Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems

Youyou Lu Jiwu Shu* Weimin Zheng
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
**Corresponding author: shujw@tsinghua.edu.cn*
luyy09@mails.tsinghua.edu.cn, zwm-dcs@tsinghua.edu.cn

Abstract

Flash memory has gained in popularity as storage devices for both enterprise and embedded systems because of its high performance, low energy and reduced cost. The endurance problem of flash memory, however, is still a challenge and is getting worse as storage density increases with the adoption of multi-level cells (MLC). Prior work has addressed wear leveling and data reduction, but there is significantly less work on using the file system to improve flash lifetimes. Some common mechanisms in traditional file systems, such as journaling, metadata synchronization, and page-aligned update, can induce extra write operations and aggravate the wear of flash memory. This problem is called *write amplification from file systems*.

In order to mitigate write amplification, we propose an object-based flash translation layer design (OFTL), in which mechanisms are co-designed with flash memory. By leveraging page metadata, OFTL enables lazy persistence of index metadata and eliminates journals while keeping consistency. Coarse-grained block state maintenance reduces persistent free space management overhead. With byte-unit access interfaces, OFTL is able to compact and co-locate the small updates with metadata to further reduce updates. Experiments show that an OFTL-based system, OFSS, offers a write amplification reduction of 47.4% ~ 89.4% in SYNC mode and 19.8% ~ 64.0% in ASYNC mode compared with ext3, ext2, and btrfs on an up-to-date page-level FTL.

1 Introduction

In recent years, flash memory technology has greatly improved. As mainstream design moves from single level cell (SLC) to multi-/triple-level cells (MLC/TLC), flash-based storage is witnessing an increase of capacity and a reduction of per-bit cost, which results in a sharp growth of adoption in both enterprise and embedded storage systems. However, the increased density of flash memory requires finer voltage steps inside each cell, which is less tolerant to leakage and noise interference. As a result, the reliability and lifetime of flash memory have declined dramatically, producing the endurance problem [18, 12, 17].

Wear leveling and data reduction are two common methods to extend the lifetime of the flash storage. With wear leveling, program/erase (P/E) operations tend to be distributed across the flash blocks to make them wear out evenly [11, 14]. Data reduction is used in both the flash translation layers (FTLs) and the file systems. The FTLs introduce deduplication and compression techniques to avoid updates of redundant data [15, 33, 34]. File systems try to reduce updates with tail packing [7] or data compression [2, 13]. However, these data reduction techniques are inefficient in reducing write amplification from file systems, because the metadata updates generated from file systems can hardly be reduced for the following two reasons. One is that data reduction should not compromise the file system mechanisms. E.g., duplicated data in the journals should not be removed in data reduction. The other is that most metadata is frequently synchronized for consistency reasons, preventing data reduction.

Unfortunately, some common mechanisms in legacy file systems exacerbate flash write amplification. First, journaling, which is commonly used to keep the updates atomic, doubles the write size as the data and metadata are duplicated to the journal logs. Second, metadata is frequently written synchronously to avoid data loss in the presence of failure. Even though metadata consumes little storage space, frequent writes generate tremendous write traffic, which has a huge impact on the memory wear. Third, writes to the storage device are page-aligned even for updates of several bytes. Actual access sizes from the applications are hidden from the block device by the system, and thus update intensity is amplified by the page access granularity. Partial

page updates nearly always require a read-modify-write. Even worse, the trend of increasing page size of the flash memory makes this problem more serious [17]. Last but not least, the layered design principle makes the file system and the device opaque to each other because of the narrow block interfaces [35, 29, 28]. The file system mechanisms over flash storage fail to address the endurance problem, and consequently, flash memory wears out more quickly with these mechanisms. Transparency also prevents mechanisms in file systems from exploiting the characteristics of the storage device.

In fact, flash memory provides opportunities for better system designs. First, each page has a page metadata area, also the OOB (out-of-band) area. The reserved space can be used to keep an inverse index for the lazy persistence of index metadata, or transaction information to provide write atomicity. Second, all of a flash block become clean after the block is erased. The metadata overhead can be reduced by tracking free space in units of erase blocks (256 KB) rather than file system blocks (4 KB). Third, random read performance is improved dramatically compared to HDDs (Hard Disk Drives). Partial page updates can be compacted without introducing significant random read penalty. As discussed above, flash memory characteristics can be exploited to mitigate the endurance problem in cooperation with the system.

We propose an object-based flash translation layer design named OFTL, which offloads storage management to the object storage and co-designs the system with flash memory to reduce write amplification. Our contributions are summarized as follows:

- We propose OFTL, an object-based flash translation layer, to facilitate semantic-aware data organization.
- To reduce the cost of the index metadata persistence and journaling, we flush the indices lazily and eliminate journaling by keeping the inverse index and transaction information in the page metadata.
- We also employ coarse grained block state maintenance to track the page statuses to lower the cost of free space management.
- With byte-unit interfaces in OFTL, we compact partial page updates and co-locate them with metadata to reduce the number of page updates.
- We implement an OFTL-based system and evaluate it under different kinds of workloads. The results show a tremendous reduction in write amplification compared with existing file systems.

The rest of the paper is organized as follows. Section 2 gives the background of flash memory and flash-based storage system architectures. We then present our OFTL design in section 3 and the co-design of the system mechanisms with flash memory in section 4. We describe the implementation in section 5 and evaluate the design in section 6. Related work is presented in section 7, and the conclusions are given in section 8.

2 Background

2.1 Flash Memory Basics

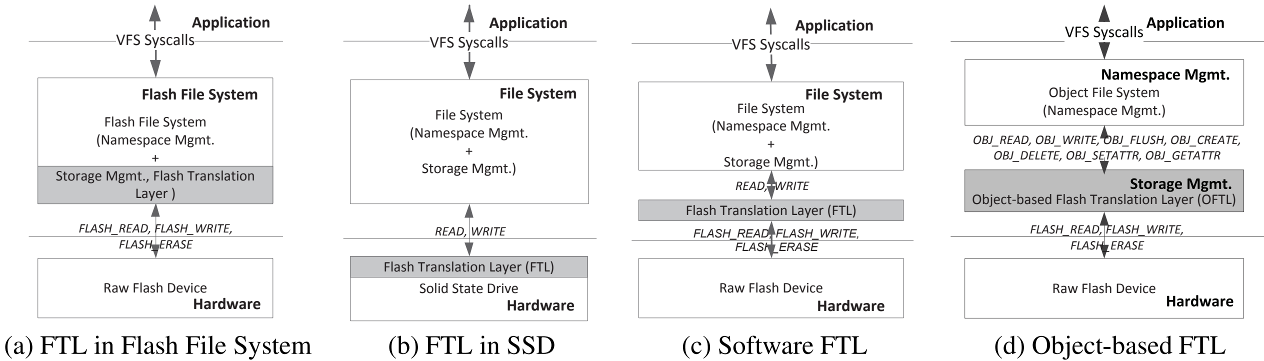
Flash memory is read or written in page units, and erased in flash block units. The page size ranges from 512B to 16KB [11, 17]. Each flash page has page metadata, which is atomically accessed along with the page data. Typically, a 4KB page has 128B page metadata [11]. A flash block is composed of flash pages, e.g., a 256KB flash block contains 64 pages [11]. Flash blocks are further arranged into planes and channels for parallelism inside a flash drive, known as *internal parallelism*.

