FTL Considered Harmful for Flash Storage

Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim*, and Arvind

Massachusetts Institute of Technology
*Seoul National University
(Submitted to SOSP '15)

Abstract

Existing flash-based SSDs employ a flash translation layer (FTL) to provide the abstraction of a generic block device. This high interoperability with HDDs, however, comes at the price of substantial inefficiency, in terms of extra I/O operations that degrade I/O performance and storage lifetime. FTL implementations also require considerable resources – multiple CPUs and a large amount of DRAM. In this paper, we propose a new **R**educed **I**/O **S**ystem **A**rchitecture (RISA) that refactors two main components of the I/O stack, namely the storage device and the file system. Unlike the FTL-based storage that runs complex management functions in the storage device, a RISA storage device is simplified to contain only low-level management functions which are required to provide reliable storage access. A RISA file system works in cooperation with the simplified RISA storage device and directly manages NAND flash in a more efficient manner by using rich system-level information. We have implemented RISA, including a RISA file system and the RISA storage device on a custom flash card, and evaluated its effectiveness using real-world applications. Our experiments show that RISA reduces the DRAM requirement by 128X, and improves performance and lifetime by up to 1.8X and 18% over the FTL-based storage solutions, respectively.

1. Introduction

NAND flash-based storage devices are replacing hard disk drives (HDDs) in many consumer devices. They are also finding increasing use in servers to speed up many applications of interest. Thanks to Moore's law and the introduction of advanced manufacturing technologies like 3D NAND [21], the prevalence of flash-based devices will continue to increase over the next decade.

In order to hide the physical properties of NAND flash and to provide interoperability with existing block I/O subsystems, NAND flash-based devices employ an intermediate software layer, called a flash translation layer (FTL) [1, 8]. Using the FTL is considered to be essential in high-performance flash storage. However, its usefulness must be reconsidered because of the following drawbacks.

First, FTL-based storage requires significant hardware resources due to the maintenance of a huge mapping table and the execution of complex firmware algorithms. Recent flash-based SSDs require multiple processors and large amount of DRAM (e.g., 3-4 ARM cores running at 300-600 MHz for consumer SSDs [40, 42], a quad-core CPU running at 1 GHz for enterprise SSDs [15], and 1 GB DRAM [40, 42]). Second, even with generous amount of resources, performance can be suboptimal. The FTL makes important decisions affecting storage performance and reliability, but these decisions are not always the best because they are made based on very limited information available at the level of the storage device. Third, the FTL-based storage works like a black-box – its inner-workings are completely hidden from the host file system, which can make the behaviors of flash storage unpredictable [39]. This is a serious problem for applications requiring balanced I/O fairness and low I/O latency.

Many researchers and SSD vendors have attempted to address these problems by *adding* more features to the storage side – (1) by running advanced FTL algorithms [10, 22, 26, 30, 35], (2) by adding custom interfaces to deliver system-level information to the FTL for better optimization [7, 11, 13, 23], and/or (3) by offloading host functions (e.g., the file system) to the storage device [24, 31, 44]. While these approaches alleviate some of the drawbacks with the FTL-based design, they do not solve the underlying issues with using the FTL and they come with increased design complexity and cost.

In this paper, we propose a **R**educed **I**/O **S**ystem **A**rchitecture (RISA) to overcome the problems of the FTL-based design. As the name implies, the design philosophy of RISA is fundamentally different from the FTL-based storage design. By refactoring two main components of the I/O stack, namely the file system and the storage device, RISA eliminates the necessity for almost all the functionalities of the FTL, *reducing* the design complexity of the storage device. A well-optimized file system directly manages NAND flash with minimum (but essential) supports from the storage side. The existing FTL-based design style, which is opposite to RISA, is called **C**omplex **I**/O **S**ystem **A**rchitecture (CISA) because of its complex design approach.

A RISA file system (RFS) is based on a log-structured file system (LFS) [41] that appends new data in a log and reclaims free space using garbage collection. RFS leverages these inherent design features of LFS which are well-suited to the physical natures of NAND flash. Because of its internal design, a conventional LFS still needs the FTL to hide the out-place update restriction of NAND flash. Unlike LFS, RFS sees underlying storage as an append-only device that is divided into physical chunks (segments). RFS then directly manages NAND flash using *built-in LFS modules*, such as a segment allocator and a segment cleaner. This eliminates the need for logical-to-physical mapping (to handle in-place updates) and garbage collection on the storage side, thus greatly reducing the resources required. This direct management of NAND flash at the host side also enables RFS to exploit rich system-level information, which improves both I/O performance and storage lifetime by reducing extra I/Os for storage management.

On the other hand, a RISA storage device (RSD) is free from almost all storage management tasks and is designed for reliable storage access and high I/O throughput. RSD contains bad-block management, wear-leveling and ECC, which are minimally required to offer error-free NAND access. For this purpose, RSD only maintains a tiny table. This is a reasonable design choice because an unreliable NAND substrate is more effectively handled by the storage controller. RSD also performs I/O queuing to maximally exploit I/O parallelism of multiple channels/ways. Finally, RSD exports standard block I/O interfaces, exposing a linear logical address space and three primitive operations (`READ`, `WRITE` and `TRIM`). These are sufficient for RFS to manage RSD.

We implement a proof-of-concept prototype of RISA, which includes a file system, flash firmware and a flash controller, on our custom flash platform. Our experiments, using micro benchmarks and real-world benchmarks, show that RISA improves I/O performance and storage lifetime by up to 1.8X and 18% over CISA with a conventional file system and a page-level FTL, respectively, while requiring a fraction (i.e., $\frac{1}{128}$) of DRAM.

This paper is organized as follows: Section 2 reviews existing I/O system architectures for flash storage. Section 3 describes RISA in detail. Section 4 evaluates the effectiveness of RISA over CISA. Finally, Section 5 concludes with summary and future directions.

2. Existing I/O System Architectures

NAND flash has very different characteristics compared to hard disks, including limited program/erase cycles, erase-before-write requirement and page/block organization [1, 8]. To hide these characteristics and to ease NAND flash access by a variety of applications and systems, several architectures have been proposed.

Flash File System: A flash file system, such as JFFS2 [17, 43] and YAFFS [32], is one of the well-known approaches to use NAND flash. A flash file system is designed with full knowledge of the physical properties and organization of NAND device. It handles physical NAND device packages by directly controlling I/O pin signals and I/O buses through low-level NAND-specific interfaces like ONFI [18]. Since an extra translation layer (e.g., FTL) is not necessary, a flash file system works very efficiently with NAND flash.

Despite these advantages, a flash file system has critical limitations for use in recent flash devices like SSDs and eMMCs. The internal flash storage organization (e.g., channel/way/plane organizations) and NAND chip properties are both complex to manage and specific for each storage vendor and process technology [1]. For example, YAFFS had to undergo a major version update from 1.0 to 2.0 when MLC NAND flash was introduced to the market which had different properties from SLC NAND. Storage vendors are also reluctant to divulge the internal architecture of their devices. For this reason, designing/optimizing a flash file system for various vendor-specific architectures is in fact difficult. The continuing decrease in reliability of NAND flash is another problem because this unreliable device can be more effectively managed inside the storage device where device-level information is available (e.g., effective wearing [9] and bit-error rates [38]). As a result, a flash file system is rarely used nowadays except in small embedded systems.

RISA offers the same advantages provided by a flash file system. However, unlike a flash file system, RISA hides internal storage architectures behind block I/O interfaces, which simplifies the development of file systems without knowledge of the underlying storage architecture and physical NAND properties. It is the role of RSD to directly control unreliable NAND devices and to provide the block I/O interface based on device-level information.

