OSSD: A Case for Object-based Solid State Drives

Young-Sik Lee†, Sang-Hoon Kim†, Jin-Soo Kim§, Jaesoo Lee¶, Chanik Park¶, and Seungryoul Maeng†
{yslee, sanghoon}@calab.kaist.ac.kr, jinsookim@skku.edu, {jaesu.lee, ci.park}@samsung.com, maeng@kaist.edu

†Computer Science Dept.
KAIST
Daejeon, South Korea

§College of Info. & Comm. Eng.
Sungkyunkwan University
Suwon, South Korea

¶Memory Division
Samsung Electronics Co.
Hwasung, South Korea

***Abstract*—The notion of object-based storage devices (OSDs) has been proposed to overcome the limitations of the traditional block-level interface which hinders the development of intelligent storage devices. The main idea of OSD is to virtualize the physical storage into a pool of objects and offload the burden of space management into the storage device. We explore the possibility of adopting this idea for solid state drives (SSDs).**

The proposed object-based SSDs (OSSDs) allow more efficient management of the underlying flash storage, by utilizing object-aware data placement, hot/cold data separation, and QoS support for prioritized objects. We propose the software stack of OSSDs and implement an OSSD prototype using an iSCSI-based embedded storage device. Our evaluations with various scenarios show the potential benefits of the OSSD architecture.

I. Introduction

Solid state drives (SSDs) based on NAND flash memory are quickly gaining popularity. Compared to hard disk drives (HDDs), SSDs bring many benefits in terms of performance, shock and vibration resistance, power consumption, and weight. With these benefits, SSDs are widely used in various environments such as mobile devices and enterprise systems. However, due to the unique characteristics of NAND flash memory such that data cannot be overwritten until it is erased and the erase operation is performed in bulk, SSDs necessitate sophisticated firmware called *flash translation layer* (FTL).

The main role of the FTL is to emulate the block-level interface so that the existing HDD-based storage stack can be used for SSDs, hiding the peculiarities of NAND flash memory. FTL maintains a mapping table to translate the logical block address into the physical address in NAND flash memory. FTL also keeps track of whether each region of flash memory has valid data or not, in order to reclaim obsolete data by performing erase operations. Since NAND flash memory does not allow in-place update, FTL writes incoming data on an already erased region and marks the previous data as invalid. The space occupied by those obsolete data is reclaimed by the *garbage collection* procedure of FTL. In addition, FTL performs wear-leveling since NAND flash memory has the limitation in the number of erase operations that can be performed on the flash.

However, the simple block-level interface, which just allows to read or write data for a range of logical block addresses (LBAs), makes it harder to optimize SSDs. Although the block-level interface has successfully abstracted the internal architecture of storage devices, it now hinders the development of intelligent storage devices. The fundamental problem of the traditional block-level interface is that it is difficult for a storage device to optimize its internal resources due to the lack of higher-level semantics [26].

To overcome these limitations of the block-level interface, object-based storage devices (OSDs) have been proposed which virtualize physical storage into a pool of objects [25], [26]. An object is a variable-sized container that can store any kind of data ranging from flat files to database tables. Each object is associated with a set of object attributes which are used to describe the specific characteristics of the object. These objects are manipulated by file-like operations such as CREATE, DELETE, READ, and WRITE. In OSDs, space allocation to an object and free space management are handled by the storage device itself, freeing the file system from performing low-level block management. The first version of the OSD command set was standardized in 2004 as an extension of the SCSI command set by the INCITS T10 committee [13].

This paper explores a case of adopting the object-based interface for SSDs. We argue that the performance and reliability of SSDs can be improved when exploiting various object properties such as data usage, size, access pattern, priority, etc., of each object. Moreover, it is relatively easy to add the object management layer on top of the existing FTL in SSDs since FTL already performs a significant amount of work such as physical space allocation, address mapping, garbage collection, and wear-leveling. There are many potential benefits of the proposed object-based SSDs (OSSDs). Specifically, OSSDs can manage the underlying flash storage more efficiently, simplifying the host file system. Object attributes can be used to deliver application-level hints to the device so that OSSDs can perform more intelligent application-aware storage management. In particular, metadata separation, object-aware allocation, hot/cold data separation, and quality-of-service (QoS) support for prioritized objects are some examples of possible optimizations using the object-based interface.

We have developed an OSSD prototype and performed several experiments utilizing object properties. For fast prototyping and ease of debugging, we have taken an approach to implementing an OSSD firmware stack on the top of a special SSD hardware which exposes native flash interface. The firmware is executed on an ARM-based embedded storage device connected to the host machine via the iSCSI protocol. Our evaluation results with various scenarios show that SSDs with the object-based interface can improve performance in many aspects.

978-1-4799-0218-7/13/$31.00

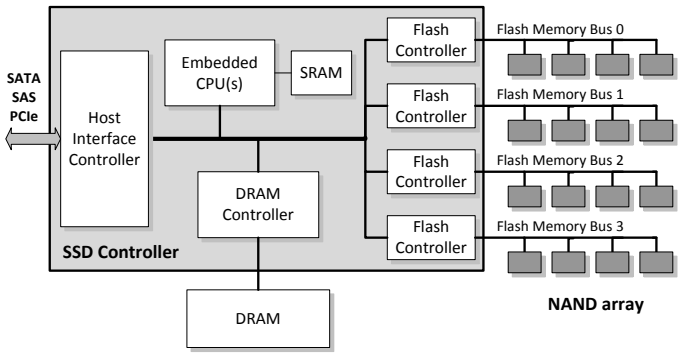


Fig. 1. The general architecture of an SSD.

The rest of this paper is organized as follows. Section II gives a brief overview of SSDs, NAND flash memory, and FTL. Section III discusses several approaches to interfacing SSDs with the host system. In Section IV, we describe the motivation of our work. Section V and Section VI present the implementation details and the performance evaluation results of our OSSD prototype with various scenarios, respectively. In Section VII, we conclude the paper.

II. Background

A. Solid State Drives (SSDs)

Figure 1 illustrates the general architecture of an SSD which consists of an array of NAND flash memory, DRAM, and an SSD controller. NAND flash memory is used as permanent storage for user data and DRAM is used to temporarily save user data and management data for NAND flash memory.

The internal architecture of an SSD controller is composed of flash controllers, a DRAM controller, one or more embedded CPUs, and a host interface controller. The host interface controller's main function is to support a specific bus interface protocol such as SATA, SAS, and PCIe. A DRAM controller and flash controllers are responsible for data transfer between DRAM and NAND flash memory using DMA control logic and guaranteeing data integrity based on ECC (Error Correction Code) algorithms such as Reed-Solomon (RS) and BCH. Finally, the embedded CPU(s) together with SRAM provide the execution environment for running flash management firmware called *flash translation layer (FTL)*. FTL parses host commands and translates associated logical block addresses (LBAs) to physical addresses on NAND flash memory based on a mapping table which will be explained in Section II-C.

In order to maximize SSD performance, the *multi-channel* interleaving is used with multiple flash controllers [34]. Each flash controller executes I/O operations for NAND flash memory chips with independent I/O bus (also called *channel*) and control signals. The flash controller also utilizes the *multi-way* interleaving over multiple NAND flash chips on a shared I/O bus using multiple chip enable signals [1]. The multi-channel and multi-way interleaving of NAND flash memory chips have been deployed as main techniques to improve not only sequential read and write performance but also random read and write performance.

TABLE I. Operational parameters of Samsung 2G x 8bit MLC NAND flash memory (K9GAG08U0M) [32]

Type	MLC (2-bit/memory cell)
Number of planes	2
Number of blocks / plane	2048
Number of pages / block	128
Page size	4224 bytes (4KB data + 128B spare)
Page read (t_R)	60 μs (max)
Page program (t_{PROG})	0.8 *ms* (typical)
Block erase (t_{BERS})	1.5 *ms* (typical)

B. NAND Flash Memory

To better understand the challenges facing development of SSDs, it is essential to understand the inherent characteristics of NAND flash memory. A NAND flash memory chip is packaged with multiple *dies*. A die has 1, 2, 4, or 8 *planes* and each plane has the same number of *blocks* or *erase blocks*[1]. Each erase block in turn consists of 64 or 128 *pages*. In addition to the data area whose typical size is either 4KB or 8KB, each page is provided with extra space called *spare area* that can be used for storing ECC and other bookkeeping information.

