizontal axis shows the number of vectors in an nvm_atomic_batch_operations, and the vertical axis shows the number of written vectors per second. In addition, we changed the number of worker thread from one 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 better performance is shown when writing many vectors. For example, when using 32 worker threads, over 894K vectors/sec. is achieved when writing 128 vectors in one nvm_atomic_batch_operations. This represents over three times better performance compared with writing one vector in one operation.

The file system taking advantage of the special features of ioDrive already exists. It is Direct File System (DFS) [3]. However, DFS provides a hierarchical namespace to cause lock contention because DFS is a POSIX-compliant file system. Therefore, the object storage taking advantage of the special

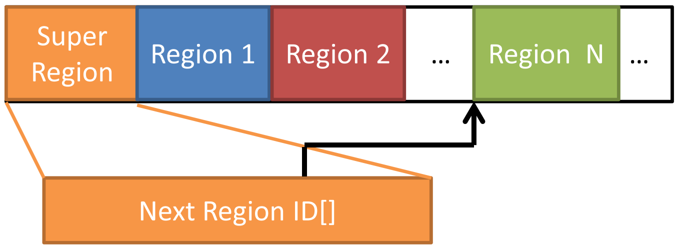


Fig. 3. Dividing sparse address space into regions

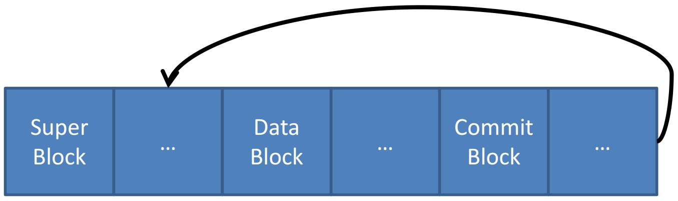


Fig. 4. Data structure of the region in Version Mode

features of ioDrive is quite crucial.

III. 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 [26].

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.

A. 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 Figure 3. Region size is specified as being sufficiently large to store the largest object in a storage system. Because the sparse address space is 64bit or larger, even if the region size is large, the object storage can store enough number of objects. However, this is not necessary. Actually, other object storage system, such as OpenStack Swift, has the limitation of object size. If the Region size is specified small, many objects are supported. As shown in Figure 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. Each of the other regions manages one object. Dividing the address space to fixed-size large regions is similar to OBFS [23]. However, OBFS stores multiple small objects in a region in order to use the limited address space efficiently, 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.

B. Data Management Technique in a Region

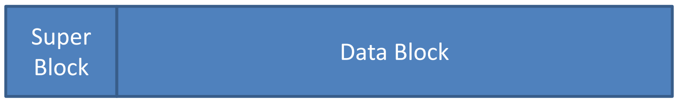
Each region, with the exception of the super region, manages object data. In this section, we provide a brief description on the data management technique for the region.

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 [7] 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 Layout 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.

1) Version Mode: This layout stores all object versions that can help efficient data migration or snapshot.

The data structure is shown in Figure 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.



(a) with the super block

(b) without the super block

Fig. 5. Data structure for region in Direct Mode

2) Direct Mode: This layout stores data directly without any version management.

The data structure is shown in Figure 5. In this figure, there are two layouts: one with the super block to manage the object metadata, such as object size as shown in Figure 5(a), and another without the super block as shown in Figure 5(b).

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

IV. 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 144PB 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 256GB for a region size, which may be larger than the physical capacity, the object storage supports approximately 590K objects.

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

A. Version Mode

Data structure in a region is shown in Figure 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.

B. Direct Mode

When creating an object, a new object ID is assigned within the reserved range for each worker thread without acquiring any lock and 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, data are accessed directly at the specified offset.

C. 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 such overhead, we update the super region every N creations rather than at every creation. This may waste the reserved object ID space when failures occur. We call this parameter (N), ‘Skip-number’.

D. 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 Section II-B, 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 times. This may waste the initialized block space when failures occur, but the space is a few kilobytes at most.

V. Evaluation

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

A. Evaluation environment

The environment for the node we evaluated in our approach is indicated in Table I.

