Towards Unaligned Writes Optimization in Cloud Storage With High-Performance SSDs

Jiwu Shu, *Fellow, IEEE*, Fei Li, Siyang Li, and Youyou Lu

Abstract—NVMe SSDs provide extremely high performance and have been widely deployed in distributed object storage systems in data centers. However, we observe that there are still severe performance degradation and write amplification under the unaligned writes scenario with high-performance SSDs. In this article, we identify that the RMW sequence which is used to handle the unaligned writes incurs severe overhead in the data path. Besides, unaligned writes incur additional metadata management overhead in the block map table. To address these problems, we propose an object-based device system named NVStore to optimize the unaligned writes in cloud storage with NVMe SSDs. NVStore provides a *Flexible Cache Management* to reduce the RMW operations while supporting lazy page sync and ensuring data consistency. To optimize the metadata management, NVStore proposes a *KV Affinity Metadata Management* which co-designs the block map and key-value store to provides a flattened and decoupled metadata management. Evaluations show that NVStore provides at most 6.11× bandwidth of BlueStore in the cluster. Besides, NVStore can reduce at most 94.7 percent of the write traffic from metadata under unaligned writes compared to BlueStore and achieves smaller data write traffic which is about 50 percent of BlueStore and 65.7 percent of FileStore.

Index Terms—Distributed system, object storage, unaligned writes, solid state drives

1 Introduction

In recent years, large-scale data centers host a large number of applications that serve several million users. To meet this requirement, the scale-out storage architecture, distributed object storage, has been widely used in modern storage systems and becomes the underlying storage layer. In the supercomputing center, the well-known cluster file systems (e.g., GPFS [1] and Lustre [2]) use object storage to store both data and metadata. In the cloud platform, Amazon Simple Storage Service (S3) and Microsoft Azure [3] use object storage to support various applications. The hybrid storage system, Ceph [4], also uses object storage to construct various storage systems. Therefore, improving the performance of object storage is extremely salient for improving data center access efficiency.

Because the object storage system is on the underlying layer and supports various systems, its I/O characteristics are rich and varied. The object-based storage devices (OSDs) not only deal with aligned I/Os from specific applications but also tend to handle more unaligned I/Os from small file accesses. In OFSS [5], Lu *et al.* collect the write statistics of iBench [6] (iPhoto, iPages) and LASR [7] (LASR-1, LASR-2, LASR-3) and find that nearly 50 to 90 percent of writes are unaligned.

In the hard disk drive (HDD) ear, unaligned I/Os always decrease storage system performance because of incurring extra small and random I/O requests [8] which are unfriendly to the HDD's mechanical properties. With the growing requirements in I/O performance, Non-Volatile Memory Express (NVMe) based Solid State Drives (SSDs), which can archive high throughput, low latency, and good random access performance [9], are involved in the object storage system. However, the storage system which equips with the newest SSDs also faces the unaligned write (UW) [8] problem. Although NVMe SSDs could achieve high performance in aligned I/O evaluations (e.g., benchmarks like `fio` [10], `iozone` [11], database [9], [12] and tier [13]), there are still severe performance degradation in the unaligned I/O tests (e.g., `filebench` [14], log system, cloud file system, mail server). Fig. 1 shows that the bandwidths tested in CephFS [4], Lustre and GlusterFS [15] with aligned I/Os is about 4∼ 5× of those with unaligned I/Os. Previous works focus on optimizing the unaligned I/O performance for specified applications (e.g., database [12], client cache [16], and log system [17], [18]). However, in the cloud scenario, the underlying object storage are serving an exceedingly broad class of applications. It is considerable to address the unaligned write problem in the object-based storage device (OSD) system level.

We propose NVStore, an OSD system which provides excellent performance in unaligned writes with NVMe-based SSDs. Our designs are based on the observation that some self-caching applications (e.g., database) will send I/O requests directly to the storage system bypassing the page cache layer which will buffer and merge the I/O requests. In this manner, the unaligned writes from applications should be handled by the application cache and the storage system. Based on this, we are able to make a more flexible cache design according to application characteristics.

• *The authors are with the Tsinghua University, Beijing 100084, China. E-mail: {shujw, lisiyang, luyouyou}@tsinghua.edu.cn, lf17@mails.tsinghua.edu.cn.*

Manuscript received 24 Oct. 2019; revised 3 May 2020; accepted 22 June 2020. Date of publication 2 July 2020; date of current version 20 July 2020. (Corresponding author: Fei Li.) Recommended for acceptance by L. Wang. Digital Object Identifier no. 10.1109/TPDS.2020.3006655

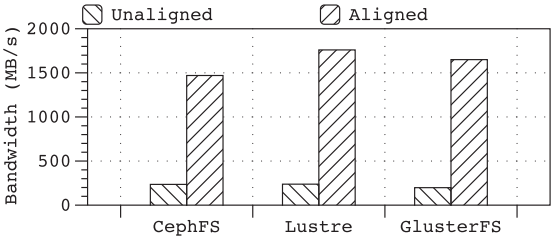


Fig. 1. Comparison of the performance of different distributed file systems (3× nodes with 6× Intel 750) in aligned I/Os and unaligned I/Os. The aligned I/O cases are tested by `iozone`'s 4KB aligned write. The unaligned I/O cases are tested by `filebench`'s fileserver which limited the page size with 4KB in average.

Our major contributions are as follows:

- We develop an OSD system, NVStore, to fully exploit the high-performance SSDs under unaligned writes while prolonging the lifetime of SSDs.
- In NVStore, we propose a *Flexible Cache Management* mechanism to reduce the write amplification and unnecessary I/O requests from unaligned writes while ensuring the data consistency. Moreover, we provide a *Key-Value (KV) Affinity Metadata Management* mechanism to optimize the KV store performance and reduce the overhead from metadata.
- Our system can reduce the write amplification from both metadata and data compared with the filesystem-based OSD system, FileStore [19] and the state-of-art OSD system, BlueStore [20]. The results show that NVStore reduces at most 94.7 percent of the write traffic from metadata under unaligned writes compared to BlueStore and achieves smaller data write traffic which is about 50 percent of BlueStore and 65.7 percent of FileStore.
- Our system could significantly promote the unaligned write performance in both benchmark tests and real workload tests. The results show that NVStore achieves $1.11\sim 3.00\times$ bandwidth of BlueStore, $1.05\sim 1.75\times$ bandwidth of FileStore in benchmark tests and $2.03\sim 6.11\times$ bandwidth of BlueStore, $1.99\sim 3.06\times$ bandwidth of FileStore in real workload tests.

We organize the rest of this paper as follows. Section 2 introduces the definition and types of unaligned writes, and then analyses the overhead incurred by unaligned writes in the data path and block map table. Section 3 describes the designs and implementation of NVStore, including the *Flexible Cache Management* and *KV Affinity Metadata Management*, and then discusses the compatibility and limitations. Section 4 shows the evaluations of NVStore. Section 5 gives the related works. Section 6 concludes the paper.

2 Motivation

2.1 Types of Unaligned Writes

As block devices, both traditional HDDs or SSDs and NVMe-based SSDs defines a minimal I/O unit, called sector, which is 512 bytes or larger according to the device model. The devices do not support partial sector read or write in the block I/O interface. Since there are limits to the number of device addresses, that an operating system (OS) can address. Modern OS uses block as the minimal data unit. A block could be a sector or several sectors. When OS sends requests to the device, it specifies the block offset, number of blocks and points to data buffer. However, an application may make write requests with address or size not aligned to the block size (`bsize`) thus engendering the unaligned write problem.

TABLE 1
Types of User Writes

$o \bmod b \neq 0$	$o \bmod b + l < b$	$(o+l) \bmod b \neq 0$	Type
0	0	0	aligned
1	0	0	within block
1	0	1	cross blocks
0	0	1	within block
-	1	-	within block

The OS storage systems (e.g., Ext4 [21], F2FS [22], BlueStore, FileStore and so on.) always divide a write buffer into aligned parts and unaligned parts. Table 1 shows different types of writes. For a user requests, o is the byte-aligned offset, l is the byte-aligned write length, b is `bsize`. When the location of the unaligned part is only in the header or tail of the write buffer (i.e., either $o \bmod b$ or $(o+l) \bmod b$ is not equal to zero), we call it as unaligned write within block. When the locations of the unaligned part are on both sides of the buffer (i.e., both $o \bmod b$ or $(o+l) \bmod b$ are not equal to zero), we call it as unaligned write cross blocks. In addition, if the header and tail are in the same block and the size of a write buffer is less than the block size (i.e., $o \bmod b + l < b$), it is also treated as unaligned write within block.

2.2 Overhead in Data Path

To deal with the unaligned application write requests, OS performs the read-modify-write (RMW) sequence. First, blocks which is written partially are read. Then, the data read is merged into the write buffer. At last, the partially written blocks will be updated with the merged data. In this process, OS transforms the unaligned write into aligned write at the cost of extra operations. Fig. 2 shows the data path in unaligned writes. The first case is the unaligned write within block, a single-block RMW operation is performed. The second case is the unaligned write cross blocks. In this case, a two-block RMW operation is performed. What's more, the RMW sequence should be performed only step-by-step.

The situation is more complicated in SSDs. The basic unit of read/write in flash based SSDs is flash page, which is about 4KB or 8KB according to the flash media architecture. A flash page should be erased before it is written again. If the OS storage system adopts a small `bsize`, such as 512 bytes, an aligned write in the OS may lead to an unaligned write inside the SSDs. In this case, an unaligned write in the

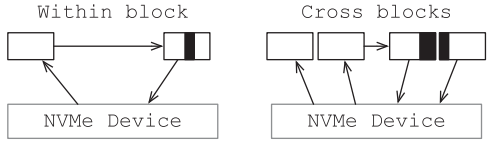


Fig. 2. Data path of unaligned writes in RMW sequence.