There are three major operations supported in NAND flash memory: *read*, *write* (or *program*), and *erase*. The read and program operations are performed on a per-page basis. To reduce the program disturb phenomenon [29], NAND flash manufacturers recommend programming each page only once and in sequential order within an erase block. Once a page is programmed, the page can be overwritten only after erasing the entire erase block to which the page belongs. Note that a set of pages (or erase blocks) from different planes can be read or programmed (or erased) simultaneously, effectively increasing data throughput. Unlike other semiconductor devices such as SRAM and DRAM, NAND flash memory has asymmetric operational latencies; page program or block erase operation is slower than page read operation by over an order of magnitude. NAND flash memory also has the limitation that each erase block can tolerate a limited number of program/erase cycles.

NAND flash memory comes in two flavors: SLC (Single-Level Cell) and MLC (Multi-Level Cell). SLC NAND flash memory stores only 1 bit per memory cell, while the newer MLC NAND flash memory represents two bits per cell allowing for higher density with lower cost. However, since MLC NAND flash memory manages four different states within a single memory cell, it takes longer to read or program a page, and incurs more bit errors as compared to SLC NAND. In addition, each erase block in MLC NAND can tolerate only 5K $\sim$ 10K program/erase cycles, which is shorter than the limit (typically, 10K $\sim$ 100K) in SLC NAND. For these reasons, SLC NAND is mostly used for industrial and enterprise storage, while MLC NAND for consumer and embedded storage. Table I summarizes operational parameters of Samsung 2GB MLC NAND flash memory [32], which is used in our OSSD prototype described in Section V.

C. Flash Translation Layer (FTL)

The main role of the FTL is to conceal the peculiarities of NAND flash memory and to provide the traditional block-level

[1]To avoid confusion, we use the term '*erase block*' to refer to the erase unit in NAND flash memory.

interface to upper layers, while enhancing performance and reliability of SSDs. As we see in Section II-B, there are several obstacles in emulating the block-level interface on NAND flash memory: (1) in-place update is not allowed, (2) the entire pages within an erase block should be erased in bulk, and (3) block erase takes a relatively long time.

To overcome these difficulties, most FTLs employ an *address mapping scheme* which decouples the logical block address (LBA) used in the upper software layer from the physical page address where the actual data is stored in flash memory. Although there are many different address mapping schemes, the common technique is to reserve a set of erase blocks, or *update blocks*, in advance, and then use these update blocks to absorb incoming write requests from the host. While a new data is written into the update block, the previous version of the data is simply marked obsolete. When a certain amount of update blocks is exhausted, a process known as *garbage collection* is invoked to reclaim obsolete pages. Garbage collection is performed in three steps: (1) a victim erase block is selected based on a policy such as greedy [31] or cost-benefit [5], (2) valid pages in the victim erase block, if any, are copied into the current update block or another temporary erase block, and (3) the victim erase block is erased and then converted into a new update block.

Depending on the address mapping granularity, FTLs are classified into page mapping, block mapping, and hybrid mapping. In page-mapping FTLs [6], [9], [23], each logical page can be freely mapped to any location on flash memory at the expense of consuming more memory to keep track of page-level mapping information. In spite of the larger memory footprint, the recent trend in SSDs is to shift towards the use of page-mapping FTL, especially in order to increase the random write performance. On the contrary, block-mapping FTLs maintain only the erase block-level mapping information by imposing a restriction that a set of consecutive logical pages should be stored in the same erase block. Hybrid-mapping FTLs [15], [21], [22] attempt to achieve higher performance than block-mapping FTLs through the selective use of page-level mapping only for a small number of update blocks.

FTLs should also ensure the reliability and integrity of SSDs. FTLs maintain a list of bad erase blocks and a new bad erase block is remapped at runtime to one of free erase blocks reserved for this purpose. To cope with the limited program/erase cycles, FTLs also perform *wear-leveling* which distributes erase operations evenly across the entire flash memory erase blocks. Finally, all the FTL metadata should be able to recover from sudden power failure.

III. Related Work

In this section, we briefly discuss several approaches to interfacing flash-based solid state storage with the host system.

A. Using the Native Flash Interface

Flash-based storage can expose the naive NAND flash memory interface to the host, namely PAGE READ, PAGE PROGRAM, and BLOCK ERASE operations. Actually, this architecture is being widely used in embedded systems, where bare NAND chips are glued onto printed circuit boards with a simple NAND controller. The host may use the existing block-based file systems as long as the operating system implements a device driver which is responsible for FTL functionalities. As an alternative, the host system may use flash-aware file systems such as JFFS2 [40], YAFFS2 [3], and UBIFS [12] which manage the flash storage directly using the native flash interface.

Josephson et al. have recently pursued a similar approach [14]. They propose a simplified file system called DFS (Direct File System) based on the virtualized storage abstraction layer. The virtualized storage abstraction layer provides a very large, virtualized block address space on top of PCIe-based flash storage which exposes direct access to flash memory chips.

Using the native interface, the file system can manage the underlying NAND flash memory efficiently since it may leverage higher-level semantics, while taking into account the characteristics of the underlying flash storage. However, it is difficult to optimize flash file systems or flash device drivers for various NAND flash memory configurations. For example, UBIFS requires rewriting read/write codes for NAND flash chips whose page size is larger than 4KB, since it uses a single kernel page for I/O.

B. Introducing a New, Simplified Interface

The NVMHCI (Non-Volatile Memory Host Controller Interface) Work Group has standardized the register-level interface for platform non-volatile memory solutions called the NVM Express (NVMe) [27]. In fact, NVM Express is largely targeted to PCIe-based SSDs. Since they are directly attached to the host system via PCIe channels, they have no reason to use ATA/SCSI commands, leading to the development of a new interface standard. Although the host can indicate several attributes for ranges of NVM pages via the optional data management command, the main limitation of NVMe is that it still works at the block level.

C. Modifying the Existing Block-level Interface

Recently, the TRIM command is standardized as part of the ATA interface standard [36]. When a file is deleted, the TRIM command explicitly informs the underlying SSD which sectors are no longer in use. This information can drastically improve the efficiency of garbage collection, as otherwise the actual data of the deleted file needs to be copied during garbage collection until the corresponding sectors are reused for another file [35]. Apparently, this is only an interim solution based on the block-level interface for maintaining backward compatibility. To implement a more intelligent storage device, more higher-level semantics such as file access pattern, file size, and file attributes are required to transfer.

Mesnier et al. [24] introduced an I/O classification architecture to transfer higher-level semantics to the storage system. They suggested new schemes for classifying data in the ext3 file system and delivering such information to the storage device via the SCSI Group Number field in the SCSI commands. The predefined data classes are based on data usage (e.g., metadata, journal, or directory) and file size, and the storage system utilizes such data classes to improve its performance. However, they had to set the predefined data

class number for every request. In addition, the SCSI Group Number field is insufficient to represent all of the higher-level semantics.

D. Using the Object-based Interface

Rajimwale et al. [30] suggested OSD as an improved interface for SSDs since the current block-level interface cannot satisfy new requirements intrinsic to SSDs. Unlike rotational and mechanical HDDs, OSD-enabled SSDs can manage space based on the internal architecture of SSDs. In addition, object attributes can be utilized to support quality-of-service (QoS) for prioritized objects (e.g., multimedia files). They claimed that a richer interface, OSD, is well-matched with SSDs. However, quantitative analysis with a matching storage software stack has not been investigated over diverse use case scenarios.

Kang et al. [17] presented the object-based storage model for SCM (Storage Class Memory) and revealed many advanced features of object-based SCM. They used metadata separation for data placement policy to reduce cleaning overhead. In addition, they showed the effect of object-level reliability to detect and correct more errors. Although many concepts are similar to our work, we focus on how to improve the performance of storage system by object-aware data placement, hot/cold data separation, and QoS support for prioritized requests. We also introduce the full architecture for OSSDs and implement the overall storage stack in a realistic environment with an iSCSI-based embedded storage device.

IV. Object-based Solid State Drives (OSSDs)

A. Object-based Storage Devices (OSDs)

In OSDs, *objects* are used to abstract physical storage management and *attributes* to convey and interpret application-level semantics. An object is a variable-sized entity that can store any kind of data including text, images, audio/video, and database tables [26]. Objects are grouped into partitions and each object is uniquely identified by an object identifier (OID), which consists of a 64-bit PARTITION_ID and a 64-bit USER_OBJECT_ID.