FTL-based Storage: The FTL-based storage is a promising alternative that addresses the problems of flash file systems [6, 8, 25]. The FTL running inside the storage device hides all of the physical details and performs storage management tasks, only exposing block I/O interfaces to the host system. Thus, file systems or applications designed for HDDs are able to run on top of the FTL-based storage. Because of this interoperability, it has become a de-facto standard in designing flash storage.

However, the FTL-based design has several drawbacks as well. To avoid in-place updates, the FTL has to maintain a mapping table that maps a logical address from the file system to a physical address of NAND flash. The mapping table requires a large amount of DRAM, and its size increases proportionally with SSD capacity. For example, if page-level mapping is used, a 1 TB SSD would require 1 GB of DRAM just for logical-to-physical mapping. The mapping table size can be reduced by employing a demand-based FTL [10] or a hybrid FTL [22, 26, 30]. However, these solutions come at the expense of extra overheads such as additional in-flash table management, read performance penalties and random write performance degradations. The FTL also has to make important decisions on flash management, such as how to do

address mapping and when to perform garbage collection. These algorithms require intensive computations and often incur many extra I/Os to move around the data.

Host-based FTL solutions like Fusion IO's DFS [3, 19, 20, 37] move key functions of the FTL to the host system (e.g., a device driver) where more powerful CPUs and larger DRAM are available. Supporting FTL functions in the host, however, cannot eliminate the root cause of employing an extra translation layer since it simply changes the software layer where the FTL runs. The host-based FTL still wastes considerable host resources and incurs many extra I/Os [19]. Moreover, since the details of underlying storage architectures and NAND technologies must be exposed to the host system, it has the same limitations as a flash file system.

RISA uses a fundamentally different approach from the FTL-based storage. RISA *eliminates* the need for employing a complex intermediate software layer in both the host device driver and the storage device. The RISA file system directly manages NAND flash using its built-in modules (that already exist in the conventional LFS), which makes it unnecessary to maintain a huge mapping table as well as to run address translation and garbage collection at the FTL level.

Optimizing FTL with Custom Interface: Delivering system-level information to the FTL has received a lot of attention from academia and industry because of its advantage in storage-level optimization [7, 11, 13, 23]. For example, file access patterns of applications are useful in detecting and separating hot-cold data, helping the FTL reduce garbage collection costs. Detailed timing behaviors of applications can be used to hint when to trigger garbage collection. To share such high-level information, non-standard and vendor-specific interfaces must be added to both the storage device and the host. Modifications to the OS kernel and the FTL must be made as well. Due to difficulty in standardization, this approach is only useful in specialized applications that require high performance under certain circumstances.

RISA runs high-level storage management algorithms that would have a large impact on performance in the host system. This allows RISA to easily exploit system-level information without adding custom interfaces and modifying the storage firmware.

Offloading Host Functions into FTL: Some techniques go one step further by offloading functions of the file system (e.g., file allocation) onto the storage device [24, 31, 44]. The FTL can fully exploit rich file-system information (e.g., inodes, dentries and data) and/or effectively combine its internal operations with the file system for better flash management. However, running part or all of file-system functions requires more hardware resources and greater storage design complexity. Additionally, the range of optimization is still limited to the file-system level because the application is separate from the storage device.

In summary, the existing I/O system architectures have evolved by *adding* more features to the I/O stack (e.g., advanced firmware algorithms, custom interfaces and host functions). Even if these approaches may have their own benefits, they fail to eliminate the fundamental inefficiency of employing an extra management layer. A common problem of these approaches is that they increase the design complexity and cost of the I/O system stack. This is why we call them complex I/O system architectures (CISA).

There were prior studies like NoFTL [14] and REDO [29] that attempted to eliminate the FTL from the device and the host. NoFTL has the same limitations as the host-based FTL because it just moves the FTL onto the database engine. REDO is similar to our study. However, since it targets low-end flash storage with two channels, it does not take into account technical issues that arise with larger channels/ways. It also does not consider a metadata management issue that affects performance and data integrity. Finally (but not least importantly), both of those works are based on simulation studies. Thus, it is difficult to understand their feasibility as well as the impact on performance in real systems.

3. RISA: Reduced I/O System Architecture

In this section, we describe the RISA file system and the RISA storage device in order.

3.1 RISA File System (RFS)

The detailed implementation of LFS is very different from one another [27, 28, 34], but their fundamental design concept is the same as Sprite LFS [41]. Therefore, we explain the high-level design of RFS by focusing on architectural differences between RFS and the conventional LFS, excluding implementation details commonly found in other LFS.

3.1.1 Layout and Operations

Figure 1 shows the logical layout of RFS, along with the corresponding physical layout in RSD. RFS sees a storage space as one huge log divided into segments, which we call logical segments. An individual segment is a unit of free-space allocation and garbage collection. All user files, directories and inodes, including any modifications/updates, are appended to free space in logical segments, called *data segment*. RFS maintains an inode map to keep track of inodes scattered across the storage space. The inode map is stored in reserved logical segments, called inode-map segments. RFS also maintains a check-point that points to the inode map and keeps the consistent state of the file system. A check-point is written periodically or when an explicit flush command (e.g., `fsync`) is issued. Logical segments reserved for check-points are called check-point segments.

RFS operates differently from the conventional LFS in following two ways. *First, in-place updates are never allowed.* Many believe that LFS does not overwrite anything, but this is not true. For fast recovery and easy inode management, LFS writes the check-point and the inode-map in an in-place-update manner [1]. The FTL should map update

[1] This is somewhat different depending on the design of LFS. Sprite LFS overwrites data in a check-point region only [41], while NILFS writes seg-

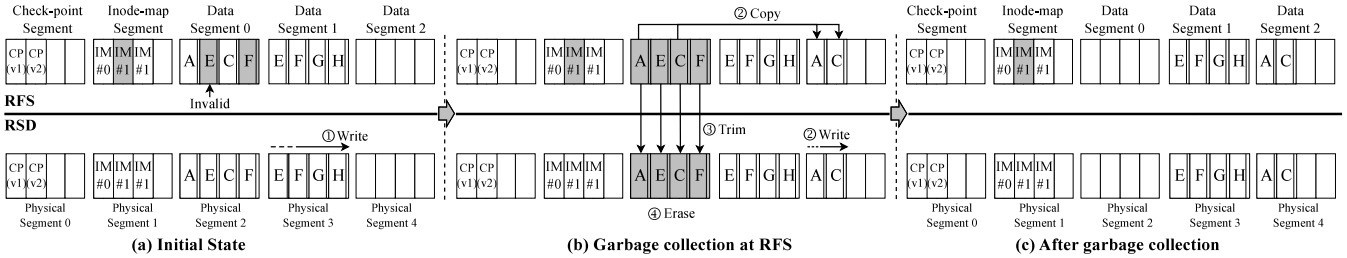


Figure 2: An example of RFS with RSD: (a) four new files E, F, G and H are written to the data segment 1. The files E and F are new versions. After writing them, RFS appends IM#1 to the inode-map segment because it points to the locations of the files E, F, G and H. A new check-point CP(v2) is appended as well. (b) Free space in RFS is nearly exhausted, so RFS triggers garbage collection. RFS copies the files A and C to the data segment 2. Since the data segment 0 has only invalid data, RFS sends TRIM commands to RSD, making it free. Finally, RSD erases the physical segment 2.