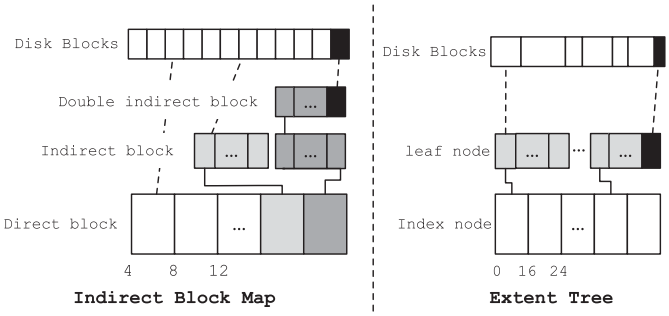


Fig. 3. Types of index tables.

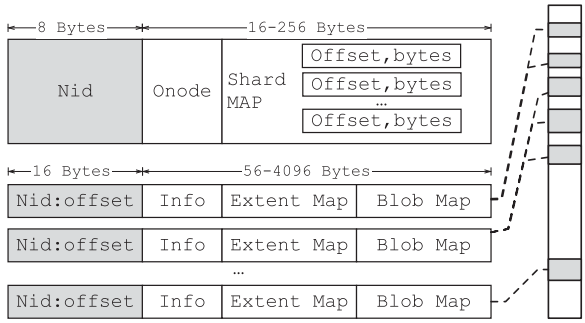


Fig. 4. BlueStore's Block Map in key-value store. The gray block is the key and the white block is value

OS may incur RMW operation both in the OS layer and the SSD device layer. For this reason, some flash-based file systems prefer to set the `bsize` to the size of a flash page. However, the RMW still degrades the SSDs performance and reduces the flash lifetime. On one hand, a single-block RMW incurs one block read and one block write. Therefore, the unaligned write incurs extra I/O requests. Besides, the mixed read/write pattern in RWM further decreases the NVMe device's write performance. On the other hand, the merge operations transfer the small size data (several bytes) to a relatively large block (4 or 8 kilobytes), this leads to write amplification in SSDs thus reducing the flash lifetime.

So, there is still a large headroom to exploit the performance of NVMe-based SSD by reducing the RMW operation in the data path.

2.3 Overhead in Block Map Table

2.3.1 Traditional File System

In a file system, files are divided into data blocks. The inode is used to store the metadata or directory of a file, and the block map table is used to index the device blocks. To support the big size file (i.e., Gigabytes to Terabytes), common file systems use two types of map tables to address the file blocks, the indirect block map (IM) [23] and the extent tree (ET) [21] as in Fig. 3. In IM, the inode points to the first few data blocks, often call direct blocks. For big files, the inode also points to an indirect block, which points to disk blocks. If it is still not enough to store the file, the inode will point to a double indirect block, which points to some indirect blocks, which point to disk blocks. In ET, it uses `Extent`, which contains a block offset and number of blocks, to describe several consecutive blocks. For small files, the inode will points to the `Extents` directly. For big files, the `Extents` are organized using B^+ tree [21]. These two methods are widely used in file systems (Ext3 [24], F2FS and XFS [25] use IM, while Ext4 and Btrfs [26] use ET).

For the IM mechanism, when a file data is updated, its direct blocks or indirect blocks should be updated. Then, the index structures such as inode, inode map are also updated recursively. This is called wandering tree problem or update-to-root problem. Since the RMW operation will rewrite the unmodified data in a partially written page, unaligned writes incur additional metadata updates and aggravate the wandering tree problem. Although F2FS [22] solves this problem by introducing NAT (Node Address Table) mechanism, it also requires an in-place-update operation which is unfriendly to flash media and requires further I/O remapping operations in FTL layer.

For the ET mechanism, an aligned write only add or change one item of ET's leaf node. An unaligned write divides an I/O into the aligned part and unaligned part, thus inserting two (within block) or three (cross blocks) items into leaf nodes. For the ET mechanism, the more fragment items which are inserted into the leaf node, the more split operations will occur in the leaf node. The split operation leads to tow leaf nodes' reorganization and index table updating, and these involve additional three metadata pages rewritten.

The methods mentioned above are lack of optimizations for the unaligned writes, and thus, increase the write amplification and decrease the write performance.

2.3.2 Key-Value Based Metadata Management

To migrate the write amplification problem from updating metadata block in the file system, some file systems (e.g., TableFS [27], IndexFS [28] and BatchFS [29]) and object storage device (e.g., BlueStore) use key-value (KV) store to store the metadata. Most of the KV store in these systems are organized in log-structured merge (LSM) tree structure, the data are stored as write-ahead-log (WAL) which sequentially write data into the storage device to fully utilize the available device bandwidth. However, the write amplification problem caused by unaligned writes remains.

Fig. 4 shows the example of the block map in the KV store of BlueStore. BlueStore uses the block map to store the extent tree structure. `Shard Map` is used to store the index node of extent tree. Each object in BlueStore is stored as a KV pair, which uses `Nid` as the key and uses the object node metadata (`onode`) and all the index nodes (`Shard Map`) as the value. For each sub-index entry, `Nid` and `offset` in `Shard Map` is used to retrieve the `Extent Map` and `Blob Map`. `Extent Map` is used to store the leaf node of the extent tree. `Blob Map` stores the mapping of logical address to physical address. In this manner, the metadata update only requires KV operations rather than page-level data syncing as in filesystems. However, a small change to a leaf node still requires updating a whole KV pair. For an unaligned write operation, BlueStore will modify one or two KV pairs. It means each write operation will write additional 56-8192 bytes data in the KV store.

In this paper, our goal is to reduce the write amplification and extra I/O requests from unaligned writes. We propose an object-based storage device (OSD) system, NVStore, to provide excellent data access performance in unaligned writes for the distributed object storage system.

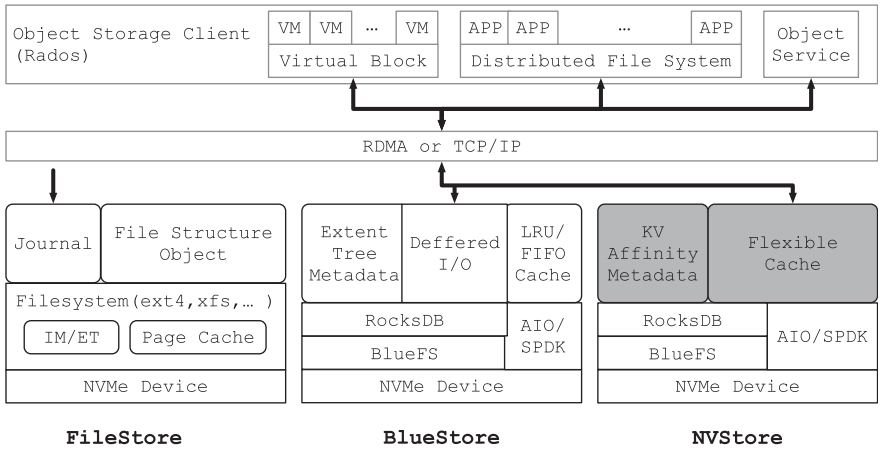


Fig. 5. The architecture of filestore, bluestore, and NVStore.

3 Design and Implementation

3.1 Architecture

NVStore is an OSD system, and it could be used in the existing object storage system, Ceph, to improve the unaligned write performance for high-performance SSD while extending the lifetime of flash media. Fig. 5 presents the overall architecture of NVStore and the differences from the current OSD systems(e.g., FileStore and BlueStore).

Since NVStore is at the lowest layer of Ceph, it receives all the data from the the upper system (e.g., the virtual machine I/O from the virtual block layer, the application I/O from the distributed file system or the application I/O from the object store service) through remote direct memory access (RDMA) or TCP/IP.

NVStore functions based on two types of storage systems, the KV store and NVMe-based raw device. The metadata in NVStore is entirely stored in KV store and the data is stored both in KV store and raw NVMe SSD device. NVStore reuses BlueStore's user-space file system, BlueFS [20], and the key-value store, RocksDB [30]. Instead of using `Extent tree` to store the block map as in BlueStore, NVStore provides a KV affinity metadata organization mechanism to reduce the write amplification from unaligned writes. What's more, unlike BlueStore which only uses KV store to record the index tables, NVStore uses a flexible cache mechanism which cooperates with KV store to accelerate the unaligned write performance while reducing write amplification and ensuring the data consistency. Compared with FileStore which delegates the block map and cache management to the local filesystem (e.g., Ext4, XFS), both NVStore and BlueStore use user-space metadata and cache management and directly send the final I/O requests to kernel's AIO block driver [31] or Intel's SPDK [32].

Besides, NVStore is compatible with existing systems and provides the same interfaces to Ceph's Rados [20] layer.

3.2 Flexible Cache Management

Instead of using kernel's page cache (e.g., FileStore) or user-space LRU (Least Recently Used) or FIFO (First In First Out) cache (e.g., BlueStore), NVStore proposes a flexible cache management mechanism to optimize the unaligned I/O by introducing the fragment page. Fig. 6 shows the architecture of the *Flexible Cache Management*. NVStore proposes three types of pages and transforms these pages under different I/O operations. Besides, to enforce the data consistency, NVStore uses the KV store to store data from unaligned writes.

3.2.1 Fine-Grained Page Types

The three types of pages are the fragment page (FP), the clean page (CP) and the dirty page (DP). In NVStore, all types of pages use FIFO or LRU algorithm when cache replacement is required. The clean page is read-only, and it will update when aligned write and page read happens. The dirty page is modified page, and it will be synced to the disk when page replacing or system reloading happens. Unlike the above-mentioned pages, the FP is organized as a list table. Each FP contains an ordered list of items which each records the page offset, data length and the data of an unaligned write item. In FP, the latest unaligned write item is always inserted to the end of the list. Fig. 6 and Table 2 show the page transformation in different operations, Fig. 7 shows the fragment page merge process, we will discuss these processes in the following sections.