A number of attributes can be associated with an object, which are used to describe specific characteristics of the object. Several predefined attributes including the object size and the object timestamps are maintained by the device. Other attributes can be used freely by applications to specify various application-specific semantics such as quality-of-service (QoS) requirements.

The OSD standard defines a fine-grained, per-object security mechanism. Each object is accessed by a cryptographically signed capability which specifies the object and allowable operations. This ensures that only trusted clients can access the objects, allowing an OSD to be directly connected to a public network.

The interface to object-based storage is very similar to that of a file system. Objects are explicitly created, deleted, read, or written using CREATE, DELETE, READ, and WRITE commands, respectively. Attributes can be specified or queried by SET_ATTR and GET_ATTR commands. In its simplest form, a file can be mapped to a single object. It is also possible to stripe a large file across multiple objects to improve performance and reliability, or to map several small files into an object to reduce metadata overhead [26]. In any case, allocating storage space to an object or managing free space is now handled by the storage device itself, freeing the file system from performing low-level block management.

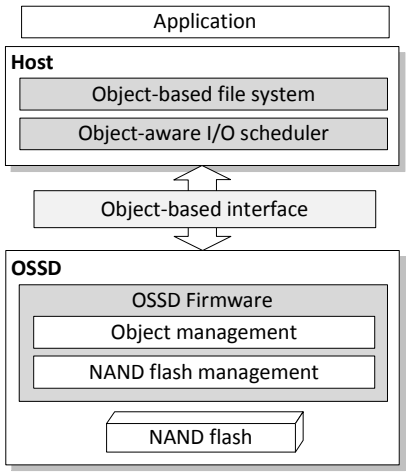


Fig. 2. The overall system architecture of OSSDs.

OSDs enable a scalable and secure data sharing in a large-scale networked environment, as exemplified by Lustre [38] and Ceph [39]. These distributed file systems consist of a metadata server (MDS) and a set of OSDs, which are simultaneously accessed by a large number of clients. In this case, delegating space management to OSDs improve scalability as the MDS no longer needs to coordinate metadata updates. Also, objects can be directly transferred from multiple OSDs to a client in parallel and in a secure fashion.

Another benefit is that the use of the object-based interface can improve the intelligence of storage devices. In this paper, we focus on this aspect of OSDs, especially in the context of SSDs.

B. Overall Architecture of OSSDs

Figure 2 illustrates the overall system architecture of object-based SSDs (OSSDs). The object-based file system accepts file system operations from applications and translates them into object operations which involve objects and object attributes. Since the low-level block management is done in the OSSD, the object-based file system is far simpler than block-based file systems. To merge and schedule a set of object operations, an object-aware I/O scheduler is needed because the current I/O schedulers work only at the block level.

Objects and object attributes delivered via the object-based interface are managed in the OSSD firmware. The OSSD firmware also controls the underlying NAND flash memory with various functions such as data placement, garbage collection, bad erase block management, and wear-leveling. The OSSD firmware has knowledge of object properties delivered by the object-based interface. Moreover, since OSSDs already know the characteristics of NAND flash memory, they can manage NAND flash memory efficiently, thus enhancing the performance and reliability of the storage system.

C. Why Object-based SSDs?

The object-based interface offers great potential for SSDs. Compared to HDDs, SSDs perform a significantly more amount work related to space management due to the nature of NAND flash memory. The performance and reliability of SSDs can be improved when we move from the block-level interface to the object-based interface and it is relatively easy to add the object management layer on top of the existing FTL in SSDs. The block-level interface is also a feasible solution to transfer higher-level semantics if it is modified or expanded. However, the object-based interface provides a general abstraction layer to manage objects and can deliver a number of object properties easily. In this section, we examine the potential benefits of the proposed object-based SSDs (OSSDs).

Simplified Host File System Since the introduction of NAND flash-based SSDs, there have been many efforts to optimize the traditional block-based file systems [35] and I/O schedulers [20] for SSDs. Unfortunately, the host-side SSD-aware optimizations are hard to achieve. This is because the performance of an SSD is dictated by many parameters such as SSD page size, the degree of multi-channel and/or multi-way interleaving, DRAM buffer size, the amount of over-provisioning (i.e., the percentage of erase blocks reserved for bad erase blocks and update blocks), address mapping scheme, garbage collection and wear-leveling policies, etc. Although some of these parameters can be extracted by a carefully designed methodology [19], many of them are still difficult to acquire and vary widely from SSD to SSD.

Because only the SSD knows exactly about its internal architecture and tuning parameters, it is desirable to delegate most of space management to the SSD. This can greatly simplify the host file system. Previously, the pathname is mapped to an inode and the <inode, offset> pair is mapped to a logical block address which is mapped again to a physical page address by the SSD. In OSSDs, this access path can be shortened; the pathname is now mapped to an object id (OID) and the OSSD directly maps <OID, offset> pair to a physical location in NAND flash memory. Therefore, with OSSDs, the file system is only responsible for pathname-to-object mapping and access control.

Utilizing Liveness Information The *block-level liveness* information can be utilized for optimizing on-disk layout, smarter caching, intelligent prefetching, etc., leading to the development of smart storage systems [37]. For OSSDs, it also helps to identify flash pages that hold obsolete data, thereby improving the garbage collection performance and the device lifetime. OSSDs have complete knowledge of whether a given data block contains valid data or not by the object-based interface. The block-based storage system relies on the TRIM command to gather the block-level liveness information. However, a single TRIM command can specify only a range of logically consecutive sectors. When a large, highly-fragmented file is deleted, the file system may produce an excessive number of TRIM commands [33].

Metadata Management It is also easy for OSSDs to classify data blocks according to their usages. As the file system metadata is small but updated frequently, FTL designers have long sought a way to separate metadata writes from normal data writes to handle them differently [17], [41]. However, it turns out to be very difficult without guessing the file system layout. Mesnier et al. show that selective caching of metadata, journal, directories, and the first 256KB of each file in an SSD results in the speedup of up to 173% in their hybrid storage system [24]. Such optimizations are all possible in OSSDs without incurring significant overhead since the storage device manages metadata by itself.

In addition, the block-based file system must deliver the file system metadata such as data block bitmaps and inode bitmaps to the storage device for the layout management. In OSSDs, the file system does not have to transfer the metadata for storage layout management, because the layout is managed inside of the storage device.

Object-aware Data Placement Another advantage is that OSSDs know which data blocks are originated from the same object. This information can be used to rearrange all the data blocks from an object in optimal locations that can maximize the sequential read bandwidth to the object. OSSDs also can reduce the garbage collection overhead by placing the data from the same object into the same update block. When the object is deleted, a large number of consecutive pages becomes invalid at once, hence minimizing storage fragmentation. Similarly, OSSDs can group semantically-related objects together. For example, objects accessed during the boot process may be grouped together to ease prefetching.

Hot/Cold Data Separation In SSDs, separating hot data from cold data is known to be very important to reduce the amount of valid pages in the victim erase block, which directly affects the efficiency of garbage collection and wear-leveling [6]. Therefore, the expected access pattern on a specific object, such as update frequency, sequentiality in object access requests, average request size, and object lifetime, can be used effectively by OSSDs to manage their flash storage. For example, if the object is frequently accessed, it generates many invalidated pages on flash memory. In this case, OSSDs can reduce the amount of valid page copies during garbage collection by dedicating an update block for the frequently updated object.

The block-based storage can also gather such access patterns, but it has significant overheads due to the lack of accurate information on the semantics among the data. If it manages access counts to identify the access pattern of the data in a fine-grained unit (e.g., sectors), it requires a large memory space in the firmware. When using a coarse-grained unit (e.g., a set of consecutive sectors), it cannot distinguish the data that has different access patterns. However, OSSDs can handle this information on an object basis with low overhead.