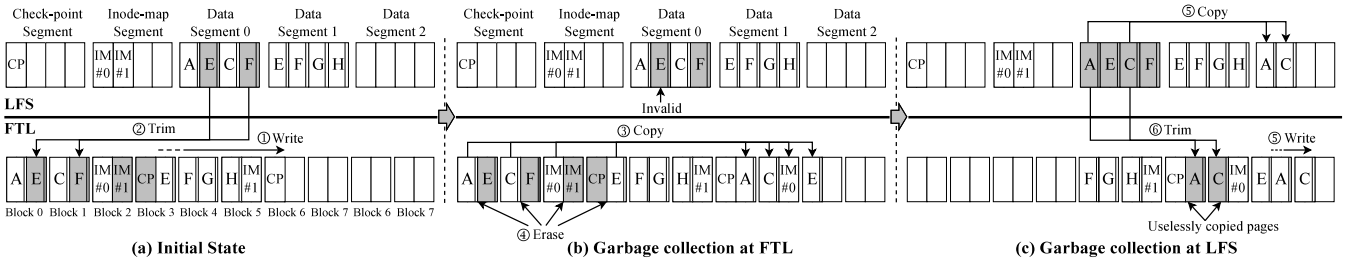


Figure 3: An example of LFS with FTL: The FTL sequentially writes all the file-system blocks to NAND flash using a mapping table according to their arrival times. (a) The files E, F, G and H are appended to free pages. LFS sends TRIM commands to the FTL because the old files E and F are obsolete. IM#1 and CP are overwritten in the same file-system locations. The FTL maps them to free pages using the mapping table, invaliding old versions. (b) The FTL decides to perform garbage collection. It copies flash pages for A, C, IM#0 and E to free pages and gets four free blocks (Blocks 0, 1, 2, 3). (c) LFS is unaware of the FTL, so it also triggers garbage collection to create free space in the file system. It moves the files A and C to free space and sends TRIM commands. The files A and C are moved twice uselessly.

data to free space using a mapping table. To reclaim free space occupied by obsolete data, the FTL should perform garbage collection. By performing out-place updates even for the check-point and the inode-map, RFS eliminates the necessity of employing such functionalities in RSD.

Second, a logical segment in RFS directly corresponds to a physical segment in RSD. As depicted in Figure 1, a physical segment is the group of flash blocks organized to fully exploit I/O parallelism for sequential writes. For example, if RSD has two channels and one way, then a physical segment would be composed of two blocks, one on each channel. This way, the data layout of a logical segment perfectly aligns with its physical segment, allowing RFS to manage NAND flash directly in the unit of a logical segment without knowledge of the underlying storage organizations. RFS also uses a logical segment as the unit of TRIM to inform RSD that which segments contain invalid data.

Figures 2 and 3 compare the behavior of RFS and conventional LFS for the same set of file operations. For the sake of simplicity, we assume that RFS and LFS have the same file-system layout. The sizes of a file system block and a flash page are the same. On the storage device side, LFS runs the page-level FTL that maps logical file-system blocks to any physical pages in NAND flash. In RSD, a physical segment is composed of two flash blocks. RSD just erases flash blocks containing only obsolete pages.

Figure 2 demonstrates how RFS manages NAND flash without any help from the FTL. In particular, RFS incurs fewer number of reads and writes for garbage collection than LFS. In the conventional LFS, two different layers, the file system and the FTL, *doubly* manage the same NAND flash using different storage management and garbage collection policies. Because of this duplicated management, unnecessary page copies often occur in LFS. Since RFS directly manages NAND flash, this problem does not occur.

It must be noted that, even though LFS seems to work poorly with the FTL, it often shows better performance than journaling file systems like EXT4 [28, 34]. This is because LFS incurs smaller in-place updates to NAND flash, thereby reducing live page copies in FTL garbage collection. However, when the storage space is highly fragmented with valid and obsolete data and many live data copies are involved both in LFS and the FTL, the performance of LFS is significantly degraded due to duplicate garbage collection. We show this problem in detail in experimental sections.

Operationally, RFS has a few key issues that must be addressed for it to function correctly and efficiently. The conventional LFS keeps check-points in a fixed location. By reading the latest check-point from that location, LFS quickly returns to the consistent state when it is mounted or

ment summary blocks in an out-place-update fashion [27]. F2FS overwrites both a check-point and an inode map [28].

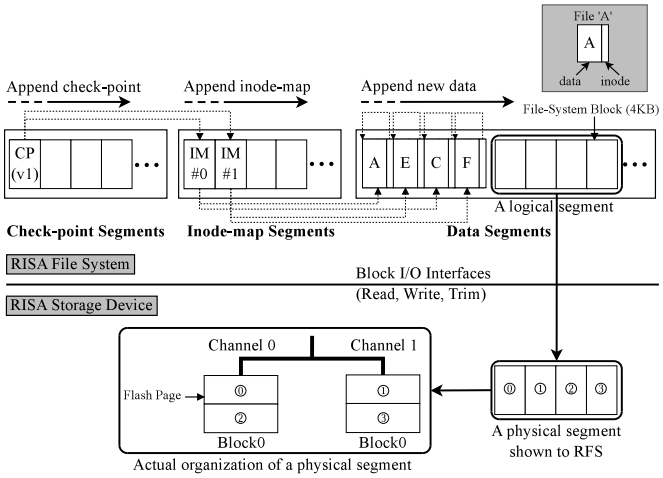


Figure 1: The upper figure illustrates the logical layout of RFS. Four files are appended to data segments along with their inodes in the following order: A, E, C and F. Then, two pieces of the inode map, IM#0 and IM#1, are written. While IM#0 points to the locations of inodes for files A and C, IM#1 indicates those for files E and F. Finally, the check-point CP(v1) is written to check-point segments. The bottom figure shows the organization of a physical segment corresponding to the logical segment. Individual file-system blocks (denoted by numbers inside circles) are statically mapped to flash pages to maximize I/O parallelism.

after power failure. LFS also often keeps the inode map in the fixed locations, which is sorted by ascending inode numbers. This allows LFS to easily find the location of inodes by using an inode number as index for the inode map. Updating the inode map in place is useful to remedy the wondering tree problem [28, 41]. Unlike LFS, RFS always appends all the changes to check-point and inode-map segments, so their locations are not fixed. This makes it difficult to find the latest check-point and the locations of inodes.

In the next sections, we explain how RFS manages check-point segments for quick mount and recovery, and show how it handles inode-map segments for fast searches of inodes.

3.1.2 Check-Point Segment

The management of check-point segments is very straightforward. RFS reserves two logical segments, logical segments #1 and #2, for check-point segments. (Note: logical segment #0 is reserved for a superblock). RFS appends new check-points with different version numbers using the available free space. If free space in both segments is exhausted, the segment containing only old check-point versions is selected as a victim for erasure. The latest check-point is still kept in the other segment. RFS sends TRIM commands to invalidate and free the victim, which can then be reused to write new check-points. Note that even though RFS repeatedly uses the same logical segments, it will not unevenly wear out NAND flash because RSD performs wear-leveling. This will be discussed in Section 3.2.

Data structure	Unit size	Unit count	Storage
Inode-map block	4 KB	524,288	Flash (inode-map segments)
TIMB	2 MB	1	DRAM
TIMB block	4 KB	512	Flash (inode-map segments)
TIMB-blocks list	2 KB	1	Flash (a check-point)

Table 1: An example of data structures sizes and locations with a 1 TB SSD. Their actual sizes could vary depending on RFS implementation (e.g., an inode-map size) and storage capacity.