3.2.2 RMW Less I/O Operation

When data writes come, the traditional page cache mechanism will write data to the memory cache and syncs the cache using background threads. In this manner, those unsynchronized data pages become dirty pages. In contrast, NVStore

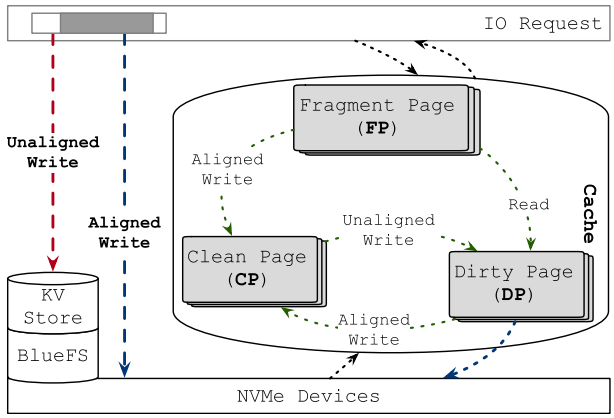


Fig. 6. The flexible cache management.

TABLE 2
Page Transformation

OP	Hit DP	Hit FP	Hit CP	Miss
AW	CP	CP	×	CP
UW	×	×	DP	FP
Read	×	DP	×	CP
Sync	CP	CP	×	×

× *means no transformation happens. AW and UW means aligned write and unaligned write operation.*

would not directly create dirty pages thus reducing the frequency of page replacement and data sync operations.

In NVStore, if an unaligned write is missed in the cache, it creates a FP and then adds the unaligned write item to the FP's list table. If an unaligned write is in CP, it will merge the unaligned data with CP and transform CP to DP. If an unaligned write is in DP, it will merge the unaligned write item to the existing DP. If an unaligned write is in FP, it will add an unaligned item to the end of a FP's list table. By using these methods, an unaligned write will not perform the RMW sequence and only need to write data to the cache. For aligned data, NVStore will write data to the page. Meanwhile, NVStore will directly write data to the NVMe SSD device by using kernel's AIO or Intel's SPDK. If the aligned write hit in the CP, it will update the CP itself. If the aligned write hit in the FP or DP, it will transform the FP or DP to CP. If the aligned write is missed in the cache, it will add a new page to the CP. Besides, if a read operation hits in a FP, it will read the page data from the NVMe device and merge it with the FP data while transferring FP to DP.

3.2.3 Data Consistency

NVStore ensures the data consistency for both aligned writes and unaligned writes. For aligned writes, it uses AIO or SPDK and copy-on-write to ensure that all data will be persisted to the back-end NVMe SSD device. This is similar to BlueFS, which also adopts copy-on-write for write requests. For unaligned writes, NVStore uses the KV store to record the unaligned data before data syncing. As Fig. 8a shows, NVStore defines the KV format of the unaligned write item. It uses the object's `uuid`, the `offset` of the page, the `order` of the unaligned item as the key and uses the offset inner a page, called `inoff` and the `data` itself as the value. This is similar to FileStore, which records all write requests in a journal file before performing the write operations. When the system crash happens, FileStore will scan the journal file for data recovery, and NVStore will use the data from KV store to recover and reorganize the DP and FP. In this way, NVStore guarantees the data consistency in the object storage level as BlueStore and FileStore do.

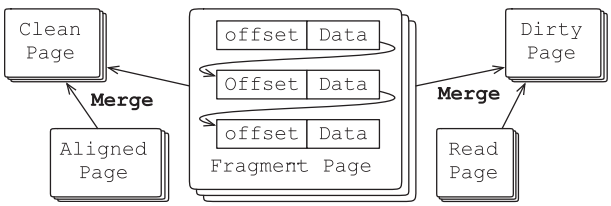


Fig. 7. Fragment page merging process.

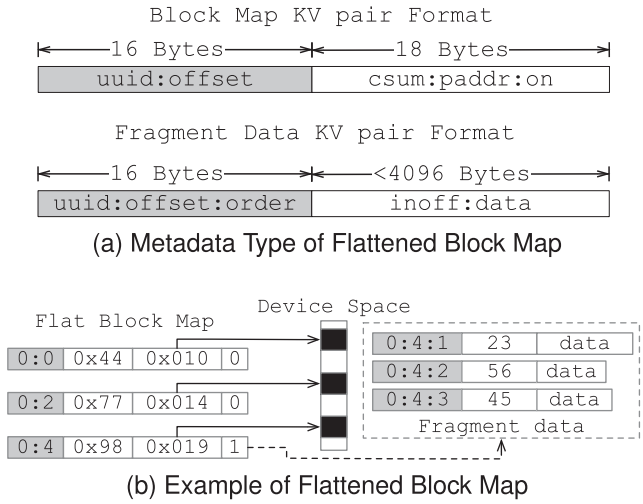


Fig. 8. Flattened Block Map. The gray block is the key field, the white block is the value field.

3.2.4 Lazy Page Sync

In NVStore, all the pages can be discarded without being synced because of the guarantee of data consistency. But NVStore still perform additional page replacement mechanism to avoid excessive data fragmentation from unaligned writes in KV store.

The DP and FP collection procedure will be executed by a background thread when the system I/O is idle. In the DP collection procedure, NVStore will sync the DP data to the disk, and then discard the fragment data of this page in KV store through the prefix operation in RocksDB thus reducing the KV space. When a DP is synced to the disk, it will be transformed to a CP. In the FP collection procedure, NVStore will read the page data from the NVMe device and merge it with the FP, and the FP will be transformed to DP. Then, NVStore performs DP collection as mentioned above.

3.2.5 Benefits

In NVStore, each unaligned write only needs to write one key-value pair to the KV store. The read, merge operations in RMW sequence are required only when the page is read by the client. This mechanism reduces the frequency of performing the RMW sequence for some applications which make unaligned write requests, and thus, reduces the mixed read/write operations and write amplifications caused by the RMW sequence. Moreover, since the KV store is used to record the unaligned write, NVStore could accelerate the unaligned write performance by reducing page sync operations while ensuring the data consistency.

3.3 KV Affinity Metadata Management

Metadata is used to manage and index the data in the storage system, and most storage systems have their accesses been dominated by metadata operations. In recent years, many storage systems choose to store metadata in the KV store. KV stores are more gifted in efficiently managing the small fragment data than file systems, for they aggregate small fragment data into aligned blocks with special data structures (e.g., LSM tree or B^+ tree), which are more friendly to both HDD and SSD. However, as illustrated in

Sectin 2.3, the key-value based metadata management will face write amplification problem, and it becomes more inefficiency when dealing with unaligned writes. So, in the OSD level, it is challenging to organize the metadata and efficiently store it in the KV store. NVStore aims at handling unaligned write problem and improving the efficiency of metadata management with two techniques (i.e., *Flattened Block Map* and *Decoupled Object Metadata*).

NVStore provides the same object interfaces as FileStore and BlueStore. For Ceph's OSD system, there are two types of metadata which must be supported, `onode` and block map. The `onode` is similar to the file's `inode` and contains the basic types of metadata (e.g., `size`, `flags`, `order`, ...). The block map (i.e., `Extent Map`) contains the extent information. Both FileStore, BlueStore and NVStore use key-value store to store above-mentioned two types of metadata. In addition, because BlueStore and NVStore organize the object in user-space thus extra block map metadata (i.e., `Blob Map`) is required.

Unlike BlueStore which uses traditional `Extent map` and KV store to organize and store the block map, NVStore uses a *Flattened Block Map* which is affinity to KV store and aims at promoting the access performance while reducing the write amplification from metadata. Besides, to support the *Flattened Block Map*, NVStore proposes the *Decoupled Object Metadata* mechanism to decouple the relationship between `onode` and block map, and this further reduces the write amplification from metadata.

3.3.1 Flattened Block Map

NVStore designs the *Flattened Block Map* in terms of the following observations:

1) The object size is always small (i.e., MB level) in current distributed object storage systems. Data with big size will be divided into multiple small objects, and the object size is configurable (i.e., the default object size in Ceph is set as 4MB). Therefore, the traditional IM and ET structures, which support large file (i.e., GB to TB level), are unnecessary in current object storage architectures.
2) The unaligned writes will lead to small block map updates. One unaligned write requires extra one or two blocks (less than 4KB) being updated. Moreover, the current block map mechanisms (i.e., IM and ET) need to update 1 to 3 metadata blocks for a block update operation. The state-of-art systems (e.g., TableFS, BlueStore) propose to use KV store to manage metadata, and thus, reduce the write amplification from metadata updates. However, the modification in value field which only updates a small part of the value requires to update the whole KV pair, this also incurs write amplification.

NVStore proposes the *Flattened Block Map* mechanism to reduce the write amplification from current KV based block map mechanism. Fig. 8a shows the structure of the *Flattened Block Map*. NVStore stores objects in unit of NVMe SSD's block size (4KB or 8KB) and indexes each block using the unique object id (`uuid, 8bytes`) and the block offset (`offset, 8bytes`) as the key. In the value field, NVStore records three items, the checksum (`csum, 4bytes`), the physical address (`paddr, 8bytes`) and the fragment data confirmation flag (`on, 1byte`).

Besides, NVStore also designs a fragment data structure to store the unaligned data in the KV store. Fig. 8a shows the KV pair format of the fragment data. The key field contains three items, the `uuid`, `offset` and the order of the fragment write (`order, 1 byte`). The value field contains two items, the offset of the data inside a block (`inoff, 2bytes`) and the fragment data (less than a block size).