Compared with hard disk drives, flash memory has two unique characteristics, a no-overwrite property and an endurance limitation, which are supposed to be taken into consideration by the file system designs. The *no-overwrite* property means that a programmed page cannot be reprogrammed until the flash block it resides in is erased. Updates to the programmed pages are redirected to clean pages in a no-overwrite manner, while the programmed pages are invalidated for later erasure. The *endurance limitation* is that each memory cell has a limited number of P/E operations. The memory cell wears out with more P/E cycles, causing the deterioration of both the lifetime and reliability.

2.2 Architectures of Flash-based Storage Systems

Flash memory came into wide use in embedded systems. In embedded systems, flash memory does not support a block interface and is directly managed by the file system, where the mapping, garbage collection and wear leveling are implemented. The file systems are specialized for flash memory, which are called flash file systems [10, 32], as shown in Figure 1(a). With increased capacity and reduced cost, flash devices are adopted in computer systems from laptops to enterprise servers as the substitution of HDDs. The FTL is implemented in device firmware to provide block interfaces, as illustrated in Figure 1(b).

As embedded FTL requires incredible computing power from embedded processors and large DRAMs for increasing device capacity, FusionIO implements FTL in software, called VSL (Virtual Storage Layer), sharing the host CPU cycles and the main memory capacity [4]. Figure 1(c) shows the architecture. The software-based VSL provides opportunities of optimizations for file systems. DFS is proposed on VSL to leverage the storage management in VSL [20], and an atomic-write



(a) FTL in Flash File System (b) FTL in SSD (c) Software FTL (d) Object-based FTL

Figure 1: Architectures of Flash-based Storage Systems

interface is exported from VSL using the log-structured FTL to provide the system with write atomicity [28].

Although software-based FTLs have shown better performance than embedded FTLs, the narrow block interfaces between the file system and FTL prevent optimizations from either the file system or the FTL. File semantics are hidden behind the narrow interfaces, preventing intelligent storage management [35, 26]. Also, flash memory properties are opaque to the file system, leading to missed opportunities in file system optimizations [20, 29, 28]. Similar to Object-based SCM [21], we propose object-based FTL design, OFTL, for better cooperation with the file system and flash memory, as shown in Figure 1(d). In OFTL-based architecture, storage management is moved from the file system to OFTL to directly manage the flash memory, so that flash properties can be investigated to optimize the file system mechanism design, such as journal removal and frequency reduction of metadata synchronization. OFTL manages the flash memory with read/write/erase operations, and directly accesses the page metadata of each flash page. What is more, OFTL exports byte-unit access interfaces to the file system, which is an object-based file system free from storage management and only manages the namespace.

3 Object-based Flash Translation Layer

The OFTL-based architecture offloads storage space management from the file system to the OFTL for better co-designs of the file system and the FTL. OFTL accesses the raw flash device in page-unit interfaces, while exporting byte-unit access interfaces to the file system. And thus, OFTL translates the mapping from the logical offset of each object to the flash page address. In this section, we will describe the OFTL interfaces and the data organization.

OFTL Interfaces. OFTL exports byte-unit read/write interfaces to the file system in order to directly pass the access size in bytes to the OFTL. Table 1 shows the object interfaces. Both *oread* and *owrite* interfaces pass the *offset* and *len* in byte units instead of sector units to the OFTL. Thus, OFTL gets the exact access size from the applications, making it possible to compact small updates into fewer pages, which is discussed in section 4.3. In addition, operations are made on each object with the *oid* given in the object interface, which makes OFTL aware of the data type of accessed pages. OFTL leverages the object semantics to cluster the update correlated data. Also, OFTL differentiates index pages from data pages with the type semantics to keep ancillary metadata in the page metadata for lazy indexing, which is discussed in section 4.1.

Table 1: Object Interface

Operations	Description
oread(devid, oid, offset, len, buf)	read data to *buf* from *offset* of object *oid*
owrite(devid, oid, offset, len, buf)	write data *buf* to *offset* of object *oid*
oflush(devid, oid)	flush the data and metadata of object *oid* stable
ocreate(devid, oid)	create object *oid*
odelete(devid, oid)	delete object *oid*
ogetattr(devid, oid, buf)	get attributes of object *oid*
osetattr(devid, oid, buf)	set attributes of object *oid*

OFTL accesses the flash memory with page-unit read/write operations and block-unit erase operations. The page metadata read/write(s) are directly accessed from the OFTL following the NVMe specification [6], which defines the host controller interface for accessing the non-volatile memory on a PCI Express bus.

Data Organization. The OFTL is comprised of two main parts, the Object Storage and the Block Information Metadata. A single Root Page identifies the location of

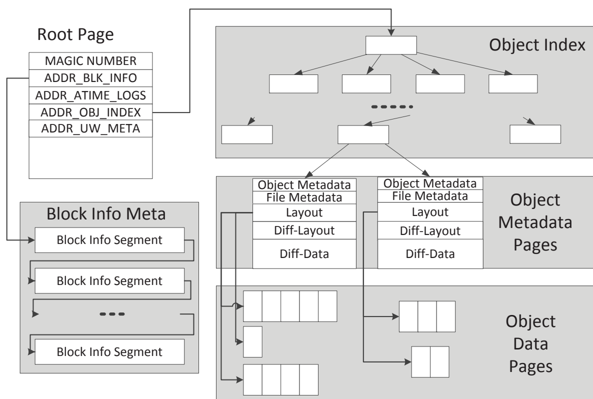


Figure 2: OFTL Layout

each of the Object Storage and the Block Information Metadata, as shown in Figure 2. The Object Storage is organized in three levels: the Object Index, the Object Metadata Pages, and the Object Data Pages. The Object Index maps an object ID to its Object Metadata using a B+ tree. The Object Metadata contains information traditionally stored in a file system inode including allocation information, which references addresses in the Object Data Pages.

The Block Information Metadata keeps the metadata information of each flash block, including the flash block states, the number of invalid pages (whose data is obsolete but has not been erased), and the erase count of each flash block. Each flash block has three states: FREE, UPDATING, and USED, and each page has three states: FREE, VALID, and INVALID, which are explained in section 4.2. The Block Information Metadata is written in a log-structured way. Each block information entry has 32 bits, of which 20 bits are used for erase count, 10 bits for invalid pages count, and 2 bits for flash block states. Instead of being stored with each flash block, the block information is stored in a separated space in the flash memory, which results in fewer page updates during garbage collection.