When RFS is remounted, it reads all the check-point segments from RSD. It finds the latest check-point by comparing version numbers of all of the candidates. This brute-force search is efficient because RFS maintains only two segments for check-pointing, regardless of storage capacity. Moreover, since segments are organized to maximize I/O throughput, this search utilizes full bandwidth and mount time is short.

3.1.3 Inode-Map Segment

The management of inode-map segments is much more complicated than check-point segments. The inode map size is decided by the maximum number of inodes (i.e., files) and is proportional to storage capacity. If the storage capacity is 1 TB and the minimum file size is 4 KB, 2^{28} files can be created. Suppose each entry of the inode map is 8 B (4 B for an inode number and 4 B for its location in a data segment), then the inode map size is 2 GB. Because of its large size, LFS divides the inode map into 4 KB blocks, called *inode-map blocks*. There are 524,288 inode-map blocks for the inode map of 2 GB (see Table 1). For example, IM#0 and IM#1 in Figures 1 and 2 are inode-map blocks. Each inode-map block contains the mapping of 512 inodes.

RFS always appends inode-map blocks to free space, so the latest inode-map blocks are scattered across segments. For this reason, we need another management scheme to quickly identify the latest valid inode-map blocks.

Inode-Map Block Management: To quickly find the locations of inodes, RFS maintains a table for inode-map blocks (TIMB) in the main memory. TIMB consists of 4-byte entries that point to inode-map blocks in inode-map segments. Given an inode number, RFS finds its inode-map block by looking up TIMB. It then obtains the location of the inode from that inode-map block. The TIMB size is 2 MB for 524,288 inode-map blocks, so it is small enough to be kept in the host DRAM. In-memory TIMB should be persistently stored in the storage device; otherwise, RFS has to scan all inode-map segments to construct TIMB during mount. RFS divides TIMB into 4 KB blocks (TIMB blocks) and keeps track of dirty TIMB blocks that hold newly updated entries. RFS appends dirty TIMB blocks to free space in inode-map segments just before a check-point is written.

TIMB blocks themselves are also stored in non-fixed locations. To easily build in-memory TIMB and to safely keep it against power failures, a list of all the physical locations of TIMB blocks (TIMB-blocks list) is written to check-point segments together with the latest check-point. The number of TIMB blocks for 1 GB of inode-map blocks is 512, so the

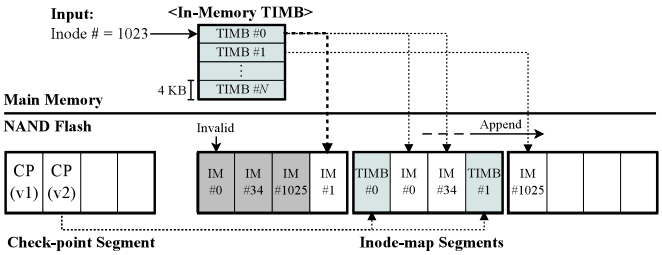


Figure 4: Individual 4 KB TIMB blocks indicate 1,024 inode-map blocks in inode-map segments. For example, TIMB#0 points to IM#0∼IM#1023 in flash, and each IM indicates 512 inodes. Up-to-date TIMB blocks are written to inode-map segments before a check-point is written. The newly written check-point indicates all of the physical locations of TIMB blocks in flash. If RFS searches for the location of an inode whose number is 1023, it looks up TIMB#0 in the in-memory TIMB and finds the location of IM#1 that points to 512∼1023 inodes in data segments.

size of a TIMB-blocks list is 2 KB. The actual size of check-point data is several hundred bytes (e.g., 193 bytes in F2FS), thus a check-point with a TIMB-block list can be written together to one 4 KB file-system block without incurring additional writes. Figure 4 illustrates how RFS manages inode-map blocks and TIMB blocks.

Remount Process: In-memory TIMB should be reloaded properly whenever RFS is mounted again. This remount process is simple. RFS first reads the latest check-point as we described in the previous subsection. Using a TIMB-blocks list in the check-point, RFS reads all of the TIMB blocks from inode-map segments and builds TIMB in the host DRAM. The time taken to build TIMB is negligible because of its small size (e.g., 2 MB for 1 TB storage).

A new check-point is materialized to NAND flash after up-to-date TIMB blocks and inode-map blocks are written to inode-map segments. By reading the latest check-point successfully written to check-point segments, RFS returns to the consistent state after sudden power failures. All the TIMB blocks and inode-map blocks belonging to an incomplete check-point are regarded as obsolete data. Note that the recovery process of RFS is the same as the remount process because RFS is based on LFS [41].

Garbage Collection: When free space in inode-map segments is almost used up, RFS should perform garbage collection. In the current implementation, the least-recently-written inode-map segment is selected as victim. All valid inode-map blocks in the victim are copied to a free inode-map segment that has already been reserved for garbage collection. Since some of inode-map blocks are moved to the new segment, in-memory TIMB should also be updated to point to their new locations accordingly. Newly updated TIMB blocks are appended to the new segment, and the check-point indicating TIMB blocks is written to the check-point segment. Finally, the victim segment is invalidated by a TRIM command and becomes a free inode-map segment.

To reduce live data copies, RFS increases the number of inode-map segments such that their total size is larger than the actual inode-map size. This wastes file-system space but greatly improves garbage collection efficiency. Since inode-map blocks have higher temporal/spatial localities than user data, many blocks are invalid before garbage collection is triggered, which means there is generally less data to copy. In addition, by separating inode-map blocks (i.e., hot data) in inode-map segments from data segments (i.e., cold data), RFS further improves garbage collection efficiency. Currently, RFS allocates inode-maps segments four times larger than its original size (e.g., if the inode map size is 2 GB, 8 GB is assigned to inode-map segments). The space wasted by these extra segments is small (e.g., 0.68% = 7 GB / 1 TB).

All of the I/O operations required to manage inode-map blocks are extra overheads that are not present in the conventional LFS. In our observation, those extra I/Os account for a small proportion, which is less than 0.2% of the total I/Os.

3.1.4 Data Segment

RFS manages data segments in a similar manner as the conventional LFS. RFS buffers file data, directories and inodes in DRAM and writes them to the storage device all at once when their total size reaches a data segment size. This buffering is particularly advantageous for RFS – it can make use of the full bandwidth of RSD because a segment is organized to maximize device I/O throughput. RFS performs segment cleaning when free data segments are nearly exhausted. As shown in Figure 2, RFS selects a victim segment and copies valid data to a free segment. RFS sends TRIM commands to inform RSD that all flash blocks belonging to the victim is obsolete. Finally, the victim segment is freed.

In order to reduce cleaning costs, RFS borrows well-known optimization techniques from the existing LFS implementation (e.g., hot-cold separation, victim selection and segment cleaning). Many flash devices adopt similar techniques in the FTL, but running such techniques at the file system level is more efficient because system-level information can be easily exploited, in addition to high-performance CPUs and fast/large DRAM available at the host.

3.1.5 Segment Size

A logical segment size is an important parameter in both RFS and LFS. Choosing a large segment size is generally beneficial for utilizing the full bandwidth of the storage device. However, a segment size that is too large can incur more live data copies during segment cleaning [33]. For this reason, in the conventional LFS, a segment size is limited to several MB (e.g., 2 MB) [28, 33]. In RFS, a logical segment must be the same as a physical one which is decided by the number of channels and ways in RSD. Modern flash devices tend to have a larger number of channels and ways for better performance. Thus, the segment size of RFS could be much larger than the conventional LFS, which potentially results in the significant increase of segment cleaning costs.