QoS Support for Prioritized Objects Object attributes may be used to specify quality-of-service (QoS) requirements. For example, audio/video players, digital camcorders, or PVRs (personal video recorders) have certain levels of bandwidth and latency requirement for real-time playback or recording. OSSDs can use this information to schedule its internal resources. Since garbage collection is usually performed in the background, a real-time read request should be able to preempt an ongoing garbage collection task. Real-time write

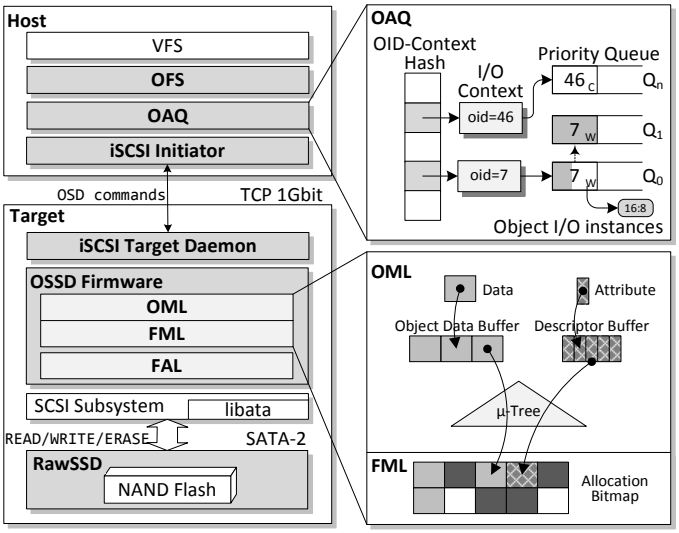


Fig. 3. The OSSD prototype system architecture.

requests are more difficult to handle, as they will periodically invoke garbage collection to make new erase blocks. Some special polices such as reserving a number of update blocks or dedicating one or more update blocks to a real-time object may be used to make real-time requests to meet their deadline.

V. OSSD Prototype Implementation

A. OSSD Prototype Architecture

We have developed a prototype to quantitatively assess the benefits of the proposed OSSD architecture. For accurate experiments including the data and command transfer overheads, we use two machines connected via the iSCSI protocol over the network. The prototype covers the entire storage software stack encompassing our own object-based file system (OFS), the object-aware I/O scheduler (OAQ) in the host machine, and the OSSD firmware in the target machine as well as the actual SSD hardware. For fast prototyping and ease of debugging, the OSSD target is implemented as a kernel module on the ARM-based embedded storage device originally designed for Network-Attached Storage (NAS). Figure 3 depicts the overall architecture of our OSSD prototype.

On the host machine, we have implemented an object-based file system to provide the standard POSIX file system interface on top of the OSSD prototype. The object-aware I/O scheduler processes I/O requests from OFS at object-level to merge I/O requests to the same object and to handle prioritized requests. By merging requests, OAQ can reduce the number of I/O operations and increase the bandwidth of the OSSD prototype. OAQ also expedites the handling of requests for an important object by using prioritized requests.

In the OSSD target, there are three layers for storage management. The Object Management Layer (OML) maintains various data structures, such as object mapping information and attributes, needed to manage a large number of objects. The physical storage space on NAND flash memory used by OML is managed by the Flash Management Layer (FML). In fact, the role of FML is similar to that of FTL in a conventional SSD except that address mapping is now done in OML. Hence, FML is only responsible for flash storage management such as page allocation, garbage collection, and wear-leveling. Flash operations from the FML are converted to actual flash I/O operations and sent to the hardware flash controller by the Flash Abstraction Layer (FAL).

We have developed a special SSD hardware platform called *RawSSD*. Unlike the conventional SSDs, we simplify the RawSSD's firmware so that it is only responsible for bad erase block management, exposing the native flash operations such as READ, WRITE, and ERASE to FAL. The actual unit of read and write operation in RawSSD is 16KB since two dies, both enabled with the two-plane mode, are combined together. To maximize parallelism during sequential reads/writes, we also form a virtual erase block by combining the total four erase blocks, each from a different channel. Therefore, a region of 8MB (16x erase block size) is erased at once by a single erase operation (cf. Section V-H).

In the following subsections, we describe the implementation details of each layer in turn from OFS to RawSSD.

B. OFS: Object-based File System

OFS is based on the EXOFS file system [11], the only object-based file system implementation in the Linux kernel since version 2.6.30. As in EXOFS, OFS maps each file to a single object. The object id (OID) is acquired statically by adding a user-object id offset (0x10000) to the inode number of the file. A directory is mapped to an object as well, which contains filename-to-OID mappings. The object for the root directory is allocated and initialized when the file system is created.

File metadata such as size, ownership, and timestamps are stored in the object attributes. OFS reconstructs inodes in memory by referring to these attributes. File operations are converted into the corresponding object operations called *object I/O instances* (OIO's) and they are submitted to the underlying object-aware I/O scheduler.

The current implementations of OFS and EXOFS do not support metadata reliability. Although our implementation does not have a metadata journaling mechanism, journaling support can be implemented efficiently with an attribute indicating transactional object and log-style NAND flash write behavior [28].

C. OAQ: Object-aware I/O Scheduler

OAQ implements the object-aware I/O scheduling layer in the OSSD prototype architecture. The *page collector* in EXOFS performs a very primitive object-aware I/O scheduling. It simply queues the sequential requests for an object. If there is a non-sequential request or a request to the other object, the queued requests are flushed. This impairs the I/O performance badly especially when multiple objects are accessed simultaneously. On the contrary, OAQ allows I/O requests to multiple objects to be queued in the scheduler.

Basically, OAQ merges *object I/O instances* (OIO's) produced by OFS into *object requests* (OREQ's). As shown in Figure 3, OAQ uses a priority queue of OREQ's, where OREQ's with the same OID are linked together and summarized into a per-object *I/O context*. For fast lookup, the I/O contexts are indexed by OID through the hash table with 16 buckets.

As OFS submits an OIO, OAQ looks up the corresponding OREQ that is destined to the same OID and has the same priority with the OIO. If the OREQ is contiguous to the current OID and the request size is less than 256 sectors[2], the OIO is merged with the OREQ. If not, a new OREQ is added to the queue.

OFS allows to set the priority of an object using the `ioctl()` system call. OAQ puts an OREQ in a different queue according to this priority. The OREQ in the head of the highest priority queue is dispatched by the underlying object management layer (OML).

D. Host-Target Connection

Every request from OAQ are translated into a proper set of OSD commands and are delivered to the OSSD target using the iSCSI protocol over the IP-based network. We use open-osd [10] and Open-iSCSI [2] kernel modules to generate and deliver OSD commands in the host machine. In the target device, tgt [8] user daemon receives requests from the host machine. After parsing the OSD command, requests are passed to OML in the kernel via the `ioctl()` system call.

E. OML: Object Management Layer

In the object management layer (OML), there are two data that have to be managed. One is object attributes which contain object metadata such as size, access time, and other inode information. The other is the object mapping information of the object data from <OID, logical offset> to the physical position on RawSSD. These data structures are indexed with μ-Tree [16] for fast lookup and flexible management. μ-Tree is an efficient index structure to manage complex data on native flash media. μ-Tree is a tree-based data structure similar to B+-tree, but it produces only one page write for inserting a single record while the original B+-tree requires many page writes due to cascaded node updates.

In our prototype, an object descriptor contains object information similar to the inode of file system. It also contains attributes of an object. Each object attribute is represented as a <attribute id, value length, attribute value> tuple. These tuples are placed in the attribute section of the object descriptor. The size of each object descriptor is 256 bytes for now. A more flexible representation of the object attributes is left as future work. Since the unit of read/write operation of RawSSD is larger (16KB) than the descriptor size, several descriptors are coalesced in the descriptor buffer. The indexing from the OID to the physical location of the descriptor is stored in a μ-Tree.

The index structure also contains the mapping information to find the physical location of the object data. We use an extent mapping scheme, which represents the region of physical flash pages mapped to the region of an object as a single entry such as: <OID, logical offset> $\rightarrow$ <flash page number, size>. This mapping information is also stored in the μ-Tree.

Before assigning a physical location of each object data, the data is stored in the object data buffer. The object data buffer absorbs repeated accesses to the same location as in the typical storage system. In addition, the object data buffer aligns the incoming requests so that they can be processed efficiently in RawSSD. Many requests from the host machine may be smaller than RawSSD's write unit (16KB), not to mention that they are rarely aligned to the 16KB boundary. In these cases, a lot of read-modify-write operations are incurred to handle small and misaligned requests. The object data buffer remedies this problem, and according to our measurement, it improves the bandwidth of sequential write by 46.8%.

It is impractical to keep the entire object descriptors in memory, since there will be a huge number of objects in an OSSD and OML should be able to run in a device with scarce resources. For these reasons, OML keeps only a portion of object descriptors in memory using the LRU policy. By default, OML and μ-Tree use up to 8MB of memory for caching object descriptors and μ-Tree nodes. The object data buffer uses up to 2MB of memory by default and it is managed with the FIFO replacement policy. When OML receives the FLUSH command which is generated by `sync()` in the file system, it flushes all the modified data in the μ-Tree cache, the object descriptor buffer, and the object data buffer.