B. Access Performance Evaluation

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

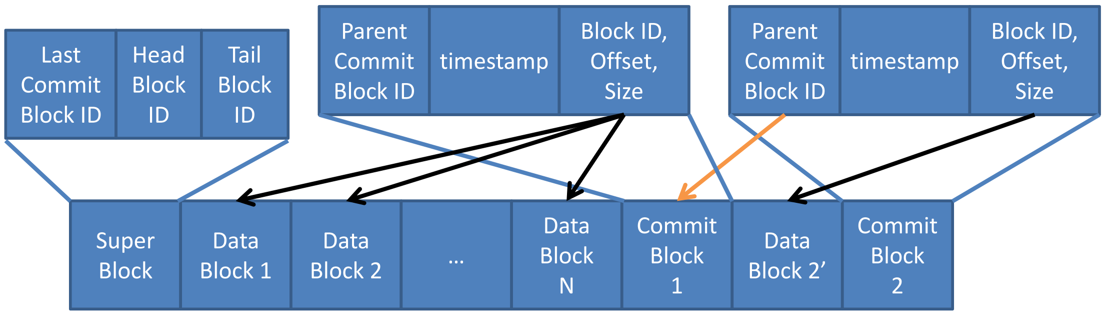


Fig. 6. Detailed data structure of Version Mode

TABLE I
NODE SPECIFICATION

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

1) Read Performance Evaluation: This evaluation measures the read performance in each block size. Before evaluating the read performance, we prepared the write object to be the same block size as the read object. For the read performance evaluation, we read 1 GiB data from the object randomly or sequentially. We evaluated ten times and calculated the average in each block size.

The results of the read performance evaluation are shown in Figure 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 Figures 7(a) and 7(b), there is no graph in Version Mode. This is because we do not assume that the application read the file with many versions.

2) Write Performance Evaluation: In this evaluation, we assess the writting performance in each block size by changing the number of worker threads. Before evaluating the write performance, we prepared the same number of objects as worker threads. For the write performance evaluation, we wrote 1 GiB data to the objects randomly or sequentially. We evaluated ten times and calculated the average in each block size.

The results of the write performance evaluation are shown in Figure 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 Figure 8 shows a different number of worker threads. In this performance evaluation, all mode show proportional performance to the block size. It achieves 830 MB/sec on sequential and random write using Direct Mode without size.

And, we also evaluated the write performance of raw device. The results of the write performance evaluation are shown in Figure 9. Comparing the evaluation results of the raw device shown in Figure 9 and the results of our approach shown in Figure 8, Direct Mode without size achieves 97.7% of the physical peak performance on average.

C. Create Performance Evaluation

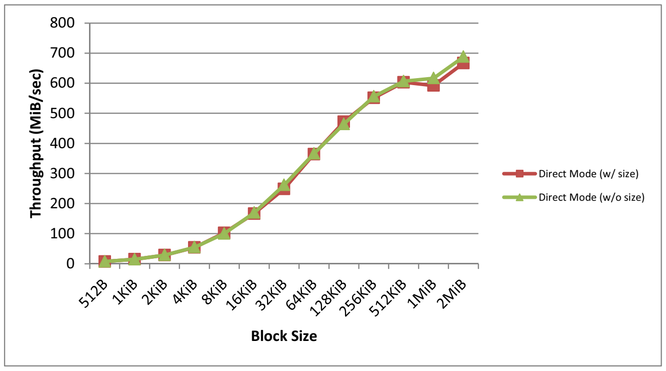
We evaluated the performance of creating objects. In the prototype implementation, there are two optimizations, and we evaluate each.

1) 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.

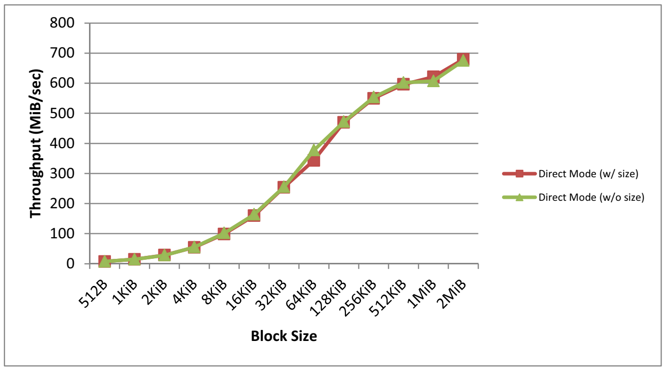
The results are shown in Figure 10. In each graph, the horizontal axis shows the number of threads, and the vertical axis shows the number of operations per second. The lines in Figure 10 are the parameter called skip-number. When this parameter is one, the optimization is not enabled. Therefore, the 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 Figure 1 and the results of our approach shown in Figure 10, 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.