4 System Co-design with Flash Memory

The OTFL uses three techniques to leverage the characteristics of the underlying flash device. In section 4.1, we introduce *Backpointer-Assisted Lazy Indexing*, a mechanism to efficiently maintain consistency between data and metadata. In section 4.2, we present our approach to *Coarse-Grained Block State Management*, which reduces the frequency of status writes. In section 4.3, we present our *Compacted Update* technique which amortizes the cost of page writes across multiple unaligned page writes.

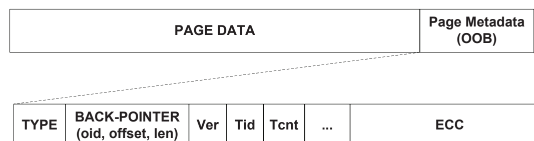


Figure 3: Page Metadata

4.1 Backpointer-Assisted Lazy Indexing

The index metadata, either the pointers for the data layout in the object metadata pages to point to the data pages, or the pointers for the object index to point to the object metadata page, should be synchronously written to the storage device in case of data loss or inconsistency. The synchronization of index metadata is also called index persistence. While a typical pointer has the size of 8 bytes, the index persistence updates a whole page, which is 4KB or even larger. Thus, the frequent index persistence causes serious write amplification.

The lazy indexing technique, in which the type specific backpointer is employed in the page metadata of each indexed page for lazy persistence of the index metadata, is developed to reduce the frequency of index persistence while maintaining consistency. As shown in Figure 2, the object index, the object metadata pages and the object data pages are indexed in tree structures and form a three-level hierarchy. Figure 3 illustrates the page metadata organization. In OFTL, we have two types of backpointers $(oid, offset, len)$. One is used in the data pages to inversely index the object metadata, and the oid, $offset$, len represent the object id, the logical page offset in the object, and the valid data length of the page, respectively. The other is used in the object metadata page to reversely index the object index, and only the oid is set to represent the object id. When the backpointer is written, the type is first set to indicate which type the inverse index is, so that the backpointer can be understood correctly for different types. We also keep the version in the *ver* to identify the latest page version on recovery. Therefore, the type specific backpointer serves as the inverse index, and the index persistence is decoupled from the page update.

To reduce the scan time of inverse index to reconstruct the index metadata after system failures, we use the *updating window* to track the recently allocated flash blocks that have not completed the index metadata persistence. The updating window is maintained by the *checkpoint* process, which is triggered when the number of free pages in the updating window drops below a threshold and a window extending is needed. The updating window describes the set of blocks whose inverse indices need to be checked after a failure, because they may not be referenced by the index. Flash blocks

are preallocated and the addresses of the preallocated blocks are written to a block called the updating window metadata. This block is written persistently to the flash. Subsequently, blocks are allocated from the updating window and referenced via in-memory indices, but the indices are not written persistently. We call this lazy persistence. Periodically the checkpoint process removes from the updating window metadata the addresses of those blocks whose corresponding indices have been written persistently and when necessary, preallocates a new collection of blocks and adds their addresses the the updating window. After a crash, the recovery process need only read the blocks whose addresses are in the updating window metadata and check that they are referenced by the index.

The updating window also provides update atomicity of multiple pages. The page metadata keeps transaction information $(tid, tcnt)$, in which tid is a unique ID in the updating window for each write operation and $tcnt$ is the count of pages in the update. For all pages of a write operation, only one page has the $tcnt$ set and the others have the $tcnt$ value of zero. The $tcnt$ is used to check the completeness of the operation after system failures. Garbage collection is not allowed to be performed on the flash blocks in the updating window, so that transaction information is complete for the atomicity check. Thus, journals can be eliminated with write atomicity provided from the flash memory.

When rebooting after system failures, object data pages in the updating window are scanned. The transaction information is first checked to determine the completeness of the write operation in which the page is involved. If a write is incomplete, all the pages of the write are discarded. In this way, the atomicity is guaranteed. After the check, the backpointers are read out to update the object layout if object layout has not been written stable before system fails. As all the updates are located in the updating window since last checkpoint, the object layout can be updated to the latest with the reconstruction of backpointers in the scanned pages from current updating window. The file size metadata is also updated with the recalculation of all the valid data sizes. While other descriptive metadata, such as modified time and access control list, may get lost after unexpected crashes, the system consistency is not hurt. Similarly, the object index can be updated with the scan of current updating window of object metadata pages. With the ancillary information in the page metadata and the updating window, the index persistence frequency is reduced and the write atomicity is provided to eliminate the journals.

Consistency and Durability Discussion. In the lazy indexing technique, metadata except the index pointer and the size may get lost after system crashes. This results in the out-of-date versions of metadata, which hurts the durability of metadata updates. Compared with durability, we take the consistency as a more important thing, because the corruption of indices will cause the inconsistency and even failures of file systems. We use the lazy indexing technique to guarantee that the indices and the size can be reconstructed during recovery. But for durability issues, we opt to let the file system or the application determine the time to make other metadata durable with explicit synchronization.

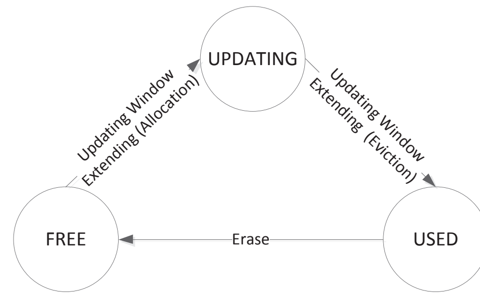


Figure 4: State Transition of Flash Blocks

4.2 Coarse-Grained Block State Maintenance

In flash memory, each page has three states: FREE, VALID, and INVALID. A *free* page is a page that has not been written. A *valid* page is a page that has been written and whose data is valid data. An *invalid* page is a page that has been written but whose data is obsolete. Valid and invalid pages are also called INUSE pages. Each flash block also has three states: FREE, UPDATING, and USED. Pages in *free* blocks are all free pages, and pages in *used* blocks are all inuse pages, while only the *updating* blocks have both free or inuse pages. Since the pages are written sequentially inside a flash block, the latest allocated page number is used to separate the free pages from the inuse for the updating block. Therefore, OFTL differentiates between free and inuse pages by tracking the flash block states. For the inuse pages, the index metadata is used to further differentiate the valid pages from the invalid. The valid pages are indexed in the index metadata while the invalid are not. As such, the free, valid, invalid pages can be identified by tracking the states of flash blocks, and free space management cost is reduced with the state maintenance in flash block units instead of page units.

OFTL further reduces the cost by bringing down the frequency of metadata persistence. The persistence of flash block states is only performed when the flash blocks are allocated to or evicted from the updating window as shown in the top of Figure 4. The state persistence is

relaxed but with the two conditions satisfied: (1) the set of persistent free blocks is a subset of actual free blocks set; and (2) the number of persistent invalid pages is no more than the actual number of invalid pages.