Fig. 8b shows an instance of the flattened block map. For aligned write, it directly writes data to the device. Similar to the existing file systems, NVStore uses the bitmap to allocate the device space. The `paddr` directly points to the physical zone. The `on` flag is set to zero to indicate it as an aligned block. For an unaligned write in a block, NVStore will record its block map in the KV store and sets the `on` flag with one when first writing this block. NVStore will record the fragment data in the KV store as well as its writing order in this block. When read operation comes, if the `on` flag is zero, we could retrieve the data from the raw SSD device, if the `on` flag is one, we could retrieve the FP in KV store and recover the whole data page.

In this way, an unaligned write only requires 1 or 2 metadata updates when the unaligned write first happens within a block or cross two blocks. Moreover, we only need to update the `on` flag when updating the metadata, and this only requires a value updating with 18 bytes in the KV store. However, in BlueStore, it requires a value updating with 56-4096 bytes in the KV store. The follow-on unaligned writes to the same block will not affect the flattened block map and only insert data to the fragment data zone until an aligned write comes or NVStore execute the data sync operation. In the data sync operation, NVStore will modify the `on` flag to zero and then discard the fragment data through the prefix operation in RocksDB.

In conclusion, the *Flattened Block Map* mechanism co-designs the metadata structure and the KV store. Compared with the solution of storing a tree structure (i.e., ET) or a multi-level index table (i.e., IM) in KV store, NVStore provides a flattened structure which is more efficient in metadata operation for unaligned writes and reduces the write amplification from metadata.

3.3.2 Decoupled Object Metadata

Decoupling the metadata is a widely used optimization method in the metadata management (e.g., IndexFS, LocoFS [33]). We also proposes a *Decoupled Object Metadata* mechanism in NVStore.

Similar to the BlueStore which decouples metadata into different parts (i.e., `onode, Shard Map, Extent Map, Blob Map`), NVStore further decouples the relationship between `onode` and the block map structures. Fig. 9 shows the extra three types of metadata in NVStore in addition to the above-mentioned flattened block map. In BlueStore, `onode` and `Shard Map` are stored in the same KV pair and `onode` will record the address of the `Shard Map` as well as some basic object metadata (i.e., data size, block size and so on). So, the metadata structures in BlueStore are highly dependent on each other, an object update operation may need to modify all the metadata structures. In terms of this,

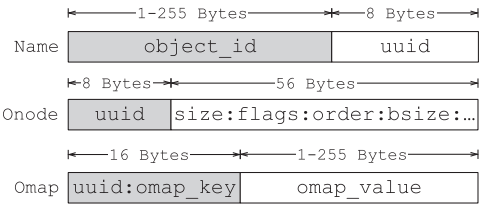


Fig. 9. Metadata of NVStore.

NVStore stores onode and the flattened block map structures separately, and the onode only stores the basic object metadata. In this manner, NVStore does not have to make any changes to the onode when updating the data of an object. Only those object operations which affect the basic metadata in onode, like append and truncate operations, will update the onode. What's more, the KV pair size of onode in NVStore is much shorter than BlueStore, this accelerates the metadata operations while reducing the write amplification from onode updates. To be compatible with Ceph, we keep the omap structure.

Since the whole flattened block map can be retrieved via the prefix search operation and then the object meta-information in onode can be recovered, NVStore would not sync the onode metadata when executing append or truncate operations. This mechanism further reduces the write amplification from the metadata sync operations.

3.3.3 Metadata Cache

Besides optimizing the metadata layout, we introduce a metadata cache to accelerate the metadata accesses. For a touch operation, the name map and onode will be updated. For a write operation, the onode and Flattened Block Map will be updated. Besides, the remove, clone, read operations will also update or access different metadata items. We observe that name map is frequently read by all types of operations, but only updated by touch and remove operations, which happens occasionally during the lifetime of an object. Moreover, onode which stores the basic metadata of objects will be accessed and updated frequently by most operations.

In terms of these observations, NVStore introduces a metadata cache as Fig. 10 shows. NVStore only caches the name map and onode in memory and use a background thread to synchronize the name map when creating or removing objects and to synchronize the onode when receiving a flush operation or a soft time-interruption. Although the index tables in Flattened Block Map are also frequently accessed by write and read operations, NVStore will not cache them in the metadata cache. In NVStore, the index tables are stored in the B^+ tree based KV storage. One object contains multiple index table items with a similar prefix in keys, and they tend to be stored adjacently in the B^+ tree data structure, thus providing advantageous locality. Such designs can accelerate the performance of read or write operations because B^+ tree based KV store will cache the adjacent items into memory beforehand. Especially, NVStore ensures the atomicity of multiple updates in one metadata operation with the transactional interface provided by the KV store.

Unlike BlueStore which caches all the metadata structures indiscriminately, NVStore caches different metadata based on their access patterns. In this way, NVStore avoids the overhead of duplicate data caches in KV store cache and the metadata cache. What's more, the metadata cache chooses to cache the metadata which is not frequently modified but accessed frequently, and thus, reduces the frequency of metadata synchronization and improve the efficiency of querying the metadata items.

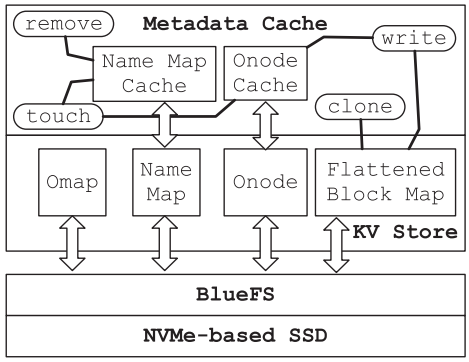


Fig. 10. Metadata cache.

3.4 Compatibility

Although NVStore uses a different metadata structure and co-designs the metadata management and KV store, it could also be compatible with the advanced function of Ceph (e.g., checksum and clone).

3.4.1 Checksum

BlueStore chooses to calculate the checksum of the whole object data block and store the whole checksum data into the extent tree. In contrast, NVStore proposes to calculate the checksum for each data page and store the checksum in csum as Fig. 8 shows. For unaligned data, NVStore will not calculate its checksum until it is synced to the block device. In this manner, when user makes a small write request to an object, NVStore only needs to recalculate the checksum of a small data block and update a small KV pair. However, BlueStore needs to re-insert the whole extent block thus degrading the KV performance.

3.4.2 Clone

NVStore also supports the block-level share and clone operations. The on flag in Fig. 8 can be used to defined a shared block when the value of on is greater than one. For other object which share with the block, the csum and paddr filed will be filled with the uuid and offset of the shared object. In NVStore, a fragment block could not be shared until data sync operations. Compared with BlueStore, the clone mechanism is simpler and needs no additional structures.

3.5 Limitation

3.5.1 Large Write

NVStore is designed to improve the performance of unaligned write and could also promote the small write efficiency. However, these mechanisms incur high metadata overhead when executing large write. For a large write (i.e., 64KB-1MB), NVStore will insert 16 to 256 key-value pairs to

TABLE 3
The Hardware Configuration

Server Name	SuperMicro
# of Machines	5
CPU	Intel Xeon 24 cores 2.5GHz × 2
Memroy	DDR4 384G
Storage	Intel 750 × 2
Network	Mellanox SX1012 Switch CX353A ConnectX-3 FDR HCA

the KV store for the NVMe SSD with 4KB block size thus involving 200 bytes to 6400 bytes metadata updates. However, BlueStore only updates 56 to 4096 bytes metadata. Fortunately, because the inserted items are sequential and the data size is quite large, the `batch` operation in RocksDB can write these data to a continuous block thus compensating for the performance loss of large write from metadata updates in NVStore.

3.5.2 Compress

The designs of NVStore are inefficient to the compress function because the size of a compressed block is unpredictable. The I/O path and metadata structure of NVStore is suitable for the character of NVMe SSDs and tries to write aligned pages to the device. The compress process may transform the uncompressed aligned data to unaligned data, and the small data updates will affect the final compressed output in a great extent. In NVStore, it will lead to excessive data fragment and bloat the KV store thus degrade the overall write performance.

3.5.3 Garbage Collection

In NVStore, the KV store records the data of unaligned writes (i.e. Fragment data in Fig. 8b). If the unaligned writes to the same page are not synced, the records in the KV store are responsible for the data consistency of this page. As mentioned in Section 3.2.4, to avoid excessive data fragmentation in the KV store, NVStore can sync the cache pages in a lazy style. Meanwhile, NVStore performs garbage collection, in which the corresponding unaligned write items of those synced pages in the KV store will be deleted. For example, When a cache page (DP or FP), which is indexed by `uuid:offset`, is synced, NVStore will delete all Fragment Data KV pairs with the prefix `uuid:offset` in the key field.

4 Evaluation

In this section, we compare NVStore with the traditional file system based OSDs (i.e., Ceph's FileStore), the state-of-art OSDs (i.e., Ceph's BlueStore) and other distributed file systems (i.e., Lustre, Gluster). Also, we compare NVStore with the NVStore-cache which only contains the *Flexible Cache Management* and the NVStore-meta which only contains the *KV Affinity Metadata Management*. First, We evaluate the write amplification optimization of NVStore (Section 4.2) in both metadata and data. Second, we evaluate the single OSD performance of NVStore with variable write I/O sizes (Section 4.3). We then evaluate the clustering performance of NVStore (Section 4.4). Finally, we evaluate the influence of some significant factors, such as I/O depth and aligned writes, on NVStore (Section 4.5).