2) 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 one to 64, and we set the skip-number to 128.

The results are shown in Figure 11. In each graph, the horizontal axis shows the number of threads, and the vertical



(a) Sequential Read



(b) Random Read

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

axis shows the number of operations per second. Each line in Figure 11 shows a different number of pre-creating objects. When this parameter is one, the optimization is not enabled. Therefore, the super block is always initialized at every object-creating request. On the other hand, when this parameter is 64, the optimization is enabled, and the super block is initialized every 64 times.

In Figure 11, the maximum performance is shown at 16 threads for both modes. This is the limit by CPU because the number of threads is 16 in the node. In the case of 16 threads, the performance is 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 Figure 2.

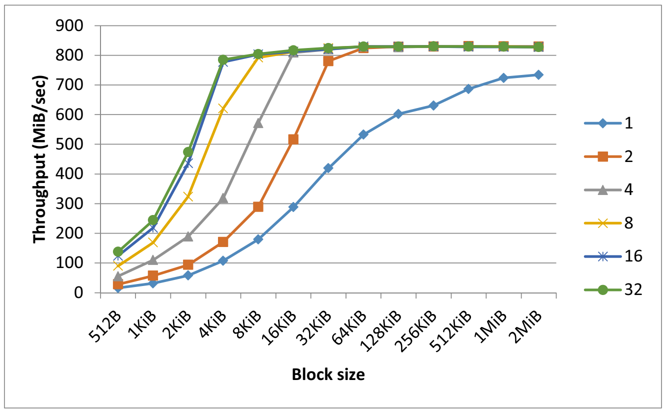
VI. Related work

A. File System

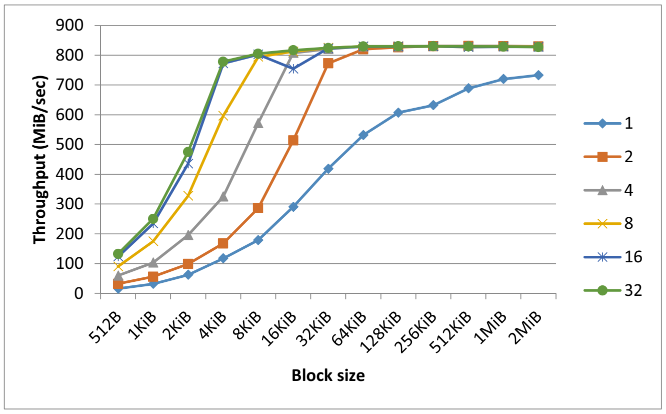
POSIX-compliant file systems, such as ext3 [22], ext4 [11], XFS [21], ZFS [1], and Btrfs [18], are often used as object storage for large-scale storage systems. For example, Ceph [25] has used the OSD-based Btrfs, and Lustre [6] has used the OSD based ext4 or ZFS. NILFS2 [5] is a POSIX-compliant log-structured file system (LFS) [19], 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 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 [8], Yaffs [15], and JFFS2 [27]. 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 [28] and NVMFS [17]. 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) [3] 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 [26] 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.

B. Obejct Storage

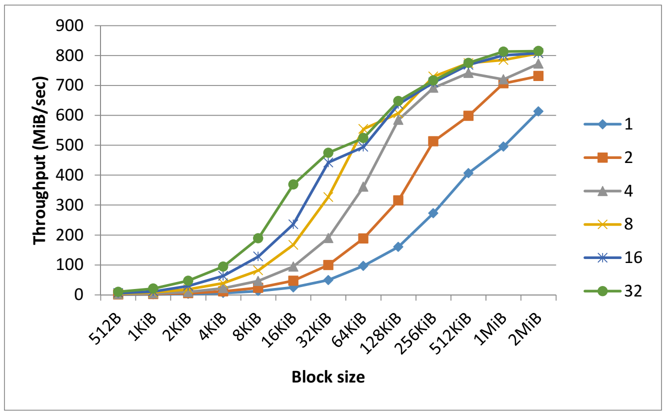
Object storage is better to use as the backend storage system of distributed file systems because it can eliminate hierarchical namespace overhead. There are several object storages, such as BlobSeer [14] and OBFS [23]. BlobSeer is a distributed data storage, and therefore, it does not write to low-level 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