The first condition means a free block can be regarded as non-free, which may lead to missing allocation. But a non-free block cannot be taken as free, or else the write fails. It requires the state transition from FREE to UPDATING to be flushed immediately, but relaxes the state persistence from USED to FREE. The second condition relaxes the number persistence of invalid pages. The number of invalid pages is used to select the evicted flash block and check the number of valid pages that are moved during garbage collection. The valid pages are differentiated from the invalid by checking the index metadata. Until the number of valid pages is reached, all the remaining pages are invalid and no more invalid page checking is needed, so that the page moving of the evicted block stops. In the worst case, all the pages have to be checked if the persistent number of invalid pages is less than the actual. As system crashes seldom appear, the effect on garbage collection efficiency is limited. Similar to the number persistence of invalid pages, the recorded erase count may stay behind, which is also acceptable as the erase count is not sensitive.

In summary, free space management benefits from the coarse-grained block state maintenance because of the reduced metadata cost both from the flash block granularity state tracking and the reduced persistence of states.

4.3 Compacted Update

With byte-unit access interfaces, OFTL is able to identify partial page updates, which update only part of one page, both for the small updates less than one page and the heads/tails of large updates. The compacted update technique compacts the partial page updates of the same object and co-locates them with its object metadata page to reduce the update pages.

Partial Page Update. Partial page updates in OFTL are compacted and inserted into differential pages, a.k.a. diff-pages. Each data segment stored in the diff-page is called diff-data. Diff-data is indexed with diff-extent using the triple $< o_off, len, addr >$, where o_off and len represent the object offset and the length of the diff-data, respectively, and $addr$ is the offset of diff-data in the diff-page. Diff-extents are kept in the ascending order of the object offset in the diff-layout. Each object has one diff-layout, which is the collection of all the diff-extents of one object. Figure 5 shows the data structures.

While partial page updates are absorbed in diff-pages, which are indexed in the diff-layout, full page updates are directly written to object pages. Object pages are indexed

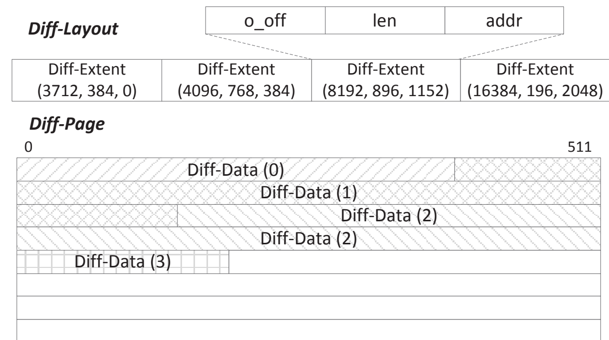


Figure 5: Differential Layout (Diff-Layout)

in the layout, which is the collection of all extents of one object that keep the start address and length pairs of full pages. As well as the diff-layout, the layout is recorded in the object metadata page, as shown in Figure 2. In the compacted update technique, read, write and merge operations are described below:

- Write: The write data is divided into pages. The parts that do not start or end page-aligned are identified as partial page updates, while the others as full page updates. Then the partial page updates are updated in the diff-pages. Because partial updates always supersede full pages, full page updates have to invalidate the corresponding diff-data and remove its diff-extents followed by updating the layout to refer to the full pages.
- Merge: A merge operation is needed when the diff-pages are full. When merging, the diff-extents are scanned to select the evicted logical page which consumes the most space in diff-pages. The object page of the evicted logical page is then read and merged with the diff-data. After that, the merged page is written to a new object page, and the corresponding diff-data and diff-extents are removed.
- Read: A read operation is first checked in diff-extents. If it is satisfied in the diff-data, the read buffer is filled with the diff-data. Otherwise, the object page is read and merged with any existing corresponding diff-data.

Update Co-location. In most cases, the metadata size of each object is far less than the page size. Instead of compacting metadata of multiple objects, OFTL co-locates diff-page with the object metadata page. Since each data update is followed by the metadata update such as the file size or the modify time, the co-location of the diff-data and metadata page usually saves one page write. In co-location, the size of diff-page is less than one page size and varies depending on the size change of object metadata. The size of diff-page is calculated by subtracting the metadata size from the flash page size,

which is used to check whether the diff-page is full. Once full, the merge operation is triggered to select some diff-data and merge them to the object data pages. In this way, the cost of partial page updates is amortized by compaction and co-location.

5 Implementation

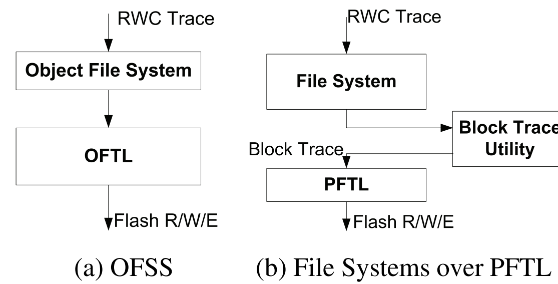
We implement OFTL as a kernel module in the Linux kernel 3.2.9. The module is composed of three layers: the translation layer, the cache layer, and the storage layer.

The translation layer follows the design shown in Figure 2. The in-flash Object Index is implemented using a Log-Structured Merge Tree (LSM-Tree) [27], which has an in-memory B+ tree for fast lookups and appends the record $< operation, object_ID, phy_addr >$ for each object index update. Each object has one in-memory object metadata data structure, which records the access statistics and two extent-based layouts, as well as the in-flash object metadata page shown in Figure 2. Both the diff-layout and the layout link their extents in the list structure in memory. On write requests, the write data is first split page-aligned. The full page updates are written to data pages followed by the layout update, while the partial page updates are inserted into diff-data followed by the diff-layout update. Object operations are transformed into read/write operations on the metadata or data pages and forwarded to the cache layer.

The metadata and data pages are cached in the cache layer. The OFTL cache follows the design of the page buffer in the linux kernel, except that the replacement is done with a whole object. On write operations, the cache checks the SYNC flag and calls the $flash_write$ interface from the storage layer if the SYNC flag is set. Otherwise, the object cache in the cache layer is updated. Page allocation is delayed until the pages need to be flushed. Then, the cache layer allocates free pages from the updating window and calls the $flash_write$ interface to write the pages via the storage layer.

The storage layer receives the flash read/write/erase operations from the cache layer, constructs the bio requests, and sends them to the generic block layer. We use the TRIM/DISCARD command in the system and SATA protocol to implement the erase operation. The DMA transmission of the OOBs follows the design in the NVMe standard[6]. Due to the hardware limitation, the OOB operations are currently simulated in memory.