4.1 Experimental Setup

4.1.1 Hardware Configuration

Our experiments are deployed respectively on a local environment and a cluster environment (shown in Table 3). The cluster consists of 5 SuperMicro servers with CentOS 7.3 and CentOS 7.3 (Lustre version) installed, each of which has 384GB DDR4 memory and two Intel Xeon 24-cores CPUs. The server in both local and cluster environments have 2×Intel 750 SSDs. Moreover, all the servers in the cluster are interconnected with Mellanox SX1012 Switch (56 Gb/s InfiniBand) to better exploit the SSD's performance.

4.1.2 Software Configuration

Table 4 lists the configurations of each experiment, including the number of machines, the benchmark tools, the compared storage systems, the object size, the write I/O size, the total write size and running time. Our evaluations use Ceph luminous [34], the latest version from Github, Lustre 2.11 and GlusterFS 3.14 as the storage system. To demonstrate our design clearly, we provide three self-modified micro-benchmark tools, Object Bench (OB, modified from `ObjectBench`), Rados Bench(RB, modified from FileBench) and Cluster Bench (CB, modified from FileBench). OB supports evaluating Ceph's Object Store (i.e., BlueStore, FileStore and NVStore) on write amplification and performance with different write patterns (i.e., append write, random write and overwrite), variant object size and write I/O size. Since FileBench is a useful benchmark to generate the

TABLE 4
Software Configuration

Experiments	Fig. 11	Fig. 12	Fig. 14	Fig. 15	Fig. 16	Fig. 17	Fig. 18
# of machines	1	1	1	1	5	1	5
Benchmarks	OB[1]	OB[1]	OB[1]	RB[2]	CB[3]	OB[1]	IOzone
Object Size	-	-	4 MB	4 MB	4 MB	4 MB	4 MB
Align	✓	×	×	×	×	✓	✓
Write I/O Size	-	2 KB	2 KB	-	-	-	-
Total Write Size	16 GB	16 GB	16 GB	-	-	16 GB	-
Running Time	-	-	-	600s	600s	-	-
Storage Systems	Ceph Luminous, GlusterFS 3.14, Lustre 2.11						

[1] *An object storage benchmark modified from `ObjectBench`. This tools can generate different fixed size of workload from indicated offset.*
[2] *A modified filebench which could run on Ceph's Rados Layer. This tools can generate the same workload as filebench.*
[3] *A modified filebench which could support multiple clients. This tools can generate the same workload as filebench.*

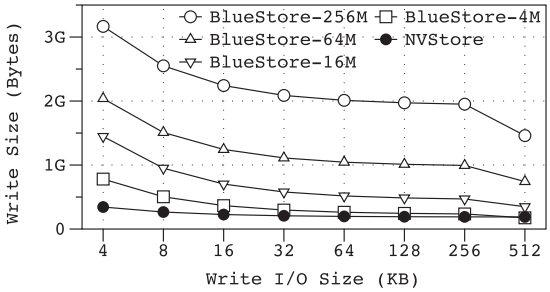


Fig. 11. Metadata write traffic under aligned writes. The write traffic of each workload is 16 GB. BlueStore-256M means the object size is 256MB in the test with BlueStore. Others are in a similar way. The object size in NVStore is 4MB.

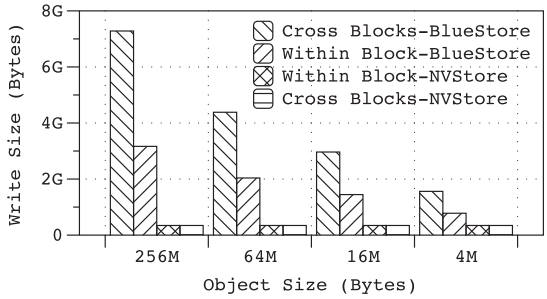


Fig. 12. Metadata write traffic under unaligned writes. The write traffic of each workload is 16 GB. Within Block-NVStore means the test of within block unaligned writes in NVStore. Others are in a similar way.

simulated real workload with unaligned writes, we develop RB and CB to allow running FileBench's workload on Rodos and support multiple clients. Therefore, we could evaluate the unaligned write performance in both single node and cluster environments. Besides, to demonstrate the impaction of aligned writes on NVStore, we employ iozone (a benchmark supports multiple clients and generate aligned writes) to evaluate the performance of common distributed file systems (i.e., CephFS, Lustre and GlusterFS).

To exploit the full performance of Ceph with FileStore, BlueStore and NVStore. We employ two SSDs to avoid the impact of mixed I/Os. For FileStore, one NVMe SSD is used as the journal disk formatted with Ext4 or XFS and the other is used as the data disk. For BlueStore and NVStore, one NVMe SSD is used to store both KV data and WAL data and the other is used as the data disk.

4.2 Write Amplification

In this section, we evaluate the write amplification from metadata and data respectively. We use OB to generate workload and use Ceph's `PerfCounter` [35] to collect the results.

4.2.1 Metadata Write Traffic in Aligned Writes

We first evaluate the benefits of the flattened block map in aligned writes. Since the overhead of extent tree is associated with the size of the object, and BlueStore uses the KV based extent tree to map the block, we evaluate NVStore and BlueStore under variant object size from 4MB (default size) to 256MB. In each experiment, we use OB to generate 16GB aligned append writes with the average write I/O size ranging from 4KB to 512KB. We collect the size of metadata write traffic from BlueStore (`onode` and `block map`) and NVStore (`onode` and `Flattened Block Map`) using `PerfCounter`.

Fig. 11 shows the total write size of metadata in each object store. Because the total write size of metadata in NVStore is independent of object size, we only show the result of 4MB object size in this figure. We could make the following observations:

- NVStore achieves the lowest metadata write traffic and the most stable metadata overhead in aligned writes under different write sizes. When the write size is 4KB, the metadata write traffic of BlueStore is about 1.5 to 8× compared with NVStore as the object size increases. When the write size is large (i.e., 256KB and 512KB), NVStore also achieves smaller metadata write traffic compared to BlueStore. This is because the total metadata update size in NVStore has no relationship with the object size, and it only concerns with the total data write size. In NVStore, the metadata write traffic of NVStore is about 1.9 percent of the total data write traffic. In this perspective, NVStore is suitable for small writes.
- The object size and write I/O size have egregious impact on the KV based extent tree structure. Since each append write will add items to the `extent map` and modified the index item in `onode`. As the object size increases and the write I/O size decreases, the size of mapping structures will increase, the leaf nodes of extent tree will also split and reorganize more frequently. This causes significant write amplification.

4.2.2 Metadata Write Traffic in Unaligned Writes

In this section, we evaluate the benefit of flattened block map in unaligned writes. We compare the metadata write traffic under unaligned write in NVStore with BlueStore in cross blocks and within block situations. We use OB to generate these workloads. The total data write traffic in each evaluation is also 16GB. For within block, we generate append write with 2KB write size, and each append write is with `offset` $= 0$. For cross blocks, we generate append write with 4KB and `offset` $= 2048$. We also collect the write traffic of metadata from BlueStore (`onode` and `block map`) and NVStore (`onode` and `Flattened Block Map`) using `PerfCounter`.

Fig. 12 shows the total write size of metadata in BlueStore and NVStore under different object size. Fig. 13 shows the extra write traffic to the KV store. The extra data write size is computed by the total metadata table file sizes in the KV store subtracted from the total metadata write size in Fig. 12. We could make the following observations:

- NVStore achieves the lowest write traffic of metadata and the most stable metadata overhead in unaligned writes under different write sizes. For within block, the metadata write traffic in BlueStore is about 2.5 to 10× of NVStore as the object size increases. For cross blocks, the metadata write traffic in BlueStore is about 4 to 19× of NVStore. This proves that NVStore can significantly reduce the write traffic of metadata in unaligned writes, especially when objects are large.

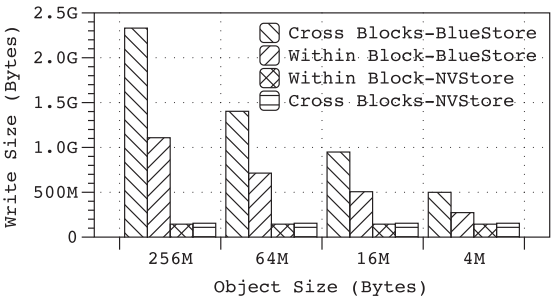


Fig. 13. Extra Write traffic in RocksDB under unaligned writes. The write traffic of each workload is 16 GB. Within Block-NVStore means the test of within block unaligned writes in NVStore. Others are in a similar way.

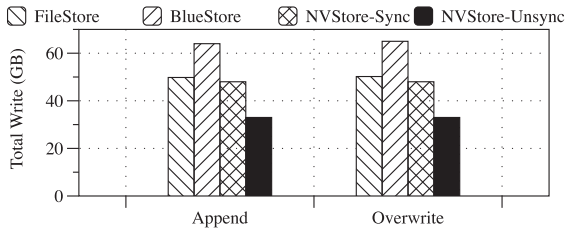


Fig. 14. Data write traffic under unaligned writes. For BlueStore and NVStore-Sync, the results are collected from the KV stores and the data disk. For NVStore-Unsync, it is only from the KV Store. For FileStore, it is collected from the data disk and the journal disk. The write traffic of each workload is 16 GB. The object size is 4 MB.

- The write amplification from the KV based extent tree structure becomes more severe under unaligned writes. Unaligned writes will lead to more leaf nodes compared with aligned writes thus incurring more node split and merging operations. Therefore, it leads to severe write amplification and significantly degrades the write performance.
- NVStore also achieves the lowest write traffic to the KV store. First, NVStore writes less metadata than BlueStore with the design of the Flattened Block Map. Besides, the value size of the metadata KV pair in NVStore is smaller than BlueStore, so the write amplification from the compaction of RocksDB in NVStore is quite smaller than BlueStore. Both NVStore and BlueStore use write-ahead-log (WAL) in the KV store. Since NVStore writes less metadata to the KV store, the WAL write traffic in NVStore is smaller than BlueStore. To simplify the write traffic collecting process, the WAL write traffic is not included in Fig. 13.