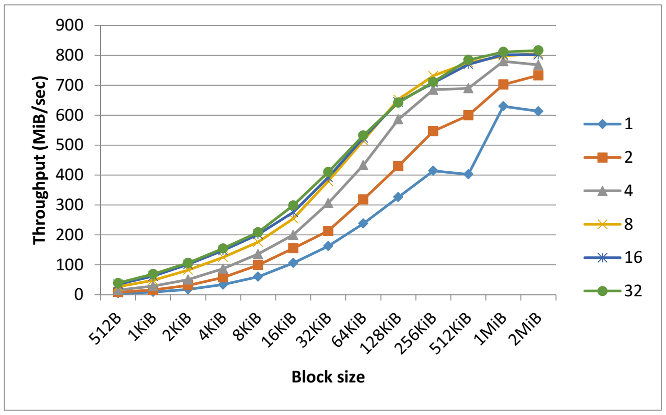
(a) Direct Mode without Size (Sequential Write)



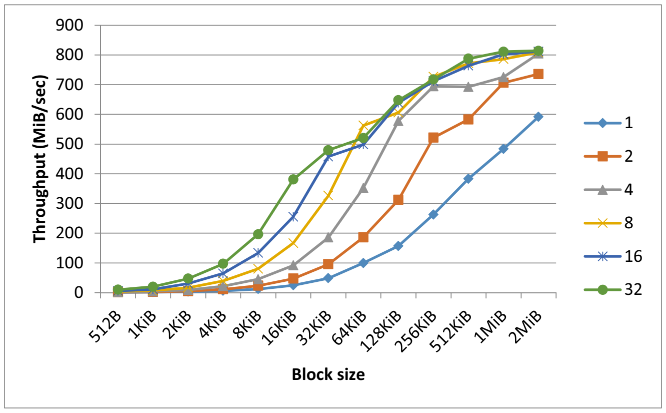
(b) Direct Mode without Size (Random Write)



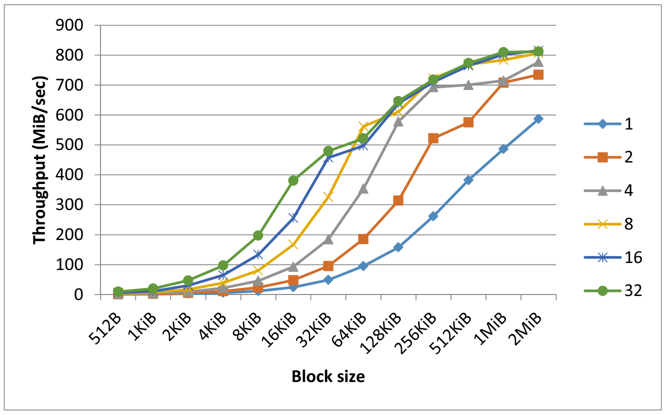
(c) Direct Mode with Size (Sequential Write)



(d) Direct Mode with Size (Random Write)



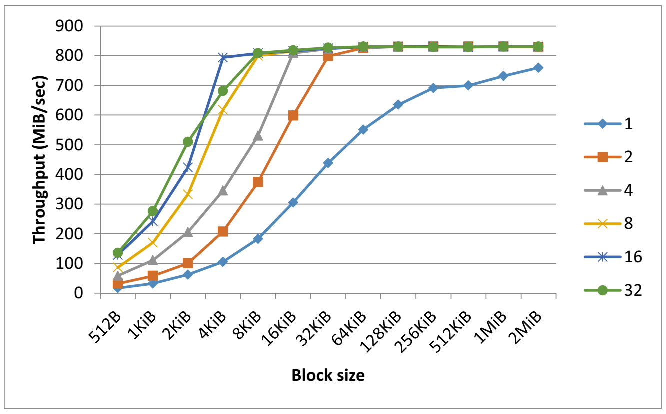
(e) Version Mode (Sequential Write)