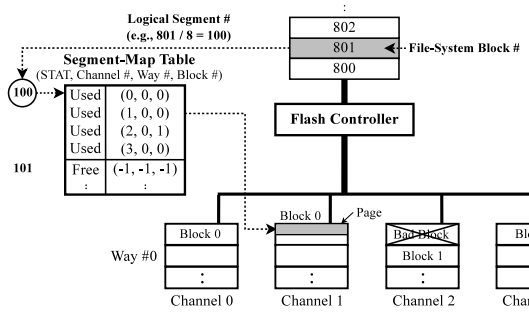


Figure 5: An example of how RSD handles writes using a segment-map table. There are four channels and one way in RSD, and each block is composed of two pages. A physical segment has 8 pages. When a write request comes, RSD gets a logical segment number (i.e., 100 = 801 / 8) using the logical file-system block number. It then looks up the segment-map table to find a flash block mapped to the logical segment. In this example, the logical block '801' is mapped to 'Block 0' in 'Channel #1'. Finally, RSD writes the data to a corresponding page offset in the mapped block.

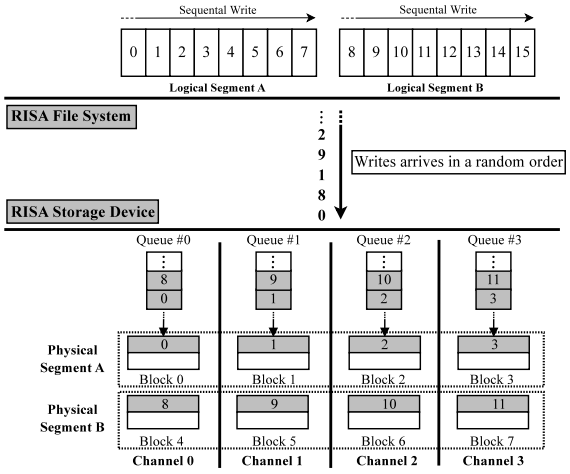


Figure 6: An example of how RSD handles write requests when RFS appends data to two segments A and B simultaneously. Numbers inside rectangles indicate a file-system logical block address. RFS sequentially writes data to individual segments A and B, but write requests arrive at RSD in a random order (i.e., 0, 8, 1, ...). They are sorted in multiple I/O queues according to their destined channels and are written to physical segments in a way of fully utilizing four channels. If a single queue with FIFO scheduling is used, the file-system block '1' is delayed until '0' and '8' are sent to flash blocks '0' and '4' through the channel 0.

According to our experiments, RFS shows better performance than the conventional LFS as well as EXT4 even with a large number of channels/ways (e.g., 8 channels/8 ways with a 32 MB segment). This is because the increase in live data copies is fewer than extra I/Os incurred by FTL garbage collection. However, when a segment size becomes very huge (e.g., 128 MB), increased segment cleaning costs outweigh its benefits. We analyze it in detail in Section 4.5 and discuss our direction to solve such a potential problem.

3.2 RISA Storage Device (RSD)

Since RFS handles most of the complex storage management tasks, RSD can be significantly simpler. The architecture of RSD is similar to a simplified version of existing flash storage with the block-level FTL [2]. However, unlike the block-level FTL, RSD does not need to remap logical file-system blocks to physical locations in NAND flash to avoid in-place updates as well as not to perform block merges for garbage collection. In this subsection, we describe the minimum requirements for RSD implementation needed to offer error-free NAND access and high I/O throughput.

3.2.1 Wear-Leveling and Bad-Block Management

As discussed in Section 3.1.1, file-system blocks in a logical segment are statically mapped to physical flash pages in a physical one. For wear-leveling and bad-block management purposes, RSD only needs a small segment-map table that maps a logical segment to a physical one. Each table entry contains the physical locations of flash blocks that are mapped to a logical segment along with a segment status flag (`STAT`). Each table entry belonging to the same logical segment points to flash blocks in spread among multiple channels and ways. `STAT` indicates Free, Used or Invalid.

Figure 5 shows how RSD handles write requests. If any physical blocks are not mapped yet (i.e., `STAT` is Free or Invalid), RSD builds the physical segment by allocating new flash blocks. A bad block is not selected. RSD picks up the least worn-out free blocks in the corresponding channel/way. To preserve flash lifetime and reliability, RSD can perform static wear-leveling that exchanges the most worn-out segments with the least worn-out ones in background [4]. If there are previously allocated flash blocks (i.e., `STAT` is Invalid), they are erased. If a logical segment is already mapped (i.e., `STAT` is Used), RSD writes the data to the fixed location in the physical segment. RFS informs RSD via TRIM commands the physical segments have only obsolete data. Then, RSD can figure out which blocks are out-of-date. Upon receiving the TRIM command, RSD invalidates that segment by changing its `STAT` to Invalid. Invalid segments are erased in on-demand or background later.

3.2.2 I/O Queueing

To maximally exploit the I/O parallelism, RSD employs per-channel/way I/O queues combined with a simple FIFO-based I/O scheduler. This multiple I/O queueing is effective in handling random writes. RFS allocates multiple segments and writes data to different segments at the same. For example, a check-point is often written to check-point segments while user files are being written to data segments. For this reason, write requests arriving at RSD could be random even if RFS sends write requests to individual segments sequentially. This degrades I/O parallelism. Figure 6 shows how RSD handles random writes using multiple queues.

In RSD, write skews do not occur for any channel or way. This is because RFS allocates and writes data in the unit of a segment, distributing all the write requests to channels and

ways uniformly. Moreover, since FTL garbage collection is never invoked in RSD, I/O scheduling between normal I/Os and GC I/Os is not required. Consequently, simple multiple I/O queueing is efficient enough to offer good performance, and complex firmware algorithms like load-balancing [5] and out-of-ordering [12, 36] are not required in RSD.

4. Experimental Results

4.1 Evaluation Setup

We implemented RFS in the Linux kernel 3.13. Instead of developing from scratch, we modified the F2FS file system – a recently released LFS [28]. Besides check-points and an inode-map, F2FS overwrites some file-system metadata, including a segment information table (SIT) and a segment summary area (SSA). To avoid these in-place updates, we modified F2FS so that it appended SIT and SSA to inode-map segments. F2FS also employed many useful features for NAND flash, such as hot-cold separation, victim selection and advanced segment cleaning. RFS borrowed all of these features from F2FS – RFS reused the exactly same file management and garbage collection modules used by F2FS. Our modifications to transform F2FS into RFS added approximately 1,200 lines of code, which was relatively minor. This shows that RFS can be easily ported to other LFS.

RSD was implemented in our in-house FPGA-based PCIe SSD prototype. Our SSD prototype had 8 channels and 4 ways with 512 GB of NAND flash, achieving 240K IOPS and 67K IOPS for reads and writes, respectively. The flash page size was 4 KB and the number of pages per block was 128. We synthesized on the FPGA an ECC engine for bit error correction and a custom flash controller to communicate with raw NAND chips. We implemented bad-block management, wear-leveling and I/O queuing modules in the block device driver because our SSD prototype did not have a dedicated embedded processor. All those software modules, however, can be implemented at the level of a storage device if an embedded processor is available.