F. FML: Flash Management Layer

To service space management, the Flash Management Layer (FML) maintains a bitmap which indicates whether a certain page has valid data or not (cf. Figure 3). When OML requests for a free page, FML allocates one and marks it as being used. If OML notifies that the page is no longer in use, the page is marked as invalid and reclaimed later.

As separating object metadata from object data results in a better storage utilization [17], [41], FML maintains three update blocks for object descriptors, μ-Tree nodes, and object data. Furthermore, FML can utilize intelligent data placement policies using various object properties as described in Section IV-C.

In our prototype, the FML has two data allocation policies. To place data from the same object consecutively in the physical location, FML dedicates an update block to each object. In this case, a big chunk of flash pages can be invalidated if the object is deleted, improving the performance of garbage collection. Since this policy is effective when the large data is written sequentially, we apply the policy when the object size is larger than 8MB which is identical to the virtual erase block size of RawSSD. For small-sized objects, FML preserves two update blocks to separate hot/cold data. FML records the write access count in each object descriptor, and then if the access count of an object is greater than τ, the next write for the object is assigned to the dedicated update block for absorbing hot accesses. We set the threshold value τ as $2 \times (object\ size / write\ unit\ size)$ since the write accesses are counted by the unit of the write operation (16KB). If the write access count of an object is smaller than τ, FML allocates a new flash page from the other update block. FML decreases the write access count if any of the flash pages of the object is copied during garbage collection. The object can be regarded as being updated infrequently if some of its flash pages are not updated until the corresponding erase block is selected as a victim by the garbage collection procedure.

As the number of free erased blocks falls below the low watermark (10, by default), FML triggers garbage collection. The

[2]This is the max request size imposed by our prototype hardware, RawSSD.

background garbage collection continues to reclaim invalid pages until the number of free erased blocks reaches the high watermark. Unlike the low watermark, the high watermark value dynamically varies depending on the *I/O idleness*. FML uses the presence of requests in the object data buffer as an indicator of I/O idleness. If there is at least one request in the buffer, the high watermark is set to minimum (11, by default) so as not to hurt the foreground I/O performance. Otherwise, the high watermark is set to maximum (15, by default) to fully utilize idle cycles. FML uses a greedy policy [31], i.e., the erase block containing the largest number of invalid pages is chosen as the victim block to reclaim. Note that a higher-priority request can be serviced between two successive valid page copies while garbage collection is in progress for guaranteeing QoS.

G. FAL: Flash Abstraction Layer

The Flash Abstraction Layer (FAL) provides five low-level operations to the *backend* device such as RawSSD: READ PAGE, WRITE PAGE, ERASE BLOCK, FORMAT, and GEOMETRY. The first three operations correspond to the primitive NAND flash operations. The FORMAT command initializes the flash storage by erasing every erase block. The GEOMETRY command obtains configuration parameters such as the number of banks, the number of erase blocks per bank, the number of pages per erase block, the number of sectors per page, the number of channels, and the number of dies per channel, and so on. These operations are translated into RawSSD-specific ATA commands described in Section V-H.

H. RawSSD

In order to enable the development of OSSD firmware for the sake of debugging and fast prototyping, we have developed an SSD named *RawSSD* which provides a native NAND I/O interface instead of the traditional block-level interface. The interface provided by RawSSD includes commands for READ PAGE, PROGRAM PAGE, ERASE BLOCK, FORMAT, and GEOMETRY operations. To support these operations, we have augmented the ATA command set with some extended commands using 0x8C as the command ID. Since RawSSD has hardware identical to a normal SSD and allows for a fine-grained control over its hardware, it serves as a perfect platform for the development of SSD firmware.

The RawSSD hardware is newly developed based on a normal SATA SSD and its architecture is virtually same as the one shown in Figure 1. More specifically, the ARM-based processor with a small SRAM provides an execution environment for the RawSSD's firmware. RawSSD has 16 NAND chips each of which contains eight 2GB dies shown in Table I, providing the total capacity of 256GB. These chips are organized into eight flash memory buses. Each bus operates at 40MHz, yielding 40MB/s of transfer bandwidth for each bus. Since we combine two flash memory channels and use two paired flash memory dies from each channel as logically one unit with two-plane operation, the RawSSD's virtual page size is 16KB and its virtual erase block size is 8MB. Our RawSSD is connected to FAL via the SATA-2 interface.

A small DRAM buffer is used for asynchronous write operations. We use 32 buffer entries. The size of each buffer entry is 64KB, meaning that the data of four adjacent virtual pages can be fit in a single buffer entry. The buffer replacement policy should preserve the in-order program constraint of NAND flash memory. To do so, we maintain buffers using the FIFO replacement policy for simplicity. When an entry in the buffer is hit by either read or write command, every entry arrived before the hit entry is written to flash memory before the current read or write is served. Note that a read request is always handled in a synchronous way.

For the improved sequential throughput, we incorporate a page-wise channel-level striping technique. Logical page numbers are assigned in a round-robin fashion between physical pages in different logical channels. When a sequential read or write is requested, it is handled in an interleaved manner among chips, each from a different flash memory channel. RawSSD also handles bad erase blocks, by reserving 1.5% of the total erase blocks.

I. Legacy Stack Support

For fair comparison with the conventional storage architecture based on the block-level interface, we have implemented a simple page-mapping FTL named *sFTL* over RawSSD as a kernel module on the target device. It also uses the iSCSI protocol and tgt user program to receive requests over the network. sFTL also uses the data buffer to absorb repeated accesses and align incoming requests as in the OSSD prototype implementation. We intentionally let sFTL use the same policies for garbage collection as those implemented in the FML. With the help of sFTL, we can run any block-based file system on the target device, including the ext4 file system used in Section VI.

VI. Evaluation

A. Methodology

We perform several experiments to verify the potential benefits of OSSDs as described in Section IV-C. Among them, we concentrate on object-aware data placement, hot/cold data separation, and QoS support because the other benefits are treated by many previous work [17], [24], [35], [41]. For each scenario, we make appropriate benchmarks and execute them on top of the OSSD prototype. The details of each benchmark and the results of our experiments are described in the following subsections.

All of our experiments are carried out on a host machine equipped with an Intel Core2Quad Q9650 3GHz processor and 4GB RAM running the Linux kernel 3.0.3. The target device is based on the Linux kernel 2.6.32 with a 1.6GHz ARM processor and 512MB RAM. These two devices are connected via the Gigabit Ethernet. Although the total capacity of RawSSD is about 256GB, we only use 4GB of space to initiate garbage collection quickly.

We compare the performance of the OFS file system with default configurations of the OSSD prototype (OFS) with that of the ext4 file system built on top of the sFTL (EXT4) explained in Section V-I. EXT4 with the block-level interface is the representative case that does not deliver or utilize higher-level semantics. We configured EXT4 to issue the discard (TRIM) operation to the target device when a file