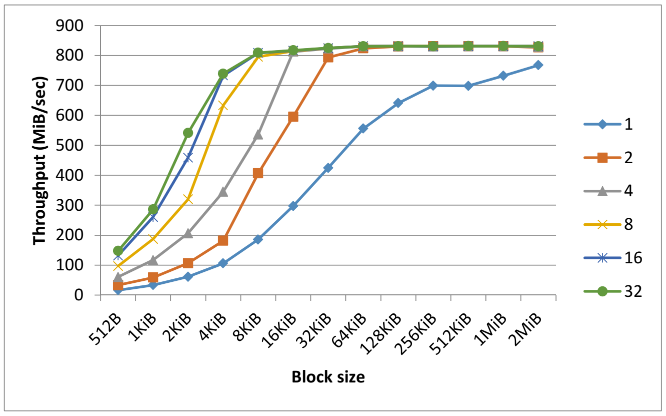
(f) Version Mode (Random Write)

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

compared with the hash table size. Object-based SCM [4] provides a file system that use the OSD interface [24]. 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.

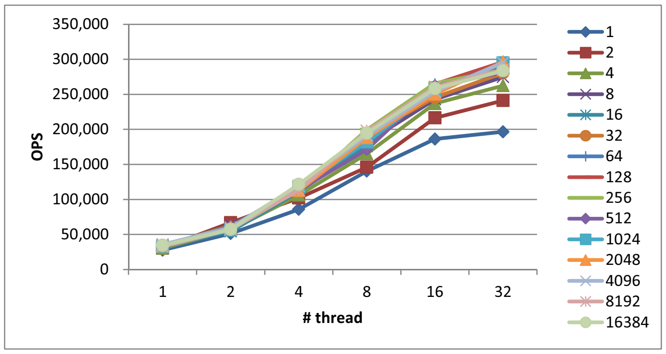


(a) Sequential Write

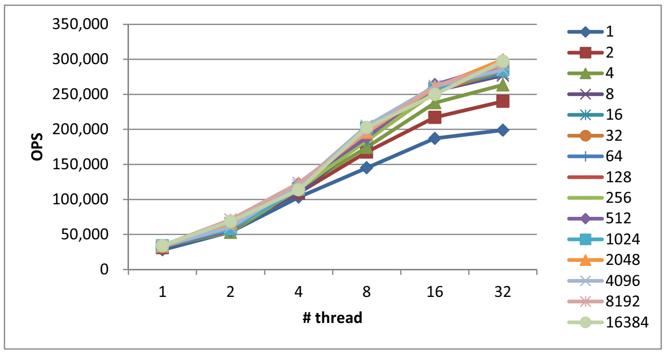


(b) Random Write

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



(a) Direct Mode



(b) Version Mode

Fig. 10. Evaluation results for optimization of updating super region

VII. Conclusion

This paper designed 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 I/O 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.

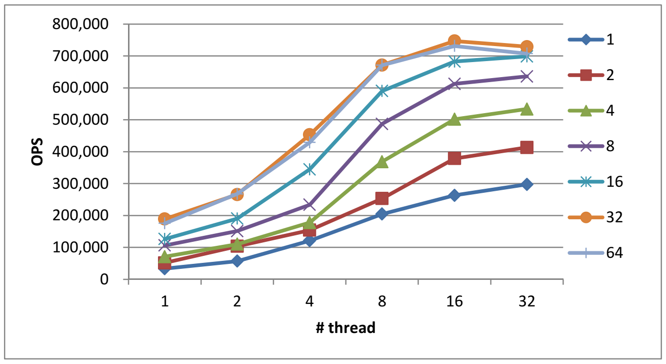
Future work includes incorporating this object storage into a high-performance distributed file system.