We compared RISA against two different CISA configurations: $\texttt{CISA}_{\text{EXT4}}$ and $\texttt{CISA}_{\text{F2FS}}$. $\texttt{CISA}_{\text{EXT4}}$ used the EXT4 file system on top of a page-level FTL. $\texttt{CISA}_{\text{F2FS}}$ was a combination of the original (unmodified) F2FS file system and the page-level FTL. The page-level FTL was based on pure page-level mapping with greedy garbage collection. An over-provisioning area was set to 5% of the storage capacity. To exploit multiple channels/ways, it also employed I/O interleaving and multiple I/O queueing policies. The FTL garbage collection was triggered when the remaining free space was less than 1%. To offer the best I/O parallelism when writing incoming user data later, the FTL reclaimed free space in all the channels and ways. Creating free space for all the channels/ways at the same time was also beneficial to achieve good garbage collection throughput. Our SSD prototype had 8 channels with 4 ways, so the FTL reclaimed 32 blocks when garbage collection was invoked.

Capacity	Block-level FTL	Hybrid FTL	Page-level FTL	RSD
2 GB	16 KB	121 KB	2 MB	16 KB
512 GB	4 MB	32 MB	512 MB	4 MB
1 TB	8 MB	62 MB	1 GB	8 MB

Table 2: FTL mapping table sizes

For EXT4, a default journaling mode was used and a discard option was enabled to use TRIM commands. For F2FS, a segment size was always set to 2 MB. In the case of RFS, a segment size was set to 16 MB which was equal to a physical segment size. RFS allocated inode-map segments which were four times larger than its original size. For both F2FS and RFS, 5% of file-system space was used as an over-provisioning area for file-system garbage collection.

4.2 Memory requirements

We compare the mapping table sizes of RSD with three FTL schemes, block-level, hybrid and page-level FTLs. While the block-level FTL uses a flash block (512 KB) as the unit of mapping, a page (4 KB) is used for address mapping in the page-level FTL. The hybrid FTL is a combination of block-level and page-level FTLs – while the block-level mapping is used to manage the storage space offered to end-users, the page-level mapping is used for an over-provisioning area. For the hybrid FTL, 5% of the total capacity is used as the over-provisioning area, which is equal to that in the page-level FTL. RSD maintains the segment-map table pointing to flash blocks for wear-leveling and bad-block management.

Table 2 lists the mapping table sizes for storage capacities of 2 GB, 512 GB and 1 TB. In particular for a 2 GB SSD, the mapping table sizes are 16 KB, 121 KB, 2 MB and 16 KB for block-level, hybrid, page-level FTLs and RSD, respectively. The mapping table sizes increase in proportional to the storage capacity – when the capacity is 1 TB, block-level, hybrid, page-level FTLs and RSD require 8 MB, 62 MB, 1 GB and 8 MB memory, respectively.

The mapping table size of the page-level FTL is very large, but in our experiments, we assume that there is sufficient DRAM memory for the entire mapping table. In practice, all of the mapping table entries are often too large to be kept in DRAM, and the FTL should store them all in NAND flash, only caching popular mapping entries in DRAM. The management of in-flash mapping entries incurs many extra I/Os (e.g., extra reads/writes and additional garbage collection for mapping entries kept in NAND flash) [10]. *In our experiments, we disregard these extra I/Os thus giving a baseline advantage to our competing setups of* $\texttt{CISA}_{\text{EXT4}}$ *and* $\texttt{CISA}_{\text{F2FS}}$ *that uses the page-level FTL.*

RSD maintains a much smaller mapping table than the page-level FTL, enabling us to keep all mapping entries in DRAM. In the following subsection, we demonstrate that RISA shows better performance than $\texttt{CISA}_{\text{EXT4}}$ and $\texttt{CISA}_{\text{F2FS}}$ in addition to using less memory.

Category	Workload	Description
File System	FIO	A synthetic I/O workload generator
	Tiobench	A multi-threaded workload generator
	Iozone	A file-system benchmark tool generating various sizes of I/O requests
Database	Non-Trans	A non-transactional DB workload
	OLTP	An OLTP workload
	TPC-C	A TPC-C workload
Hadoop	DFSIO	A HDFS I/O throughput test application
	TeraSort	A data sorting application
	WordCount	A word count application

Table 3: A summary of benchmarks

4.3 Performance Analysis

We evaluated RISA using 9 different workloads, spanning 3 categories: file-system, DBMS and Hadoop applications (see Table 3). To understand the behaviors of RISA under various file-system operations, we conducted a series of experiments using three well-known file system benchmarks, FIO, Tiobench and Iozone. We also evaluated RISA using response-time-sensitive database workloads: Non-Trans, OLTP and TPC-C. Finally, we assessed the performance of RISA with Hadoop applications from HiBench [16], HFSIO, TeraSort and WordCount, which required high I/O throughput for batch processing. We explain detailed benchmark settings and show our experimental results below.

For performance measurements, we focused on analyzing the effect of garbage collection on write performance. $\text{CISA}_{\text{EXT4}}$, $\text{CISA}_{\text{F2FS}}$ and RISA all perform very differently from the perspective garbage collection. For $\text{CISA}_{\text{EXT4}}$, since EXT4 is a journaling file system, only the FTL in the storage device performs garbage collection. In $\text{CISA}_{\text{F2FS}}$, both F2FS and the FTL do garbage collection. In RISA, only RFS performs garbage collection.

For fast experiments, we set the storage capacity to 2 GB, with the exception of Hadoop benchmarks where we used 8 GB to accommodate the entire data set. The benchmarks were configured to issue a large number of I/Os such that sufficient I/O traffic for garbage collection would be generated. The host DRAM size was set to 512 MB (i.e. smaller than the storage capacity) to ensure that requests were not entirely served from the page cache.

4.3.1 File System Benchmarks

FIO: We evaluate sequential and random read/write performance using the FIO benchmark. FIO creates a single file (1.4 GB) and performs sequential reads (SR), random reads (RR), sequential writes (SW) and random writes (RW) on

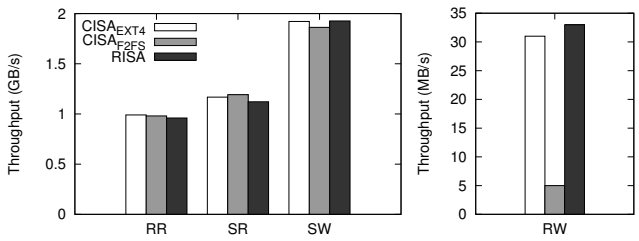


Figure 7: Experimental results with FIO

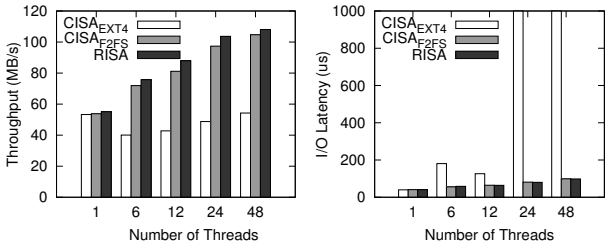
	$\text{CISA}_{\text{EXT4}}$	$\text{CISA}_{\text{F2FS}}$		RISA
	FTL GC	FS GC	FTL GC	FS GC
FIO (SW)	1.00	1.01	1.00	1.01
FIO (RW)	1.24	1.43	2.80	1.38
Tiobench (w/ 1 thread)	1.04	1.03	1.02	1.02
Tiobench (w/ 48 threads)	1.15	1.04	1.02	1.02
Iozone	1.14	1.05	1.09	1.09
Non-Trans	1.97	1.58	2.90	1.59
OLTP	1.45	1.23	1.78	1.24
TPC-C	2.33	1.81	2.80	1.87
DFSIO	1.0	1.0	1.0	1.0
TeraSort	1.0	1.0	1.0	1.0
WordCount	1.0	1.0	1.0	1.0

Table 4: Write amplification factors (WAF). For $\text{CISA}_{\text{F2FS}}$, we display WAF values for both the file system (FS) and the FTL. In FIO, the WAF values for the read-only workloads FIO (RR) and FIO (SR) are not included. For Tiobench, the WAF values with 1 and 48 threads are shown.

it separately. We empty the page cache prior to running the benchmark. Figure 7 shows our experimental results. $\text{CISA}_{\text{EXT4}}$, $\text{CISA}_{\text{F2FS}}$ and RISA all show excellent performance for SR and RR, reaching the maximum read throughput of our SSD prototype. The throughput of SW is higher than that of SR due to write buffering by the page cache. These high performance numbers can be realized with very little garbage collection in the file system and the FTL.