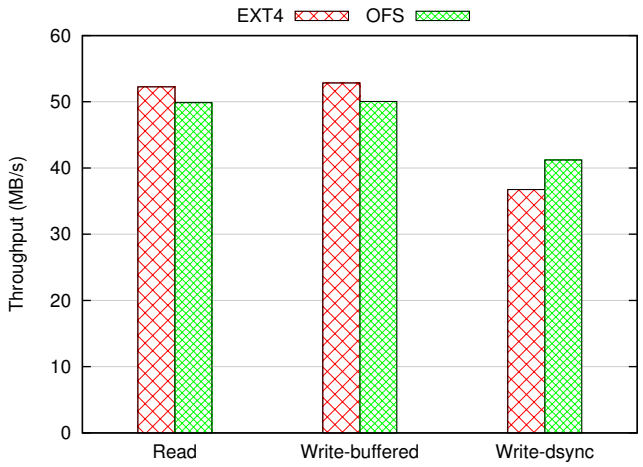
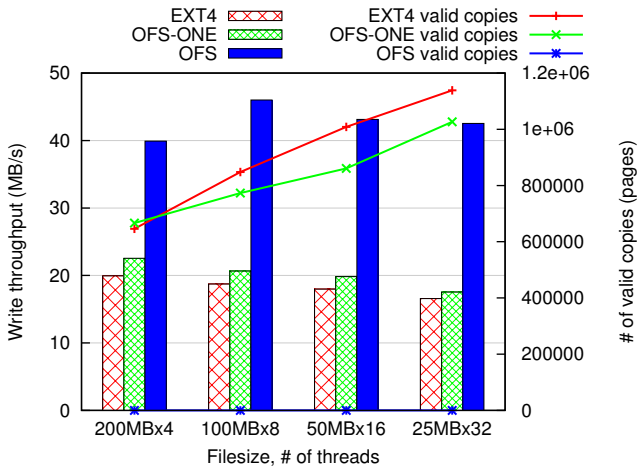
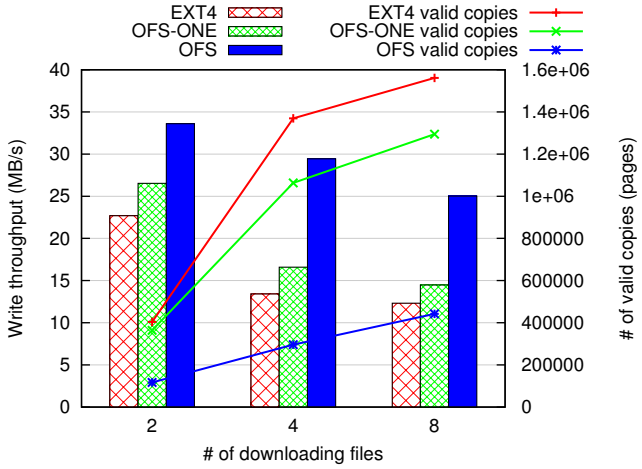


Fig. 4. A comparison of the base performance.



(a) Multi-threaded write benchmark.



(b) Torrent benchmark.

Fig. 5. Results of multi-stream benchmarks.

is deleted. Without the discard operation, EXT4 showed very low performance. Since OFS does not perform journaling, we have redirected the journaling traffic of EXT4 to a RAM drive for fair comparison. To investigate the impact of the object-aware data placement policy, we also measure the performance when all objects share a single update block (OFS-ONE).

B. Base Performance

Figure 4 illustrates the sequential performance of OFS and EXT4 on the formatted RawSSD. We measure the throughput of reading or writing 1GB of data using sequential requests with 128KB I/O size. We also measure the throughput for synchronous write using `fdatasync()` (Write-dsync) to analyze the overheads of data transfer in detail. All experiments issue `sync()` at the end to flush data in the page cache.

OFS has the overhead for read and buffered write access since it has to generate and parse OSD commands. More optimization of the request path including the I/O scheduler can reduce the performance gap between OFS and EXT4.

The throughput of read operations is less than the maximum read bandwidth of RawSSD (107MB/s). Because the read operations are handled in a synchronous manner, the actual throughput suffers from the latency of delivering commands over the network and the overhead to parse OSD commands. On the other hand, since write operations can process in an asynchronous manner in the target device firmware, it can result in near the full bandwidth of RawSSD (53MB/s). For synchronous write operations (Write-dsync), it has lower performance than the buffered write case (Write-buffered) due to the command overhead.

In case of EXT4 with `fdatasync()`, it has lower performance than OFS. When a program issues synchronous operations, EXT4 requires write operations for metadata blocks. Although EXT4 does not have to write inode with `fdatasync()`, it still needs to write at least one data block bitmap for every 128KB I/O request. The `fdatasync()` function flushes data in the page cache and updates the usage information of block bitmap for every request. Note that journaling traffic of EXT4 is not delivered to the underlying NAND flash memory since it is redirected to the RAM drive. Due to these metadata writes, EXT4 has lower throughput than OFS in the case of Write-dsync. However, OFS does not have to write this information since block allocation is done in the OSSD firmware internally. Using the object-based interface, the metadata write operations needed for space management are unnecessary.

C. Object-aware Data Placement

In this subsection, we evaluate the improvement under the proposed object-aware data placement policy. We prepare two multi-stream write benchmarks to investigate the performance results of object-aware data placement when several objects are written simultaneously.

The first benchmark is a multi-threaded write benchmark. Before measuring any results, this benchmark creates a number of files with the predefined size until all files occupy 3.2GB of space. Then the benchmark deletes a total of 800MB by selecting files randomly. Next, the fixed number of threads writes to a file dedicated to each thread. The amount of total space written at this phase is also 800MB. The benchmark repeats deleting and writing phases until the total amount of write becomes larger than 24GB, which generates a lot of garbage collections. The file size is varied from 25MB to

200MB and the number of threads from 4 to 32, as shown in Figure 5(a).

The other benchmark replays traces collected by using a Windows version of μTorrent [4] to evaluate the performance in a realistic scenario. The traces are captured while downloading 2, 4, and 8 files concurrently with the average size of 400MB in the μTorrent program. Whenever the disk usage exceeds 3.2GB and downloading of a file completes, we delete a file randomly selected from the already downloaded files. Then, we download another file again to keep the number of downloaded files constant. We also repeat deleting and downloading phases until the total amount of write becomes larger than 24GB as in the multi-threaded write benchmark.

Figure 5 shows the performance results of object-aware data placement over two multi-stream write benchmarks. The throughput results are represented in the bar graphs with the left axis and the number of valid page copies during garbage collection corresponds to the line graphs with the right axis.

In the Figure 5(a), OFS outperforms OFS-ONE and EXT4 in all cases. This is because OFS reduces the number of valid copies during garbage collection due to the per-object allocation policy in the FML. Although the multi-threaded write benchmark makes write operations on several objects concurrently, OFS can place data from the same object in the physically contiguous area on flash memory. Therefore, OFS can find a big chunk of invalidated flash pages easily if a file is deleted, which significantly reduces the number of valid page copies during garbage collection. We note that the number of valid copies of OFS in the multi-threaded write benchmark records *zero*. OFS-ONE shows slightly better performance than EXT4 because of the reduced metadata write overhead as mentioned in Section VI-B.

OFS also exhibits better throughput than OFS-ONE and EXT4 in Figure 5(b) which depicts the results of the torrent benchmark. In this case, we note that the number of valid copies for OFS is not zero. Before downloading a file, the μTorrent program preserves the space required by the target file by writing dummy data sequentially to minimize file fragmentation. Thus, when downloading the file, an average space of 3.4GB already has valid data, making OFS use only 600MB of space as the free space to service write requests. On the other hand, the multi-threaded write benchmark can utilize 1.2GB of free space on average since it invalidates 800MB space in the deleting phase before starting the writing phase. Due to the small free space for absorbing write requests, OFS incurs garbage collection more frequently in the torrent benchmark. Nevertheless, EXT4 also has the same problem, and OFS has a very low number of valid copies in this benchmark.

D. Hot/Cold Data Separation

We create a hot/cold benchmark to analyze the effectiveness of the hot/cold data management in our OSSD prototype. With the total 8,000 files with 128KB file size at the beginning, the benchmark selects 10% of files with the probability of 90% and rewrites the whole file on each transaction. The remaining 90% of files has the probability of 10% of being rewritten. We obtain the results after the amount of write becomes larger than 8GB to generate a sufficient number of garbage collections.

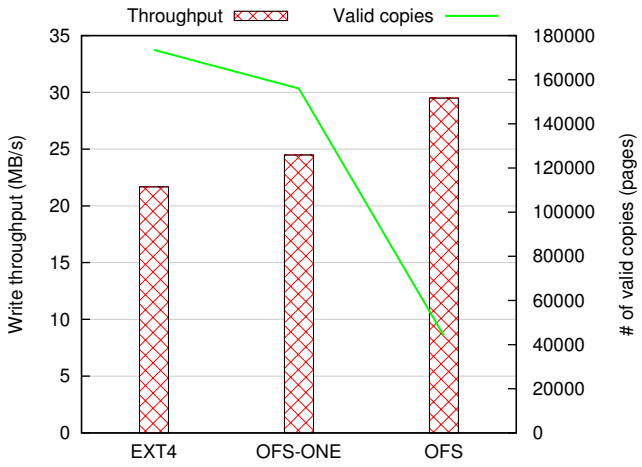


Fig. 6. Results of the hot/cold benchmark.

Figure 6 illustrates the results of the hot/cold benchmark. The bar graphs associated with the left axis show the throughput results while the line graph with the right axis depicts the number of valid page copies during garbage collection. OFS also outperforms OFS-ONE and EXT4 while the results have the similar tendency with those of Figure 5.