Acknowledgments

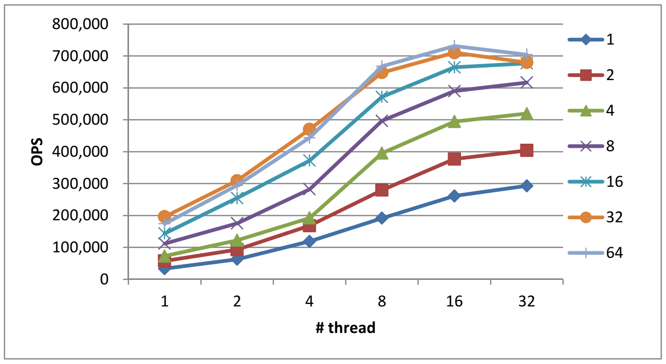
This work is supported by JST CREST,“ System Software for Post Petascale Data Intensive Science ”and “ EBD: Extreme Big Data Convergence of Big Data and HPC for Yottabyte Processing ”.

References

[1] J. Bonwick and B. Moore, “ZFS: The last word in file systems,” 2007.



(a) Direct Mode



(b) Version Mode

Fig. 11. Evaluation results for optimization of initializing super block

[2] Fusion-io, “NVM Primitives Library,” 2014. [Online]. Available: http://opennvm.github.io/nvm-primitives-documents/

[3] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, “DFS: A File System for Virtualized Flash Storage,” *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 14, 2010.

[4] Y. Kang, J. Yang, and E. L. Miller, “Object-based SCM: An Efficient Interface for Storage Class Memories,” in *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, ser. MSST ’11, Washington, DC, USA, 2011, pp. 1–12.

[5] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, “The Linux Implementation of a Log-structured File System,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 102–107, Jul. 2006.

[6] P. Koutoupis, “The lustre distributed filesystem,” *Linux J.*, vol. 2011, no. 210, Oct. 2011.

[7] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[8] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2FS: A New File System for Flash Storage,” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, Feb. 2015, pp. 273–286.

[9] E. K. Lee and C. A. Thekkath, “Petal: Distributed Virtual Disks,” *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 84–92, Sep. 1996.

[10] L. Mármol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, “NVMKV: a scalable and lightweight flash aware key-value store,” in *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, 2014, pp. 8–8.

[11] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The New ext4 Filesystem: Current Status and Future Plans,” in *Proceedings of the 2007 Linux Symposium*, Jun. 2007, pp. 21–33.

[12] M. Mesnier, G. R. Ganger, and E. Riedel, “Object-based storage,” *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.

[13] D. Nellans, M. Zappe, J. Axboe, and D. Flynn, “ptrim ()+ exists (): Exposing new FTL primitives to applications,” in *2nd Annual Non-Volatile Memory Workshop*, 2011.

[14] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, “BlobSeer: Next Generation Data Management for Large Scale Infrastructures,” *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 168–184, Feb. 2011.

[15] A. One, “YAFFS: Yet Another Flash File System,” 2002. [Online]. Available: http://www.yaffs.net/

[16] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, “Beyond block I/O: Rethinking traditional storage primitives,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 301–311.

[17] S. Qiu and A. Reddy, “NVMFS: A hybrid file system for improving random write in nand-flash SSD,” in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, May 2013, pp. 1–5.

[18] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree filesystem,” *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[19] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-structured File System,” *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[20] M. Saxena, M. M. Swift, and Y. Zhang, “Flashtier: a lightweight, consistent and durable storage cache,” in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 267–280.

[21] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the XFS File System.” in *USENIX Annual Technical Conference*, vol. 15, 1996.

[22] T. Y. Ts’o and S. Tweedie, “Planned Extensions to the Linux Ext2/Ext3 Filesystem,” in *USENIX Annual Technical Conference, FREENIX Track’02*, 2002, pp. 235–243.

[23] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long, “OBFS: A File System for Object-based Storage Devices,” in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage systems and Technologies, College Park, MD*, 2004, pp. 283–300.

[24] R. Weber, “SCSI Object-Based Storage Device Commands,” 2004.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: a scalable, high-performance distributed file system,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI ’06, Berkeley, CA, USA, 2006, pp. 307–320.

[26] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “ANViL: advanced virtualization for modern non-volatile memory devices,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 111–118.

[27] D. Woodhouse, “JFFS: The journalling flash file system,” in *Ottawa Linux Symposium*, vol. 2001, 2001.

[28] X. Wu and A. L. N. Reddy, “SCMFS: A File System for Storage Class Memory,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11, New York, NY, USA, 2011, pp. 39:1–39:11.