4.2.3 Data Write Traffic in Unaligned Writes

In this section, we evaluate the write traffic of data under unaligned writes with different write patterns (append write and overwrite) in NVStore, FileStore and BlueStore. We use OB to generate these workloads. The total data write traffic in each evaluation is 16GB. The unaligned writes are with 2KB write size and `offset` $= 0$. We evaluate NVStore-Sync mode and NVStore-Unsync mode of NVStore respectively. In the NVStore-Sync mode, the unaligned writes will be synced to. In the NVStore-Unsync mode, the lazy page sync mechanism is applied. We collect the write traffic of data (including the metadata) from BlueStore and NVStore using `PerfCounter`, from FileStore using `blktrace` [36].

Fig. 14 shows the total write size in FileStore, BlueStore and NVStore. In this section, we calculate BlueStore and NVStore's WAL write traffic as the workload traffic (16GB). We could make the following observations:

- NVStore-Unsync achieves the smallest write traffic, and it is about 50 percent of BlueStore and 65.7 percent of FileStore. In NVStore, we write the unaligned data to the KV store and use the KV store to guarantee the data consistency rather than performing RMW operations. In the KV store, the unaligned data is first appended to the write-ahead-log and then persisted to the back-end devices as aligned data. The KV store overhead is quite smaller than the RMW operations. Because the WAL syncing is off the critical path of writes, and the KV store writes data to disk in big batches.
- BlueStore achieves the largest write traffic in this evaluation. For BlueStore, small data (less than 64 KB) is also written to the KV store first, and then the data will be written back to data disk using RMW operations. In this way, a 2 KB write is processed as a 2 KB KV pair write, a 2 KB WAL record write, and a 4 KB final write in RMW. So the write traffic is almost $4\times$ of the data size. In the NVStore-Sync mode, NVStore merges the unaligned data in the KV store and write the merged data back to the data disk. This reduces the overhead of RMW operations in BlueStore.
- FileStore achieves smaller data write traffic than BlueStore. FileStore uses a logging mechanism, a 2 KB data write is processed as a 2 KB batched journal append-write and a 4 KB final write in RMW. So the write traffic is almost $3\times$ of the data size. Since blktrace can only record all I/O operations, the metadata write traffic is included in the results.

In summary, NVStore can effectively reduce the write amplification from unaligned writes, and writes less than the existing system when no real-time unaligned data synchronization is performed. When the synchronization is introduced, its write traffic is about the same as the BlueStore. However, NVStore does not require real-time data synchronization, this gives NVStore the advantages over BlueStore in write performance. Besides, NVStore achieves the lightest write traffic in aligned small writes.

4.3 Local Performance

In this section, we evaluate the single-node performance of NVStore, FileStore (based on Ext4) and BlueStore under unaligned writes. To demonstrate the performance improvement brought by different optimizations, we evaluate three versions (i.e., NVStore-Cache, NVStore-Meta, NVStore-All) of NVStore. NVStore-Cache only adopts the *Flexible Cache Management* mechanism to optimize the cache management. NVStore-Meta only adopts the *KV Affinity Metadata Management* mechanism to optimize the metadata management. NVStore-All is the fully-functioned version with all the optimizations. The modified Filebench (RB) which could run on Ceph's Rados layer is used for our evaluations. We deploy Ceph on a single node, and runs RB on it to collect the performance evaluation under different block sizes.

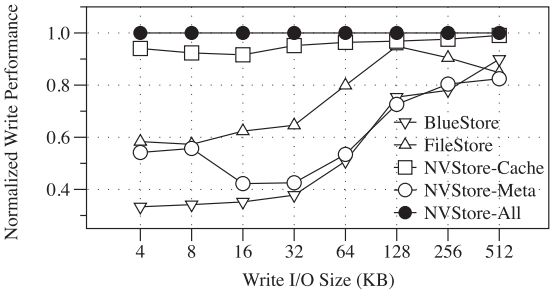


Fig. 15. Local performance. Each test runs 600s. The object size is 4 MB.

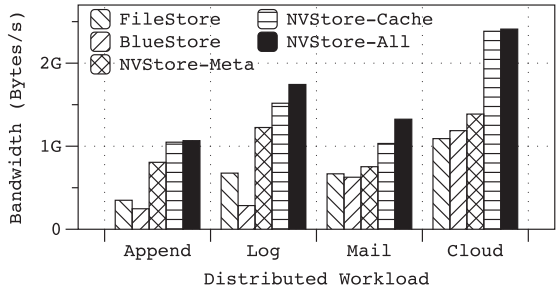


Fig. 16. Real workload performance in cloud.

Fig. 15 shows the single node write throughput (MB/s) of BlueStore, FileStore and NVStore as the write I/O size increases from 4KB to 512KB. NVStore-All achieves the highest write performance in all evaluations. The write throughput of NVStore-All is about 1.11 to 3.00× of BlueStore and 1.05 to 1.75× of FileStore under different write I/O sizes. NVStore-Cache outperforms all the other systems except NVStore-All. In most cases, the write performance of all the systems increases as the block size increases, and the write performance of NVStore-Meta is between FileStore and BlueStore. When the write size reaches 512KB, the write performance of FileStore decreases and is lower than both NVStore-Meta and BlueStore. From this evaluation, we could conclude that NVStore could effectively improve the performance of unaligned writes, and the cache optimizations play a key role in it. This is because NVStore reduces the extra I/O overhead from unaligned writes and reduces the write traffic without data sync operations. Moreover, the metadata optimizations further promote the write performance of NVStore. In this evaluation, we use 4 MB objects, the write amplification in BlueStore and NVStore is similar. Compared with BlueStore, NVStore-Meta still achieves at most 20 percent performance improvement in 4KB write size. Since FileStore adopts the asynchronous write method and returns when data is written to the log, so the write performance is better than BlueStore in most cases.

4.4 Cluster Performance

In this section, we evaluate the cluster performance of NVStore, FileStore and BlueStore under unaligned writes. Since the CephFS and Rados Block Devices (RBD) in Ceph adopt a page based data management, the unaligned writes could not be perceived by the underlying OSDs. To this end, we use the modified Filebench (CB) which could run on Ceph's Rados interfaces to evaluate the unaligned writes performance under a 5-node cluster. We use CB to generate unaligned write patterns based on real workloads (i.e., appendfile, logfile, mail server, cloud server), and then evaluate the cluster bandwidth in FileStore, BlueStore and NVStore (NVStore-Cache, NVStore-Meta and NVStore-All).

As in Fig. 16, we could observe that NVStore-All achieves the highest bandwidth under all the scenarios, it is about 2.03 to 6.11× of BlueStore and 1.99 to 3.06× of FileStore. This is because of the small and random writes in these workloads. Appendfile, logfile, and mail server workloads tend to generate more small writes, this has a great impact on the performance of BlueStore. FileStore has better performance than BlueStore due to its logging mechanism which shields the effects of unaligned writes. NVStore can greatly improve the unaligned writes performance because it uses an optimized cache mechanism.

In summary, NVStore can greatly improve the unaligned writes performance in real distributed applications. It should be noted that since the client of the distributed file system currently adopts a page-based client cache management mechanism, NVStore cannot be directly used for the existing distributed file system.

4.5 Overhead Evaluation

4.5.1 Sensitive to I/O Depth

I/O depth is the number of the on-the-fly I/O requests. It greatly affects the performance of storage systems, especially for direct I/O. To understand NVStore's sensitivity to the I/O depth, we evaluate Ceph with NVStore, BlueStore and FileStore using two I/O depth settings, 100 and 1000.

Fig. 17 shows Ceph's bandwidth when using NVStore, BlueStore and FileStore with different I/O depth settings. We could make the following observations:

- I/O depth has a higher impact on direct I/O (like in NVStore and BlueStore) than buffered I/O (like in FileStore), because buffering mitigates the impact from I/O depths. With different I/O depths, CephFileStore maintains similar performance, with a maximum difference of 21 percent. In contrast, NVStore has a maximum difference of 60 percent, and BlueStore has a maximum difference of 200 percent.
- NVStore is less sensitive to the I/O depth than BlueStore. When the write size is less than a block size, BlueStore has greater bandwidth difference between different I/O depths. One possible reason is that when the number of on-the-fly requests is limited, the SSD bandwidth can not be saturated. Since NVStore use the KV store to store the unaligned

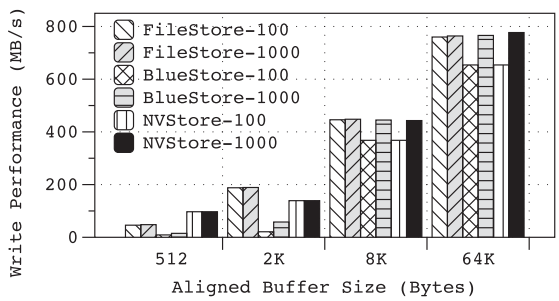


Fig. 17. Sensitivity to I/O Depth. The write traffic of each workload is 16GB. The object size is 4MB. NVStore-100 means testing NVStore with the I/O depth 100. Others are in a similar way.

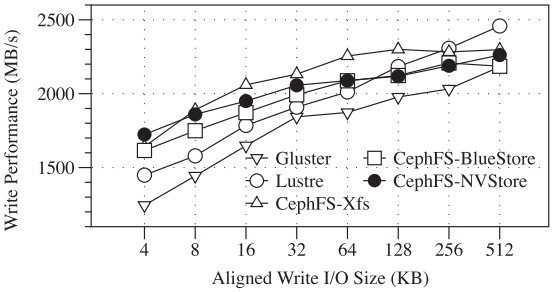


Fig. 18. Performance evaluation with varied write I/O sizes under aligned writes.

writes, the unaligned write performance is not affected by the I/O depth, but by the performance of the KV store.