However, the performance gain is small in comparison with Figure 5. This problem is related to the efficiency in the multi-channel architecture of RawSSD. When only one update block is used in OFS-ONE and EXT4, all the incoming write requests are placed in the update block sequentially, maximizing the write throughput by interleaving them over multiple channels in RawSSD. However, OFS redirects the incoming writes to one of several update blocks, which degrades the interleaving efficiency. Especially, small writes for directory data disturb utilizing multiple channels. In spite of this problem, OFS improves the write throughput since the profit of hot/cold data separation is greater than the slight loss of multi-channel efficiency.

E. Handling Prioritized Requests

This section examines how well the prioritized requests are handled in the presence of background garbage collection. For the scenario of processing prioritized read requests, we play music video of 200 seconds long with 30 frames per second. We also run the multi-threaded write benchmark described in Section VI-C in the background to generate garbage collections. The object corresponding to the music video file has the higher priority than any other objects used in the background job.

Figure 7 shows the variation in the actual frames per second (FPS) while playing the music video along with the background job. EXT4 (prioritized) denotes the results of the EXT4 configuration where the video player informs the Linux kernel of its higher priority through the `ioprio_set()` system call. For OFS (prioritized), the requests to the music video file (object) are given the higher priority. There is no support for handling prioritized requests in the case of EXT4 (normal) and OFS (normal). Figure 8 illustrates the cumulative distribution of read latencies for the music video file.

OFS plays the music video stably at 30 FPS which is

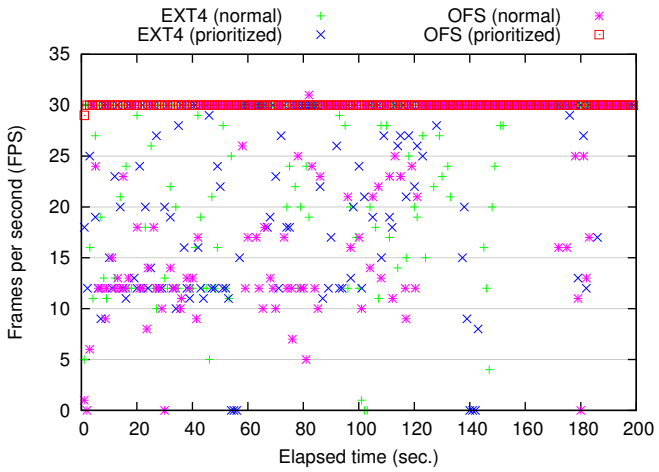


Fig. 7. Results of FPS while playing music video.

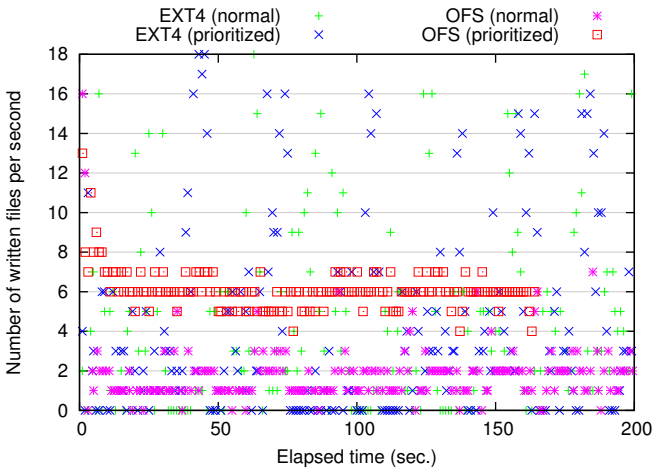


Fig. 9. Results of written files per second.

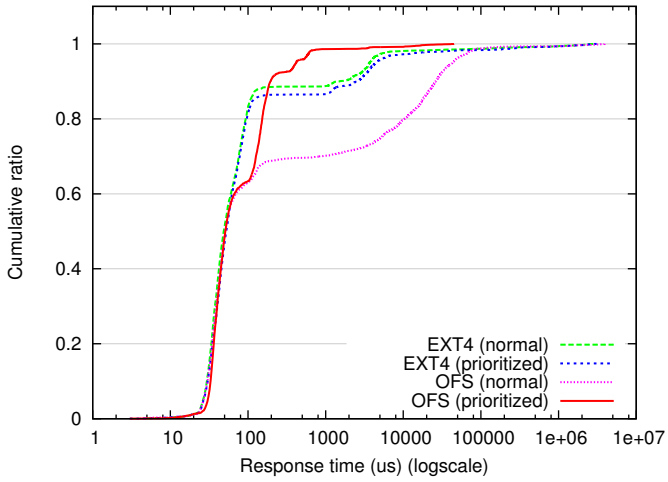


Fig. 8. Cumulative distribution of read latencies.

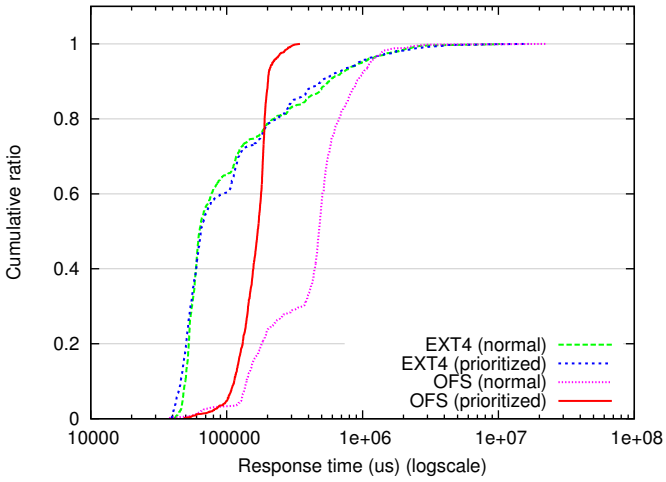


Fig. 10. Cumulative distribution of write latencies.

the same value as the music video file. However, the FPS results of other configurations are fluctuated severely since they cannot service prioritized read requests while running garbage collections. Actually, we can see frequently that the music video is paused for a moment while playing.

Figure 8 also depicts that other configurations suffer from the delay caused by garbage collections. For OFS, read requests from the player have just small delay to wait for one valid copy or one erase operation. However, in other configurations, they have to wait until garbage collection makes one free block by performing many valid page copies followed by an erase operation. Therefore, they have long latencies if garbage collection is already in progress while processing the incoming read requests.

For the scenario of prioritized write requests, we write 2MB files 1,000 times with the higher priority while running the same background job with the scenario of prioritized read requests. Note that the FML in OFS is aware of the request priority as the information is delivered through the object-based interface. However, this information is not carried by the block-level interface, making sFTL unable to differentiate prioritized requests despite of the `ioprio_set()` system call in EXT4.

Figure 9 depicts the number of written files per second in the prioritized write requests scenario. Figure 10 also illustrates the cumulative distribution of write latencies for writing one 2MB file.

Again, OFS exhibits stable results with about 6 written files per second. Although EXT4 shows higher values of written files per second if the request does not collide with garbage collection, many results scored very low values since they have to wait for the completion of on-going garbage collection. Actually, in OFS (prioritized), the benchmark is finished in 164 seconds with stable latency results while the benchmark is finished in 230 seconds in EXT4 (prioritized) due to the waiting time for garbage collection. Figure 10 also indicates the same consequence. In comparison with EXT4, OFS has consistent latencies.

The consistent latency while processing prioritized write requests is important in many systems. In case of a video recorder such as digital camcorder, the recorder can write a video stream steadily only if the underlying storage device services write requests with stable performance. Moreover, many recent smartphones and digital cameras are also providing the continuous shot functionality. With the support for prioritized requests, this operation can save images stably while taking pictures even if the other processes generate write operations

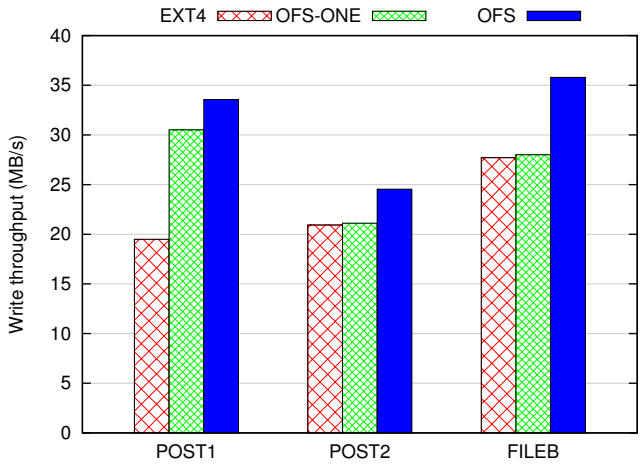


Fig. 11. Write throughput results of file system benchmarks.

and incur garbage collections.