For RW that tends to incur many extra copies for garbage collection, RISA outperforms both $\text{CISA}_{\text{EXT4}}$ and $\text{CISA}_{\text{F2FS}}$. RISA achieves 1.07X higher bandwidth than $\text{CISA}_{\text{EXT4}}$. In general, RISA has smaller write amplification factors (WAF) than $\text{CISA}_{\text{EXT4}}$ (see Table 4) because it performs garbage collection more intelligently with information from the file system. However, for FIO with RW, we observe that the WAF of RISA is greater than $\text{CISA}_{\text{EXT4}}$. This is because FIO creates only a single file and writes 4 KB data pages randomly, so it is difficult to leverage file-system information (e.g., metadata) as well as data locality for better garbage collection. Furthermore, while RISA selects a logical segment (i.e., 16 MB) with the fewest valid data for garbage collection, $\text{CISA}_{\text{EXT4}}$ can select smaller flash blocks (i.e., 512 KB) with fewest valid pages in a particular channel/way. $\text{CISA}_{\text{EXT4}}$ thus incurs fewer live page copies than RISA.

Despite the larger WAF value in RW, RISA works more efficiently than $\text{CISA}_{\text{EXT4}}$. This is because RISA can exploit page caching in the host system for garbage collection. If the data to move for garbage collection is available in the page cache, RISA writes it directly to the storage device without first reading it from the device. Although the page cache size is relatively small compared to the working set of the benchmarks, considerable data can often be found. Thus, RISA just needs to write buffered data along with new user data to the current active segments, and then it simply reclaims free space by invalidating the victim segment. $\text{CISA}_{\text{EXT4}}$ is unaware of the existence or contents of the page cache, so it must first read all of the valid pages from NAND flash during garbage collection. Since the FTL has to stop servicing normal I/O requests to create sufficient free space, user-perceived performance is inevitably degraded.



(a) Sequential writes (b) Random writes

Figure 8: Experimental results with `Tiobench`

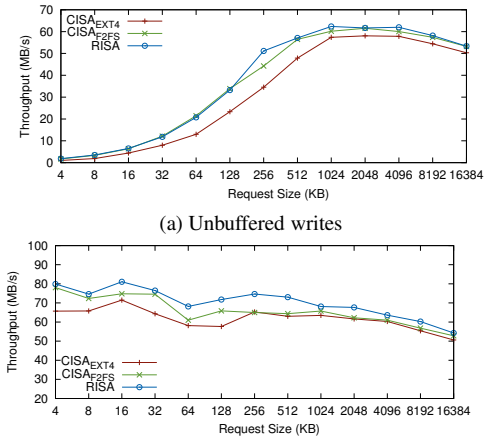
Finally, $\texttt{CISA}_{\text{F2FS}}$ shows the worst performance among all the storage configurations. As listed in Table 4, $\texttt{CISA}_{\text{F2FS}}$ file system has similar WAF values as `RISA`. However, due to high garbage collection costs in the FTL, its performance is greatly degraded. This inefficiency comes from duplicated garbage collection. In our observation, the storage space becomes highly fragmented with RW, which incurs many writes for segment cleaning. In $\texttt{CISA}_{\text{F2FS}}$, these extra writes from F2FS also trigger additional garbage collection in the FTL, and this results in high garbage collection overheads.

Tiobench: We evaluate the performance of RISA under multi-threaded workloads generated by `Tiobench`. Our experiments vary the number of threads from 1 to 48. Each thread creates a file and performs two different types of I/O operations: sequential writes followed by random writes.

Figure 8 illustrates our experimental results. `RISA` and $\texttt{CISA}_{\text{F2FS}}$ both show good aggregate throughput – write throughput of `RISA` and $\texttt{CISA}_{\text{F2FS}}$ is improved as the number of threads increases. Overall write latency for a 4 KB block increases from 41μs to 98μs when 48 threads are running, but relatively speaking, it is not a serious degradation. Unlike `FIO`, $\texttt{CISA}_{\text{F2FS}}$ works well in `Tiobench` because of less storage fragmentation with lower file-system level cleaning costs. $\texttt{CISA}_{\text{EXT4}}$ suffers from high garbage collection overheads under multi-threaded workloads. As listed in Table 4, the WAF value of $\texttt{CISA}_{\text{EXT4}}$ increases with multiple threads. Due to a large number of extra I/Os generated by the FTL, user I/O requests are often delayed and this results in low write throughput with high I/O latency.

Iozone: Using `Iozone`, we evaluate the performance of different I/O request sizes. `Iozone` creates various sized files ranging from 64 KB to 256 MB and performs write operations with different sizes from 4 KB to 16 MB. We study two types of writes: unbuffered (direct) writes and buffered writes. For unbuffered writes, `Iozone` opens files with a `O_SYNC` option so that file data and associated metadata are completely transferred to the storage device before finishing `write()` calls. This allows us to measure write performance without a page cache. For buffered writes, `Iozone` uses a `fwrite()` function that keeps user data in a temporal buffer before sending it to the storage device.

Figure 9 illustrates write throughput according to different sizes of write requests. For all request sizes, `RISA` shows the best performance for both unbuffered writes and



(a) Unbuffered writes

(b) Buffered writes

Figure 9: Experimental results with `Iozone`

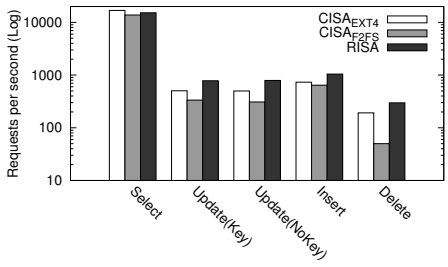
buffered writes with smallest garbage collection overheads. $\texttt{CISA}_{\text{F2FS}}$ outperforms $\texttt{CISA}_{\text{EXT4}}$, but cannot surpass `RISA` because of a relatively large number of extra copy operations in the FTL which are not observed in `RISA`.

4.3.2 Application Benchmarks

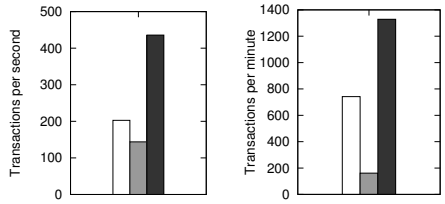
Database Application: We compare the performance of $\texttt{CISA}_{\text{EXT4}}$, $\texttt{CISA}_{\text{F2FS}}$ and `RISA` using database systems benchmarks. MySQL 5.5 with an Innodb storage engine is selected as the DBMS. Default parameters are used for both MySQL and Innodb. `Non-Trans` is used to evaluate performance with different types of queries: Select, Update (Key), Update (NoKey), Insert and Delete. The non-transactional mode of a SysBench benchmark tool is used to generate individual queries. `OLTP` is an I/O intensive online transaction processing (OLTP) workload generated by the SysBench tool. For both `Non-Trans` and `OLTP`, the table size is set to 5,000,000 and 6 threads run simultaneously. `TPC-C` is a well-known OLTP workload. We run `TPC-C` on 14 warehouses with 16 clients each for 1,200 seconds.