As such, we conclude that I/O depth has an impact on storage systems which use direct I/Os. With better data layout, this impact can be reduced as in NVStore.

4.5.2 Aligned Write Performance

In this section, we use iozone to evaluate the overall system performance of NVStore under aligned writes. The compared systems are Lustre and CephFS, which are configured with 1×metadata servers (`mdt` in Lustre and `mds` in Ceph), 5×OSD servers (`ost` in Lustre and `osd` in Ceph) and 5×clients.

Fig. 18 shows the write bandwidth of the two evaluated file systems with different write I/O sizes and different backend storage systems. With 4KB write I/O size, NVStore shows the highest write bandwidth, and is 1.6× of CephFS-XFS, 1.5× of Lustre and 2.64× of CephFS-BlueStore. This is because NVStore directly write aligned data to the disk and update the metadata with finer granularity. Moreover, NVStore performs better than any other system except Ceph-XFS when the write I/O size is below 64 KB. Since Ceph-XFS is based on FileStore and returns when data is written to the log, and these log writes is aligned sequential writes which have a large advantage when the write granularity is small. Due to the poor performance of the *Flattened Block Map* for large writes, the performance of large writes in NVStore is not as good as the existing distributed file system, but the performance gap is not obvious, which is 5 percent worse than BlueStore and 10 percent worse than Lustre.

5 Related Work

Handling Unaligned Accesses. The unaligned I/O patterns of computational science has long been considered as one of the challenges at leadership scale [37]. Campello *et al.* [38] reveal the causes of unaligned access: the mismatch in data access granularities (bytes accessed by the application, and pages accessed from storage by the operating system). Client-based Caching can reduce the throughput loss caused by frequent small and unaligned I/Os [39], [40], [41]. Settlemyer B. [39] conducts a study of client-based caching for parallel I/O and proposes progressive page caching that represents cache data using dynamic data structures rather than fixed-size pages of file data. With emerging high-speed storage devices (e.g., SSD, NVRAM, PCM), the burst buffer is considered as a promising solution for the I/O intensive workloads on the HPC systems [42], [43], [44], [45], [46]. BurstFS [47] is an SSD-based distributed file system to be used as burst buffer for scientific applications. NVFS [42] adopts a NVM-based burst buffer for running Spark jobs on top of parallel file systems. To optimize the process blocking during page fetch when writing to non-cached file data, Campello *et al.* [38] decouple the writing of data to a page from its presence in memory by buffering page updates elsewhere in OS memory. iBridge [8] proposes to utilize SSDs to compromise the weakness of hard-disk-based servers in serving small fragment requests. TokuFS [48] uses Fractal Tree indexes for microdata write workloads which features creating and destroying many small files, performing small unaligned writes within large files and updating metadata. Unlike these works, NVStore focuses on the unaligned write problems both from the OS and NVMe SSD device perspectives.

Flash based File Systems. Flash based SSDs are adopted widely in the last decade. The unique characteristics in SSDs compared to hard disk drives calls for disruptive changes in file systems to exploit its potentials. Direct File System (DFS) [49] simplifies the data allocation in file systems by leveraging the data allocation functions in flash translation layer (FTL). The removed redundancy leads to better performance. Object-based Flash Storage System (OFSS) [50] proposes to manage flash memory directly via software (this architecture is later called open-channel SSD), and re-designs an object-based file system in a software (SW)-hardware(HW) co-designed way. Due to the tight SW/HW co-design, write amplification in the file system is significantly reduced, thereby improving flash endurance. Cheng Ji *et al.* [51] propose an empirical study of filesystem fragmentation problems and provide two pilot solutions to enhance file defragmentation. ReconFS [52] redesigns the directory tree in a reconstructable way to reduce the metadata write overhead by leveraging the asymmetric read/write features of flash. ParaFS [53] further exploits the internal parallelism of flash based SSDs by co-designing functions that corresponding to both FTL and file system layers. Comparatively, F2FS [54] is more conservative and has gone into the Linux kernel. F2FS also optimizes the layout for the flash features. While a myriad of efforts have been made to local file systems, which could improve storage nodes' performance of distributed file systems, optimizations to distributed file systems have not well studied for high-performance SSDs. NVStore is towards this direction.

Kernel Bypassing. Since recent network and storage hardware provides extremely high performance, software overhead is no longer a negligible part [55], [56]. For high speed networking, user-level networking stack is intensively researched to reduce data copies along the TCP/IP stack [57]. Similarly, RDMA (Remote Direct Memory Access) bypasses the operating systems and supports zero-copy networking [58]. For high performance non-volatile memory, storage system software takes similar ways. Moneta-D [55] designs a user-space storage system by transparently bypassing the operating system. Recent persistent memory file systems, including BPFS [59], SCMFS [60], PMFS [61], HiNFS [62], and Nova [63], read or write files in a direct access (DAX) way. The DAX is also supported in the Linux community to support persistent memory [64]. Even flash memory is slower than non-volatile

main memories, the high-end SSDs support hundreds of thousands of IOPS (i.e., input/output operations per second). To exploit the hardware benefits, Intel proposes the SPDK (storage performance develop kit) which is designed in user-space and uses polling to reduce the latency of accessing NVMe devices. SSDFA [65] is a user-space file system that manages a number of low-cost commodity SSDs to achieve a million IOPS for data accesses. BlueStore [66] direct performance I/O operations to SSDs by bypassing the Linux kernel to explore the SSD performance. Differently, our proposed NVStore is designed for high-end SSDs. NVStore uses direct and buffered I/O in a combinative way to take both advantages.

Key-Value Based Metadata Management. Key-value store shows high performance for small data writes, and thus is regarded as a promising way to store metadata. TableFS [67] and Ceph's BlueStore respectively use LevelDB and RocksDB to store both metadata and the small files. IndexFS [68] and BatchFS [29] use LevelDB [69] to store the metadata of distributed file system, and achieves linear metadata scalability of batch file accesses. In addition to metadata management using key-value stores, some research works also try to manage data in a key-value access way. KVFS [70] is one of the file systems which manages files in a key-value way using VT-tree. GlobleFS [71] and Ceph's Kstore use LevelDB to store both data and metadata. Our proposed NVStore manages metadata and store the unaligned writes in the key-value store. For the key-value inefficiency problem, WiscKey [72] has pointed out that co-locating values with keys leads to inefficient organization of keys, which results in slow reads. HashKV [73] uses hash-based data grouping, which deterministically maps values to storage space so as to make both updates and GC efficient. Chen *et al.* [74] identify that the existing fixed-sized management strategies of flash-based devices would potentially result in low storage space utilization and propose a KV flash translation layer design to improve storage space utilization as well as the performance of the KV SSDs. These optimizations could also be adopted by NVStore.

6 Conclusion

To optimize the unaligned writes in cloud storage with high-performance SSDs, we propose an OSD systems called NVStore. For the overhead incurred by unaligned writes in data path, we designs a *Flexible Cache Management* mechanism. By introducing the fragment page and redesigning the cache management, we reduce the RMW operations, and accelerate the unaligned write performance by reducing page sync operations while ensuring the data consistency. For the overhead incurred by unaligned write in the block map table, we propose a *KV Affinity Metadata Management* mechanism. We co-designs the block map and key-value store to provide a flattened block map and a decoupled object metadata management. In this manner, NVStore promotes the access performance while reducing the write amplification from metadata. Evaluations demonstrates the effectiveness of NVStore in improving the performance of unaligned writes and reducing the write amplification both from metadata and data under unaligned writes. Besides, NVStore is compatible with the advanced function of Ceph.

Acknowledgments

This work was supported in part by National Key Research & Development Program of China (Grant No. 2018YFB1003301), in part by the National Natural Science Foundation of China (Grant No. 61772300 and 61832011), in part by Research and Development Plan in Key Field of Guangdong Province (Grant No. 2018B010109002) , and in part by SenseTime Research Fund for Young Scholars.

References

[1] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002.
[2] P. J. Braam and others, "The Lustre storage architecture," 2004.
[3] B. Calder et al., "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syste. Princ.*, 2011, pp. 143–157. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043571
[4] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proc. ACM/IEEE Conf. Supercomputing*, 2004, Art. no. 4. [Online]. Available: https://pdfs.semanticscholar.org/bd2d/e7db1009211e56e1aa1ff91c53782c1e468a.pdf
[5] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 257–270.
[6] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: Understanding the I/O behavior of apple desktop applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 3, 2012, Art. no. 10.
[7] "Lasr system call IO trace," [Online]. Available: http://iotta.snia.org/tracetypes/1
[8] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 381–392.
[9] Q. Xu *et al.*, "Performance analysis of NVMe SSDs and their implication on real world databases," in *Proc. 8th ACM Int. Syst. Storage Conf.*, 2015, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2757667.2757684
[10] AXBOE, "fio-flexible I/O tester," 2014. [Online]. Available: http://freecode.com/projects/fio
[11] W. D. Norcott and D. Capps, "Iozone filesystem benchmark," 2003.
[12] J. Bhimani *et al.*, "Understanding performance of I/O intensive containerized applications for NVMe SSDs," in *Proc. IEEE 35th Int. Perform. Comput. Commun. Conf.*, 2016, pp. 1–8.
[13] Z. Yang *et al.*, "AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter," in *Proc. 36th IEEE Int. Perform. Comput. Commun. Conf.*, 2017, pp. 1–8.
[14] R. McDougall and J. Mauro, "FileBench," 2005. [Online]. Available: http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf
[15] A. Davies and A. Orsaria, "Scale out with glusterfs," *Linux J.*, vol. 2013, no. 235, Nov. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555789.2555790
[16] D. Kim, H. Kim, and J. Huh, "vCache: Providing a transparent view of the LLC in virtualized environments," *IEEE Comput. Architecture Lett.*, vol. 13, no. 2, pp. 109–112, Jul.–Dec. 2014.
[17] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "CORFU: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 1–24, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2542150.2535930http://dl.acm.org/citation.cfm?id=2535930%5Cnhttp://dl.acm.org/citation.cfm?doid=2542150.2535930
[18] M. Balakrishnan *et al.*, "Tango: Distributed data structures over a shared log," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 325–340. [Online]. Available: http://www.cs.cornell.edu/taozou/sosp13/tangososp.pdf
[19] "FILESTORE config reference," 2016. [Online]. Available: https://docs.ceph.com/docs/master/rados/configuration/filestore-config-ref/
[20] A. Samuels, "Ceph high performance without high costs," [Online]. Available: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160810_K21_Samuels.pdf

[21] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The next generation of Ext2/3 filesystem," in *Proc. Linux Storage Filesystem Workshop*, 2007.