In OFTL, we use a simple garbage collection and wear leveling strategy. The garbage collection starts when the percentage of free blocks drops below 15% and continues until the percentage exceeds the value. In wear leveling, an upper bound of erase count is used to prevent flash blocks with higher erase count from erases. The upper bound is raised periodically after the average erase count is increased. We will incorporate better strategies in the future.



(a) OFSS (b) File Systems over PFTL

Figure 6: Trace-driven Simulation Framework

For evaluation against other file systems solutions, we also implement a simple object file system to resolve the namespace as shown in Figure 1(d). The object file system uses an in-memory hash table to keep the mapping from the path to the object ID, and passes the IO requests to OFTL by substituting the object ID for the path. We call the OFTL-based system Object Flash Storage System (OFSS) in the evaluation section.

6 Evaluation

We measure the write amplification, the total size/count of writes to the flash memory divided by the total size/count of writes issued from the application, to evaluate the write efficiency of OFSS, as well as ext3, ext2, and btrfs on an up-to-date page-level FTL.

In this section, we first evaluate the overall performance of these four systems and then analyze the metadata amplification. We also measure the impact of flash page sizes and the overhead of extending the updating window brought by the OFTL design.

6.1 Experimental Setup

We extract the read, write and close operations (RWC Trace) from the system level IO traces and replay them on file systems and OFSS. Figure 6 shows the trace-driven simulation framework. In the replay program, the close operation incurs a *fsync* operation in the file systems or an *object_flush* operation in OFSS. In OFSS, we collect the count and size of I/Os to the flash memory in the storage layer of OFTL. In file system evaluations, we use *blktrace* utility [1] to capture the I/Os on the storage device in file system evaluations and then replay the block trace on PFTL, a simulated page-level FTL implemented as a kernel module in Linux kernel-3.2.9. We collect the count and size of I/Os to flash memory in PFTL. In the simulation, PFTL

Table 2: Characteristics of the Workloads

workload	write cnt	write size (KB)	flush cnt	% of un-aligned writes
iPhoto	496,542	6,651,962	37,054	51.2%
iPages	75,661	183,728	565	99.6%
LASR-1	32,249	42,600	1,714	93.3%
LASR-2	111,531	216,114	14,998	99.0%
LASR-3	21,956	24,426	3,056	96.1%
TPC-C	26,144	219,689	489	7.1%

transforms the logical page number to the physical page number using the two-level page table similar to the main memory management. LazyFLT [23] features are integrated into PFTL to reduce the mapping overhead.

We evaluate two workloads from iBench [19] in the desktop environment, three one-month traces of LASR [5] in the server environment, and one TPC-C trace from the database management system. The TPC-C trace is collected from the DBT2 workload [3] running on PostgreSQL using the *strace* utility [8]. The characteristics of the six workloads are shown in table 2.

The experiments are conducted on SUSE Linux 11.1 with Linux kernel 3.2.9 running on the computer with a 4-core Intel Xeon X5472 3GHz processor and 8GB memory. In addition to the disk drive used for the operating system, a Seagate 7200rpm ST31000524AS disk drive is used for trace collection. In the experiments, ext3 and ext2 are mounted with *noatime*, *nodiratime* options, and btrfs is mounted with *ssd*, *discard*, and *lzo* options. In OFSS settings, the default page size is 4KB and the flash block size is 256KB. Both updating windows for object data and metadata are set to the size of 64 flash blocks. Default OFTL cache size is 32MB.

6.2 Overall Comparison

In this section, both the write efficiency and the IO time are evaluated for OFSS against ext2, ext3, and btrfs built on the PFTL. To provide data consistency, ext3 is mounted with *data journal* option, and btrfs updates data using Copy-on-Write, while ext2 provides no consistency. OFSS provides data consistency by leveraging the no-overwrite property of flash memory and keeping the transaction information in page metadata. The four systems are evaluated in both SYNC and ASYNC mode. In the SYNC mode, both data and metadata are required to be flushed stable before the request returns. Ext2 and ext3 are mounted with *sync* option to support the SYNC mode. Btrfs uses *O_SYNC* flag in file open operations, as the *sync* mount option is not supported in btrfs. In the ASYNC mode, the data and metadata are buffered in memory until the explicit synchronization, time expiration, or cache eviction. Default mount options of the three file systems support the ASYNC mode.

Table 3 shows the write amplification measured with total write count and write size of each system in the SYNC mode. The average write amplification of the write count in ext3, ext2, btrfs, and OFSS is 4.60, 2.98, 6.85 and 1.11, respectively. Write amplification of the write size shows similar results, and the average is 17.47, 5.03, 24.99 and 2.64 in ext3, ext2, btrfs and OFSS, respectively. The differences mostly come from the metadata costs associated with the write operations. In ext2, a write operation has the sub-operations of a data update, an inode update, and sometimes the bitmap updates if space allocation is needed. In ext3 with data journal, the data and metadata are duplicated to the journal logs followed by a commit/abort synchronous write, and later are checkpointed to the actual locations by the journal daemon. Even though the write counts are close in ext3 and ext2 due to the coalesced writes while checkpointing, the write size in ext3 is still much larger than that in ext2 because of the duplicated data and metadata appended in the journal logs. In the SYNC mode, btrfs logs the updates to a special log-tree to eliminate the whole system update [30], resulting in a high write amplification. Btrfs is optimized for performance, which uses ”ssd” allocation scheme for seek free allocation, and has not been optimized for flash endurance [2]. With journals eliminated, the reduced metadata synchronization frequency, and the compacted update, OFSS offers a write amplification reduction of 47.4% ˜ 89.4% against the other three legacy file systems.

As shown in Table 3, write amplification in the ASYNC mode is 0.13, 0.09, 0.23 and 0.09 measured with the write count, and 2.73, 1.23, 1.93 and 0.98 with the write size, respectively for ext3, ext2, btrfs and OFSS. In the ASYNC mode, the write amplification is not as bad as that in the SYNC mode. The reason is that the metadata synchronization is infrequent and metadata updates are coalesced in the buffer as well as the data. However, the write intensity in ext3 doubles because of the journaling. Btrfs has to update the index metadata for the Copy-on-Write update mechanism, consuming more pages. Besides, the page-aligned update mechanism wastes space when the updates are page unaligned, which contributes the write amplification to all the three legacy file systems. Comparatively, OFSS performs better and offers a write amplification reduction of 19.8% ˜ 64.0% compared with the three legacy file systems.