From these results, we can see that it is indeed possible to provide differentiated storage services in OSSDs with very little effort.

F. File System Benchmarks

We run PostMark [18] and Filebench [7] benchmarks to measure the overall file system performance after aging the disk layout. In the first configuration of PostMark (POST1), the benchmark creates 2,000 files with the random size from 16KB to 256KB, and run 20,000 transactions with 16KB I/O size. During each transaction, the POST1 benchmark performs one of four operations: creating a new file, deleting a file, reading the entire contents of a file, and appending a file with a random amount of data up to 16KB. In the second configuration (POST2), we increase the file size to 256KB-512KB to evaluate the performance with large file size. We set the create bias to -1 to reduce small write requests originated from accessing directory files.

We also run the Filebench benchmark to measure the performance of the multi-threaded writes with massive I/O operations (FILEB). We use the modified version of the videoserver workload in the Filebench base workloads to run the benchmark in our device setting. The workload creates 64 files with 32MB file size. Then, four threads delete a randomly chosen file and create another file which has the same file size as the previously deleted file. Again, we call `fsync()` immediately after a file is deleted to deliver the delete operation to the target device.

Before we run these benchmarks, the device is aged by running the hot/cold benchmark shown in Section VI-D. To generate garbage collections during the benchmark run, the total 85% of the disk space (3.4GB) has been filled.

Figure 11 compares the write throughput results of each file system benchmark configuration. OFS outperforms OFS-ONE for each benchmark by 9.9%, 16.2%, and 27.9%, respectively. This is because it has the smallest number of valid copies by separating update blocks according to object properties. By the object-aware data placement policy, OFS can place the data from the same object in the same update block. If the object is deleted afterwards, OFS can reduce the fragmentation of free pages. In addition, hot/cold data separation while aging the device is also helpful to reduce the garbage collection overhead.

In case of the POST1 benchmark, EXT4 has very low performance than OFS-ONE. Since there are many write operations on small-sized directory files in the POST1 benchmark, EXT4 requires many read-modify-write operations due to external fragmentation. When the incoming write request updates the file which has smaller size than one flash page size, EXT4 needs the read-modify-write operation for the entire flash page because sFTL does not know the file size. However, since OFS-ONE knows the exact object size, OFS-ONE does not have to perform the read-modify-write operation if the write request updates all data of the object in the single flash page.

For OFS and OFS-ONE, the results of the POST2 benchmark are smaller than the POST1 benchmark. This is because the POST2 benchmark generates just the write requests of up to 16KB in appending transactions. On the other hand, the POST1 benchmark results in better throughput as it produces large requests when creating files.

VII. Conclusion

This paper presents the design and prototype implementation of the object-based SSDs (OSSDs). We have proposed the general system architecture and the storage software stack needed to support OSSDs, clarifying the role and responsibility of each layer. We confirmed that the object-based interface offers great potential for SSDs using the various evaluations with our OSSD prototype constructed on an iSCSI-based embedded storage device.

There are still many issues left to be explored. As future work, a more comprehensive analysis on various object properties is required. Second, since both EXOFS and OFS do not have any mechanism for ensuring metadata reliability, we are going to add the journaling support in our OFS. Finally, our object-aware I/O scheduler, OAQ, also requires improvement as it has overhead while processing I/O requests.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (No. 2010-0026511). This work was also supported by the IT R&D program of MKE/KEIT (No. 10041244, SmartTV 2.0 Software Platform).

References

[1] N. Agrawal, V. Prabhakaran, and T. Wobber, "Design tradeoffs for SSD performance," in *Proc. USENIX ATC*, June 2008, pp. 57–70.

[2] A. Aizman and D. Yusupov, "Open-iscsi," http://www.open-iscsi.org/, 2005.

[3] Aleph One Limited, "Yet another flash file systems (YAFFS)," http://www.yaffs.net.

[4] BitTorrent, Inc., "μTorrent," http://www.utorrent.com/, 2013.

[5] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, November 1999.

[6] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software – Practice & Experience*, vol. 29, no. 3, pp. 267–290, March 1999.

[7] File system and Storage Lab at Stony Brook University, "Filebench," http://filebench.sourceforge.net/, 2011.

[8] T. Fujita and M. Christie, "Linux SCSI target framework (tgt)," http://stgt.sourceforge.net/, 2011.

[9] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. ASPLOS*, February 2009, pp. 229–240.

[10] B. Harrosh, "open-osd," http://www.open-osd.org.

[11] B. Harrosh and B. Halevy, "The Linux Exofs object-based pNFS metadata server," 2009.

[12] A. Hunter, "A brief introduction to the design of UBIFS," http://www.linux-mtd.infradead.org/doc/ubifs.html, 1995.

[13] INCITS T10 Committee, "SCSI object-based storage device commands (OSD)."

[14] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A file system for virtualized flash storage," in *Proc. FAST*, February 2010.

[15] D. Jung, J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme," *ACM Transactions on Embedded Computer Systems*, vol. 9, no. 4, pp. 40:1–40:41, March 2010.

[16] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "μ-tree: An ordered index structure for NAND flash memory," in *Proc. EMSOFT*, September 2007, pp. 144–153.

[17] Y. Kang, J. Yang, and E. L. Miller, "Object-based SCM: An efficient interface for storage class memories," in *Proc. MSST*, May 2011, pp. 1–12.

[18] J. Katcher, "PostMark: A new file system benchmark," Network Appliance, Inc, Tech. Rep. TR3022, 1997.

[19] J.-H. Kim, D. Jung, J.-S. Kim, and J. Huh, "A methodology for extracting performance parameters in solid state disks (SSDs)," in *Proc. MASCOTS*, September 2009, pp. 1–10.

[20] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drives," in *Proc. EMSOFT*, October 2009, pp. 295–304.

[21] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, May 2002.

[22] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computer Systems*, vol. 6, no. 3, July 2007.

[23] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-FTL: A memory-efficient flash translation layer supporting multiple mapping granularities," in *Proc. EMSOFT*, October 2008, pp. 21–30.

[24] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated storage services," in *Proc. SOSP*, October 2011, pp. 57–70.

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

[26] D. Nagle, M. E. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran, "The ANSI T10 object-based storage standard and current implementations," *IBM Journal of Research and Development*, vol. 52, no. 4/5, pp. 401–411, November 2008.

[27] NVMHCI Work Group, "NVM Express 1.0," 2011.

[28] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. OSDI*, December 2008, pp. 147–160.

[29] W. Prouty, "NAND flash reliability and performance – the software effect," Flash Memory Summit, August 2007.

[30] A. Rajimwale, V. Prabhakaran, and J. D. Davis, "Block management in solid-state devices," in *Proc. USENIX ATC*, June 2009.

[31] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *Proc. SOSP*, October 1991, pp. 1–15.

[32] Samsung Electronics Co., "K9XXG08UXM flash memory data sheet," 2007.

[33] M. Saxena and M. M. Swift, "FlashVM: virtual memory management on flash," in *Proc. USENIX ATC*, June 2010.

[34] Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min, "Hydra: A block-mapped parallel flash memory solid-state disk architecture," *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 905–921, July 2010.

[35] F. Shu, "Windows 7 enhancements for solid-state drives," Microsoft WinHEC, 2008.

[36] F. Shu and N. Obr, "Data set management commands proposal for ATA8-ACS2," INCITS T13/e07153r6 (Revision 6), December 2007.

[37] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Life or death at block-level," in *Proc. OSDI*, December 2004.

[38] Sun Microsystems, Inc., "Lustre file system, white paper," 2007.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long, "Ceph: A scalable, high-performance distributed file system," in *Proc. OSDI*, November 2006, pp. 307–320.

[40] D. Woodhouse, "JFFS: The journalling flash file system," in *Proc. of Ottawa Linux Symposium*, July 2001.

[41] P.-L. Wu, Y.-H. Chang, and T.-W. Kuo, "A file-system-aware FTL design for flash-memory storage systems," in *Proc. DATE*, April 2009, pp. 393–398.