[22] L. Changman, S. Dongho, H. JooYoung, and C. Sangyeun, "F2FS: A new file system designed for flash storage in mobile," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.

[23] A. Kumar, M. Cao, J. R. Santos, and A. Dilger, "Ext4 block and inode allocator improvements," in *Proc. Linux Symp.*, 2008, pp. 263–273.

[24] S. Tweedie, "Ext3, journaling filesystem," pp. 24–29, 2000.

[25] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1996, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268300

[26] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, 2013, Art. no. 9.

[27] K. Ren and G. Gibson, "TABLEFS: Embedding a NoSQL database inside the local file system," in *Proc. Digest APMRC*, 2012, pp. 1–6. [Online]. Available: http://www.mendeley.com/research/tablefs-embedding-nosql-database-inside-local-file-system

[28] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, *ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems*. New York, NY, USA: ACM, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2806777.2806844

[29] Q. Zheng, K. Ren, and G. Gibson, *BatchFS: Scaling the File System Control Plane With Client-Funded Metadata Servers*. New York, NY, USA: IEEE Press, 2014.

[30] "Facebook RocksDB," [Online]. Available: http://rocksdb.org/

[31] A. Hutton *et al.*, "Asynchronous I / O support in Linux 2.5," 2003.

[32] "SPDK: Storage performance development kit," [Online]. Available: http://www.spdk.io

[33] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "LocoFS: A loosely-coupled metadata service for distributed file systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 4:1–4:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126928

[34] "Ceph—A scalable distributed storage system," [Online]. Available: https://github.com/ceph/ceph

[35] "PERF COUNTERS," 2016. [Online]. Available: https://docs.ceph.com/docs/master/dev/perf_counters/

[36] A. D. B. Jens Axboe and N. Scott, "blktrace(8) - Linux man page," 2006. [Online]. Available: https://linux.die.net/man/8/blktrace

[37] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proc. Conf. High Perform. Comput. Network. Storage Anal.*, 2009, pp. 1–12.

[38] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche, "Non-blocking writes to files," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 151–165.

[39] B. Settlemyer, "A study of client-based caching for parallel I/O," 2009.

[40] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman, "Collective caching: Application-aware client-side file caching," in *Proc. 14th IEEE Int. Symp. High Perform. Distrib. Comput.*, 2005, pp. 81–90.

[41] X. Ma, J. Lee, and M. Winslett, "High-level buffering for hiding periodic output cost in scientific simulations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 3, pp. 193–204, Mar. 2006.

[42] N. S. Islam, M. Wasi-Ur-Rahman, X. Lu, and D. K. Panda, "High performance design for HDFS with byte-addressability of NVM and RDMA," in *Proc. Int. Conf. Supercomputing*, 2016, pp. 1–14.

[43] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: Burst buffer based I/O orchestration," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 194–203. [Online]. Available: http://www.mendeley.com/research/trio-burst-buffer-based-io-orchestration

[44] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 71–79.

[45] J. Bent *et al.*, "PLFS: A checkpoint filesystem for parallel applications," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–12.

[46] N. Liu *et al.*, "On the role of burst buffers in leadership-class storage systems," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[47] T. Wang, K. Mohror, A. Moody, W. Yu, and K. Sato, "BurstFS: A distributed burst buffer file system for scientific applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015.

[48] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The tokuFS streaming file system," in *Proc. 4th USENIX Conf. Hot Topics Storage File Syst.*, 2012.

[49] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," *ACM Trans. Storage*, vol. 6, 2010, Art. no. 14.

[50] Y. LuJ. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2013, pp. 257–270.

[51] C. Ji, L.-P. Chang, L. Shi, C. Wu, Q. Li, and C. J. Xue, "An empirical study of file-system fragmentation in mobile storage systems," in *Proc. 8th USENIX Workshop Hot Topics Storage File Syst.*, Jun. 2016, pp. 76–80. [Online]. Available: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ji

[52] Y. Lu, J. Shu, and W. Wang, "ReconFS: A reconstructable file system on flash storage," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 75–88.

[53] J. Zhang, J. Shu, Y. Lu, J. Shu, and Y. Lu, "ParaFS: A log-structured file system to exploit the internal parallelism of flash devices," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 87–100.

[54] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS - A new file system for flash storage," in *Proc. USENIX Conf. File Storage Technol.*, 2015. [Online]. Available: http://dblp.org/rec/conf/fast/LeeSHC15

[55] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proc. 17th Int. Conf. Architect. Support Program. Languages Operating Syst.*, 2012, Art. no. 387. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2150976.2151017

[56] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: An RDMA-enabled distributed persistent memory file system," in *Proc. USENIX Annu. Techn. Conf.*, 2017, pp. 773–785.

[57] E. Jeong *et al.*, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2014.

[58] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler, "An RDMA protocol specification," IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[59] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 133–146.

[60] X. Wu and A. L. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 39.

[61] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.

[62] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2901318.2901324

[63] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th Usenix Conf. File Storage Technol.*, 2016, pp. 323–338.

[64] M. Wilcox, "DAX: Page cache bypass for filesystems on memory storage," *Oct*, vol. 24, 2014, Art. no. 4.

[65] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system IOPS on low-cost, commodity hardware," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, Art. no. 69. [Online]. Available: http://dl.acm.org/citation.cfm?id=2503210.2503225

[66] S. Weil, "Bluestore: A new storage backend for ceph one year in," [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/20170323%20bluestore.pdf

[67] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 145–156. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren

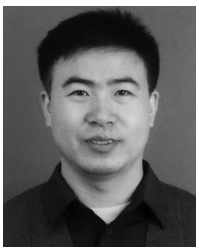
[68] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248. [Online]. Available: http://ieeexplore.ieee.org/document/7013007/

[69] "LevelDB, A fast and lightweight key/value database library by Google," [Online]. Available: https://code.google.com/p/leveldb/

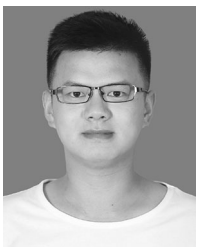
[70] P. Shetty, R. Spillane, and R. Malpani, "Building workload-independent storage with VT-Trees," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 17–30. [Online]. Available: https://www.usenix.org/system/files/conference/fast13/fast13-final165_0.pdf

[71] L. Pacheco, R. Halalai, V. Schiavoni, F. Pedone, E. Rivière, and P. Felber, "GlobalFS: A strongly consistent multi-site file system," in *Proc. IEEE Symp. Reliable Distrib. Syst.*, 2016, pp. 147–156.

[72] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 133–148.
[73] Y. Li, H. H. Chan, P. Lee, and Y. Xu, "Enabling efficient updates in kv storage via hashing: Design and performance evaluation," *ACM Trans. Storage*, vol. 15, pp. 1–29, 2019.
[74] Y. Chen, M. Yang, Y. Chang, T. Chen, H. Wei, and W. Shih, "Co-optimizing storage space utilization and performance for key-value solid state drives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 1, pp. 29–42, Jan. 2019.



Jiwu Shu (Fellow, IEEE) received the PhD degree in computer science from Nanjing University, in 1998, and finished the postdoctoral position research at Tsinghua University, in 2000. Since then, he has been teaching at Tsinghua University, and is currently a professor with the Department of Computer Science and Technology, Tsinghua University. His current research interests include network storage systems, nonvolatile memory-based storage systems, storage security and reliability, and parallel and distributed computing.



Fei Li received the BS degree in computer science and technology from Tsinghua University, in 2015. He is currently working toward the master's degree with the Department of Computer Science and Technology, Tsinghua University. His research interest includes flash-based storage system. One of his research work is published at the top-tier conference DAC, in 2019.

Siyang Li received the BS and MS degrees from the National University of Defence and Technology, in 2012 and 2015, respectively. He is currently working toward the PhD degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing and visiting PhD student of Tsinghua University. His research interest includes distributed storage system. His research works have been published at a number of top-tier conferences and Journal including SC and TPDS etc.

Youyou Lu received the BS degree in computer science from Nanjing University, in 2009, and the PhD degree in computer science from Tsinghua University, in 2015. He is an assistant professor with the Department of Computer Science and Technology, Tsinghua University. His current research interests include file and storage systems spanning from architectural to system levels. His research works have been published at a number of top-tier conferences including FAST, USENIX ATC, EuroSys, SC, MSST, ICCD etc. His research won the Best Paper Award at NVMSA 2014 and was selected into the Best Papers at MSST 2015. He was elected in the Young Elite Scientists Sponsorship Program by CAST (China Association for Science and Technology), in 2015, and received the CCF Outstanding Doctoral Dissertation Award, in 2016.