Total IO time is also shown in Table 3 for performance comparison. We estimate the performance by issuing the requests to the SSD instead of the raw flash memory due to the hardware limitation. As the focus of our design is the write amplification reduction rather than

Table 3: Overall Evaluation on Write Amplification

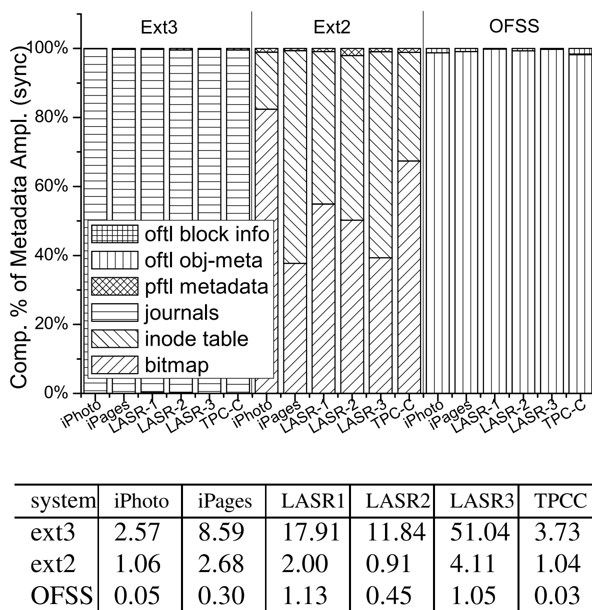
Workload	System	write-cnt-amplification		write-size-amplification		IO-time (unit: s)	
		sync	async	sync	async	sync	async
iPhoto	Ext3	2.7519	0.2024	3.5354	1.8725	487.744	257.356
	Ext2	5.2292	0.1206	2.4030	0.9163	325.571	126.185
	Btrfs	8.9320	0.2602	5.6071	1.0595	770.942	144.671
	OFSS	1.2304	0.1428	**1.1786**	**0.8916**	64.146	33.264
iPages	Ext3	2.3711	0.0267	9.8083	2.0491	37.174	7.764
	Ext2	2.7763	0.0182	5.3058	1.0137	20.538	3.833
	Btrfs	6.1918	0.0313	23.1998	1.0914	87.451	4.107
	OFSS	1.7143	0.0097	**3.5739**	**0.9758**	6.837	0.732
LASR-1	Ext3	3.7777	0.1127	19.2951	2.5593	16.959	2.245
	Ext2	1.8656	0.0617	6.3874	1.1445	5.633	1.044
	Btrfs	6.1207	0.1992	33.3570	2.0008	29.246	1.744
	OFSS	1.2081	0.0636	**4.0123**	**0.9779**	2.884	0.928
LASR-2	Ext3	3.7964	0.2147	13.5078	3.2976	60.167	14.784
	Ext2	1.7143	0.1892	4.2734	1.5034	19.202	6.761
	Btrfs	7.2232	0.5822	32.9221	3.0874	145.612	13.739
	OFSS	2.1302	0.1719	**4.4045**	**1.1021**	13.332	4.065
LASR-3	Ext3	11.5083	0.1602	53.7069	4.7793	27.083	2.413
	Ext2	2.4258	0.1184	9.4100	2.0676	4.768	1.044
	Btrfs	6.0922	0.2823	44.7353	3.4276	2.263	1.715
	OFSS	1.3935	0.1345	**5.2492**	**1.2647**	2.095	0.835
TPC-C	Ext3	3.3749	0.0666	4.9863	1.8068	22.532	8.196
	Ext2	3.8604	0.0321	2.3813	0.7050	10.800	3.201
	Btrfs	6.5125	0.0336	10.1339	0.8980	45.711	4.044
	OFSS	1.0696	0.0352	**1.0461**	**0.6822**	2.588	1.380

the performance oriented design of the mapping policy, garbage collection or wear leveling, the FTL inside the SSDs poses little impact on the performance evaluation. We collect and accumulate the device I/O time of each operation, which is measured from the issue of the operation to the SSD to the acknowledge from the SSD. As is shown, the OFSS performance in both modes significantly outperforms the others. The write size reduction not only extends the lifetime of the flash-based storage, but also improves the performance. Also, the performance improvement partially comes from the object-based FTL design, which uses delayed space allocation in OFTL cache for better I/O scheduling.

Workload Discussion. In general, OFSS brings larger benefits for workloads that have a large number of page unaligned updates or those that have frequent data synchronization. When a write operation is synchronized, the legacy file systems require a number of pages, including the pages for the data, inode, and bitmap file system blocks, to be updated. The metadata overhead is significant for small updates, in which only a few bytes are updated. The LASR-1 and LASR-3 workloads, which have the average update size less than 1KB, expose unacceptable write amplification in the SYNC mode, while OFSS reduces the metadata overhead tremendously and makes the amplification much lower. The unaligned page updates can also benefit more from OFSS because of the compacted update technique. A counter example is the TPC-C workload, which has its own buffers in the userspace and writes data in page units. The improvement of TPC-C workload from OFSS is not as significant as that of the other workloads.

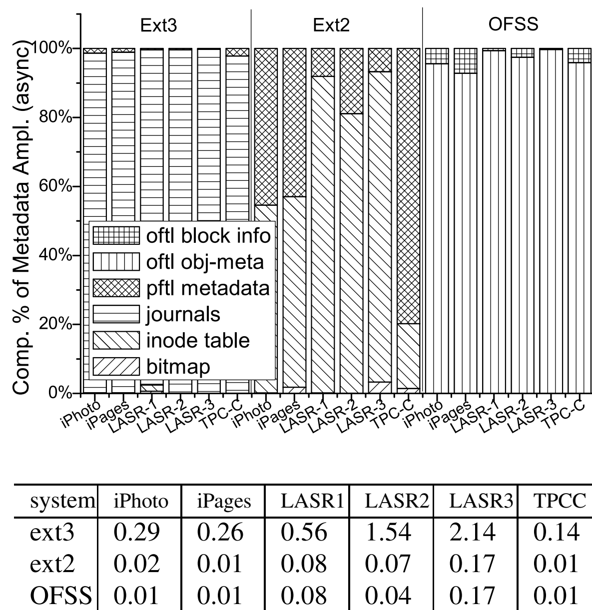
6.3 Write Amplification of Metadata

To further understand the improvement of OFSS and the effects of the journal removal and the metadata synchronization reduction, we take a closer look at the metadata amplification. In evaluation, we identify the file system block type of each access by checking the physical layout of ext2 and ext3 using the dumped information of the $dump2fs$ command. The journal type is identified with the ”$kjournald$” in the block trace. We omitted btrfs from this analysis, because we could not differentiate the metadata and data updates from the write addresses in the block trace. Figure 7 shows the total metadata amplification in the tables and the



system	iPhoto	iPages	LASR1	LASR2	LASR3	TPCC
ext3	2.57	8.59	17.91	11.84	51.04	3.73
ext2	1.06	2.68	2.00	0.91	4.11	1.04
OFSS	0.05	0.30	1.13	0.45	1.05	0.03

(a) Metadata Amplification (sync)



system	iPhoto	iPages	LASR1	LASR2	LASR3	TPCC
ext3	0.29	0.26	0.56	1.54	2.14	0.14
ext2	0.02	0.01	0.08	0.07	0.17	0.01
OFSS	0.01	0.01	0.08	0.04	0.17	0.01

(b) Metadata Amplification (async)

Figure 7: Write Amplification of Metadata

percentage of each kind of metadata in the figures above the tables for both SYNC and ASYNC modes.

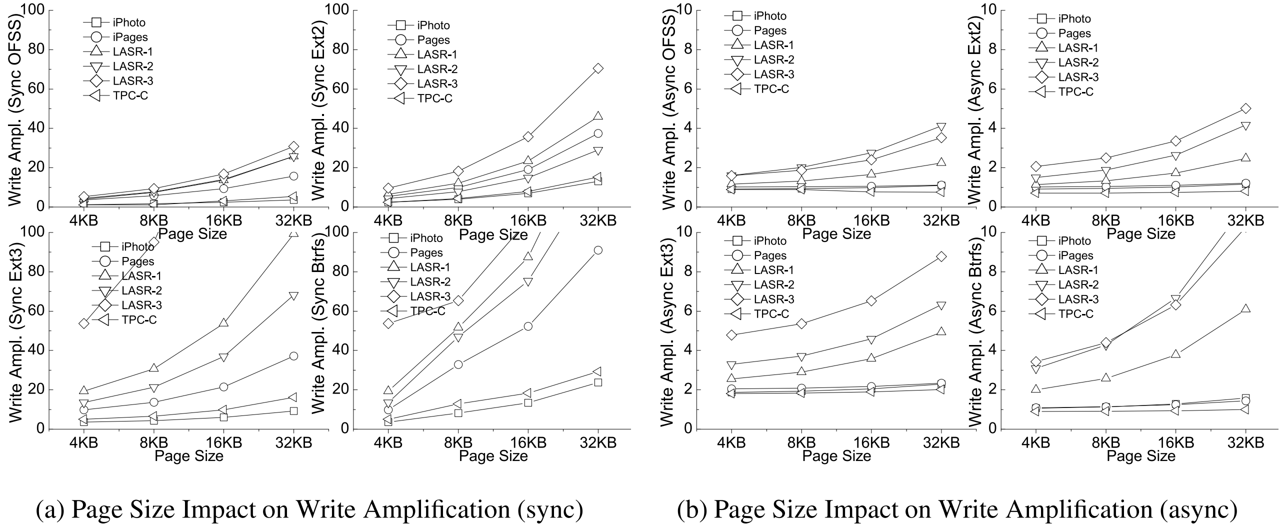
From Figure 7, there are two observations in the metadata amplification of ext3. One is that the write amplification of PFTL is negligible compared with the write amplification from the file system metadata. The other is that the journaling amplifies the writes dramatically and dominates the cost. The difference between ext2 and ext3 verifies the benefit of the journal removal. In the following, we are going to further understand the benefits of the metadata synchronization reduction by comparing OFSS with ext2.

In the metadata synchronization reduction evaluation, we remove the diff-data from the metadata pages in OFSS when calculating the metadata amplification of OFTL. As shown in Figure 7(a), the inode and bitmap updates consume the most part of the amplification in the SYNC mode. The write amplification of inode updates drops from 0.99 in ext2 on average to 0.50 in OFSS measured with OFTL metadata updates. The benefit comes not only from the lazy indexing but also from the amortized metadata page update cost with compacted update. The write amplification of file system block bitmap updates drops from 0.93 in ext2 on average to 0.0019 in the OFSS measured with OFTL flash block information updates, which shows the benefits of coarse-grained block state maintenance. As the free space is managed in flash block units, which is 64 pages (256KB) in the experiment and much larger than the 4KB block in ext2, the coarse-grained block state maintenance greatly benefits the free space management.

In the ASYNC mode shown as Figure 7(b), the inode and the metadata update cost is the dominated cost of ext2 and OFSS, respectively. Both of them have the write amplification cost around 0.05 on average. Legacy file systems compact multiple inodes into one inode table file system block, which is a different approach to reduce metadata amplification compared with the compacted update technique in OFSS. Both of the approaches have the similar effects in the ASYNC mode. The file system block bitmaps in ext2 are buffered and coalesced in the ASYNC mode, bringing the cost close to that of flash block state maintenance in OFSS.

6.4 Impact of Flash Page Size

The size of flash page has exposed an increasing trend in flash memory manufacturing [17]. We evaluate the write efficiency in ext2, ext3, btrfs and OFSS with different page sizes of 4KB, 8KB, 16KB and 32KB. As ext2 and ext3 support a maximum file system block size of 4KB, and btrfs is unstable when the lead/node/sector size is set to 8KB or higher, we align the 4KB block trace accesses to different flash page sizes in PFTL for evaluation. As shown in Figure 8, the write amplification increases dramatically with the page size increment in legacy file systems, while OFSS shows a significant improvement. Workloads with small access sizes, like



(a) Page Size Impact on Write Amplification (sync)

(b) Page Size Impact on Write Amplification (async)

Figure 8: Page Size Impact on Write Amplification

LASR-1 and LASR-3, deteriorate much more quickly. Another observation is that the write amplification in SYNC mode shown as Figure 8(a) is much worse than ASYNC shown as Figure 8(b) when the flash page size increases. This is because the access sizes in the SYNC mode are much smaller than the page size in most cases. Comparatively, the accesses in the ASYNC mode are buffered and coalesced into larger requests. The compacted update technique in OFSS further compacts partial page updates and co-locates them with the metadata to reduce the pages to be updated, leading to the higher page utilization. Thus, OFSS performs better with the write amplification in the SYNC mode of around 30, which is nearly 9% in ext3, and the write amplification in the ASYNC mode of approximate 3, nearly 25% in btrfs.

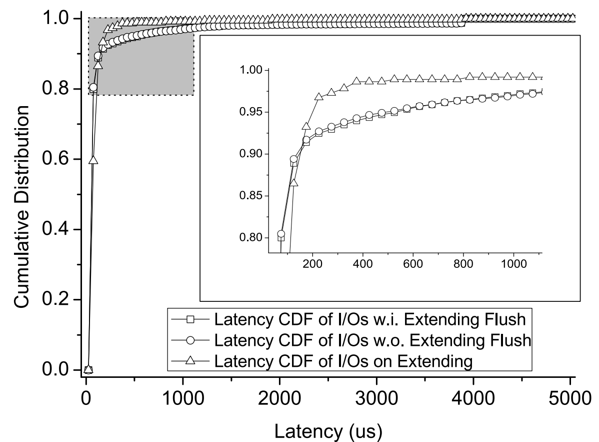


Figure 9: Overhead of Extending the Updating Window

6.5 Overhead of Extending the Updating Window

The updating window is employed to reduce the scan time after system crashes, but causes extra writes because of the index persistence while extending the updating window. To evaluate the overhead, we collect two data sets of the I/O latency of each operation, one for the normal extending and the other for the extending without index persistence. The cumulative distributions of the latencies in the two data sets are depicted with the nearly identical lines in Figure 9, which show the close performance. We also identify the I/Os during the extending period and collect the latencies of each I/O to show the impact on the latency of external I/Os. The latencies of I/Os during extending show little difference with the average of all I/Os. In the enlarged part of the figure, even though the cumulative distribution of I/Os on extending grows slower than the average when the latency is smaller than 200us, it grows much faster when the latency is around 200us. As a result, 99.6% of I/Os on extending have latencies smaller than 400us. Besides, the extending operations rarely appear, and only 0.04% of I/Os meet with the extending. Thus, we conclude that the updating window extending has limited impact on the performance.

7 Related Work

Flash file systems [32, 10, 9] have leveraged the no-overwrite property of flash memory for log-structured updates to optimize the performance by directly manag-

ing the flash memory. But they have not done much to the flash endurance other than wear leveling. Nameless Writes [35] and Direct File System [20] propose to remove the mappings of the file system to allow the FTL to manage the storage space in order to improve the performance. But file semantics fail to be passed to the FTL for intelligent storage space management. OFTL takes the concept of object-based storage [25] to export the object interfaces to file systems, which easily passes the file semantics to the device for intelligent data layout.

Recent research has also proposed eliminating the duplicated writes caused by the journaling in the flash storage. Write Atomic [28] exports an atomic write interface by leveraging the log-based structure of VSL [4] in FusionIO ioDrives. TxFlash [29] uses a cyclic commit protocol with the help of the page metadata to export the same interface. However, the cyclic property is complex to maintain, and the protocol requires a whole-drive scan after failures. Comparatively, OFTL takes a simple protocol similar to the transaction support in the log-structured file system [31], keeps the transaction information in the page metadata, and tracks the recently updated flash blocks in the updating window for fast recovery.

Backpointers have been used in disk storage systems to avoid file system metadata updates when moving file system blocks in Backlog [24], or to provide later consistency checking by embedding the backpointers in data blocks, directory entries, and inodes in Backpointer-based Consistency [16]. In flash-based storage, LazyFTL [23] has proposed to lazily update the mapping entries of a page-level FTL by keeping the logical page number (LPN) as the backpointer in page metadata and logging the pages in an update flash block area, so as to provide the performance of page-level FTLs at the cost of block-level FTLs. But it does not touch the metadata in the file system, which contributes much to the write amplification. Instead, OFTL leverages the page metadata to optimize the system design and uses the backpointer as the inverse index for lazy persistence of the index metadata.

Both CAFTL [15] and DeltaFTL [33] coalesce redundant data and compress the similar pages in the FTLs, while LFS [13], JFFS2 [32], UBIFS [9], and btrfs [2] compress the data pages in the file systems. All of them use compression by exploiting the page content similarity, which is orthogonal to the compacted update technique using access size information in OFTL. IPL (In-Page Logging) [22] takes an approach similar to OFTL and reserves a log region in each flash block to absorb the small updates. But it suffers from synchronous writes if data and metadata pages spread across multiple flash blocks. Tail packing in reiserfs [7] is most related to the compacted update technique in OFTL. Reiserfs packs the tails of each file in its inode. OFTL is different as OFTL updates in a no-overwrite way, so that OFTL compacts not only the tails but also the heads of each write operation, including both the append and update operation.

8 Conclusion

Legacy file systems focus on sequential access optimization instead of the write amplification reduction, which is critical for flash memory. System mechanisms, such as journaling, metadata synchronization and page-aligned updates, tremendously amplify the write intensity, while transparency brought by the indirection prevents the system from exploiting the flash memory characteristics. In this paper, we propose an object-based design name OFTL, in which the storage management is offloaded to the FTL from the file system for direct management over the flash memory. Page metadata is used to keep the inverse index for the lazy indexing and the transaction information to provide write atomicity for journal removal, with the help of the updating window to track the latest allocated flash blocks that have not been checkpointed. Also, free space management tracks the flash block states instead of the page states, and brings down the frequency of state persistence to further reduce the metadata cost. Using the byte-unit access interfaces, partial page updates identified in OFTL are compacted and co-located with the metadata for update reduction. With the system co-design with flash memory, write amplification from file systems is significantly reduced.

Acknowledgments

We would like to thank our shepherd Margo Seltzer and the anonymous reviewers for their insightful comments and detailed suggestions, which have substantially improved the content and presentation of this paper. We also thank Shuai Li for his help with the experimental setup and Guangyu Sun for his help with the presentation. This work is supported by the National Natural Science Foundation of China (Grant No. 60925006), the State Key Program of National Natural Science of China (Grant No. 61232003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the research fund of Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology.

References

[1] blktrace(8) - linux man page. `http://linux.die.net/man/8/blktrace`.

[2] Btrfs. `http://btrfs.wiki.kernel.org`.

[3] Dbt2 test suite. `http://sourceforge.net/apps/mediawiki/osdldbt`.

[4] Fusionio virtual storage layer. `http://www.fusionio.com/products/vsl`.

[5] Lasr system call io trace. `http://iotta.snia.org/tracetypes/1`.

[6] The nvm express standard. `http://www.nvmexpress.org`.

[7] Reiserfs. `http://reiser4.wiki.kernel.org`.

[8] strace(1) - linux man page. `http://linux.die.net/man/1/strace`.

[9] Ubifs - ubi file-system. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

[10] Yaffs. `http://www.yaffs.net`.

[11] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of USENIX 2008 Annual Technical Conference*, 2008.

[12] S. Boboila and P. Desnoyers. Write endurance in flash drives: measurements and analysis. In *Proceedings of the 8th USENIX conference on File and storage technologies (FAST)*, 2010.

[13] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 1992.

[14] L.P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, 2007.

[15] F. Chen, T. Luo, and X. Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[17] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[18] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[19] 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. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[20] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: a file system for virtualized flash storage. In *Proceedings of the 8th USENIX conference on File and Storage Technologies (FAST)*, 2010.

[21] Y. Kang, J. Yang, and E. L. Miller. Object-based scm: An efficient interface for storage class memories. In *Proceedings of IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[22] S. W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.

[23] D. Ma, J. Feng, and G. Li. Lazyftl: A page-level flash translation layer optimized for nand flash memory. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD)*, 2011.

[24] P. Macko, M. Seltzer, and K.A. Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the 8th USENIX conference on File and storage technologies (FAST)*, 2010.

[25] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

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

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[28] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[30] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. Technical report, IBM Almaden Reserach Center, 2012.

[31] M. Seltzer. Transaction support in a log-structured file system. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[32] David Woodhouse. Jffs2: The journalling flash file system, version 2. `http://sourceware.org/jffs2`.

[33] G. Wu and X. He. Delta-ftl: improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[34] Q. Yang and J. Ren. I-cash: Intelligently coupled array of ssd and hdd. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[35] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.