Figure 10 shows the number of transactions performed under the different configurations. `RISA` outperforms $\texttt{CISA}_{\text{EXT4}}$ and $\texttt{CISA}_{\text{F2FS}}$ under these complex I/O workloads. Performance gains of `RISA` primarily come from less garbage collection overheads. For `Non-Trans`, `OLTP` and `TPC-C`, the WAF values of `RISA` are 20%, 15% and 20% lower than those of $\texttt{CISA}_{\text{EXT4}}$, respectively. For the Select query in `Non-Trans`, `RISA`, $\texttt{CISA}_{\text{EXT4}}$ and $\texttt{CISA}_{\text{F2FS}}$ show similar performance because it is a read-only workload. $\texttt{CISA}_{\text{F2FS}}$ exhibits the worst performance because of high garbage collection overheads in the file system and the FTL.

Hadoop Application: We show measured execution times of Hadoop applications in Figure 11. Hadoop applications run on top of the Hadoop Distributed File System (HDFS) which manages distributed files in large clusters. HDFS does not directly manage physical storage devices. Instead, it runs on top of regular local disk file systems, such as EXT4, which deal with local files. HDFS always cre-



(a) Non-Trans



(b) OLTP (c) TPC-C

Figure 10: Experimental results with database apps.

ates/deletes large files (e.g., 128 MB) on the disk file system to efficiently handle large data sets and to leverage maximum I/O throughput from sequentially accessing these files.

This file management technique of HDFS is well-suited for NAND flash. In our observation, a large file is sequentially written across multiple flash blocks, and these flash blocks are invalidated together when the file is removed from HDFS. Therefore, FTL garbage collection is done by simply erasing flash blocks without any live page copies. Similar behavior is observed for all 3 storage configurations. This is the reason `RISA`, $\texttt{CISA}_{\text{EXT4}}$, and $\texttt{CISA}_{\text{F2FS}}$ all show good performance for Hadoop applications.

The results from all of these benchmarks clearly show that existing FTL-based storage is excessively over-designed. With the exception of error management modules (e.g., bad-block management), almost all storage management modules currently implemented in the storage device are not strictly necessary. `RISA` uses significantly less hardware resources, while maintaining, if not exceeding, the performance of `CISA` systems.

4.4 Lifetime Analysis

We analyze the lifetime of the flash storage for 10 different write workloads. We estimate expected flash lifetime using the number of block erasures performed by the workloads since NAND chips are rated for a limited number of program/erase cycles. As shown in Figure 12, `RISA` incurs noticeably fewer erase operations overall compared to $\texttt{CISA}_{\text{EXT4}}$ and $\texttt{CISA}_{\text{F2FS}}$. This lifetime benefit of `RISA` mainly comes from reduced number of copy operations during garbage collection. The only exception is `FIO(RW)` where `RISA` exhibits a higher WAF value than $\texttt{CISA}_{\text{EXT4}}$ as listed in Table 4.

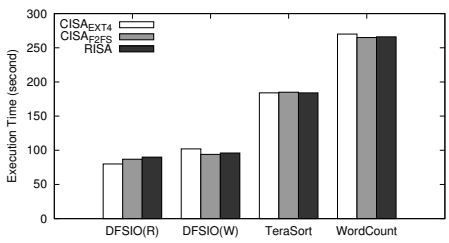


Figure 11: Experimental results with Hadoop apps.

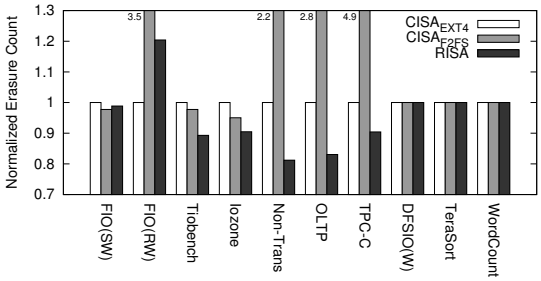


Figure 12: Erasure operations normalized to $\texttt{CISA}_{\text{EXT4}}$

4.5 Detailed Analysis

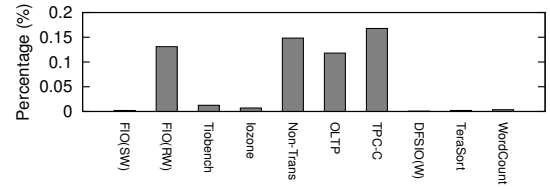
We also analyze the inode-map management overheads and the effect of a segment size on performance in RISA.

Inode-map Management Overheads: I/O operations required to manage inode-map segments in RFS are extra overheads that are not observed in the conventional LFS. For example, RFS writes TIMB blocks to inode-map segments whenever a check-point is written and performs additional garbage collection for inode-map segments.

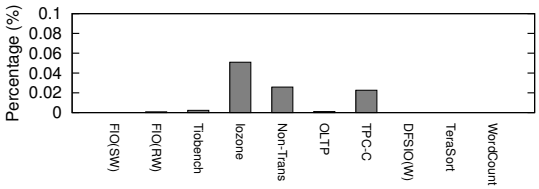
Figure 13(a) shows the percentage of TIMB writes to flash storage. We exclude read-only workloads. For all benchmarks, TIMB writes account for a very small proportion of the total writes. File-system check-pointing is not frequently invoked, so the number of writes to a check-point is very small compared to other writes. Moreover, the number of dirty TIMB blocks written together with a new check-point is small – only 2.6 TIMB blocks are written, on average, when a check-point is written.

Figure 13(b) illustrates how many extra copies occur for garbage collection in inode-map segments. Even though there are minor differences among the benchmarks, overall extra data copies for inode-map segments are insignificant compared to the total number of copies performed in the file system. This shows that inode-map garbage collection does not negatively affect I/O latencies and throughput.

Effect of Segment Size: In order to understand the effect of various segment sizes on performance, we measure extra I/Os incurred by the file-system garbage collector while changing a segment size from 4 MB to 128 MB. For a 4 MB segment, RSD is organized to have 4 channels and 2 ways; 8 512-KB flash blocks make up one segment. For a 128 MB segment, 16 channels and 16 ways are used; 256 512-KB flash blocks make up one segment. We conduct a set of experiments for $\texttt{CISA}_{\text{F2FS}}$ and $\texttt{CISA}_{\text{EXT4}}$ with different channels/ways organizations. Since our SSD platform has limited



(a) TIMB writes



(b) Inode-map garbage collection

Figure 13: Inode-map management overheads analysis

number of channels and ways, we used a DRAM-based SSD emulator for this experiment. The same file system and device driver are used.

Figure 14 plots the change of WAFs with different channels/ways under the `OLTP` workload. For $\texttt{CISA}_{\text{F2FS}}$, we show both F2FS and FTL WAF values independently on the graph. All WAF values are normalized to the 4 channels and 2 ways organization. As a segment size increases with more channels and ways, the value of WAF also increases in `RISA`. A similar trend is observed in $\texttt{CISA}_{\text{EXT4}}$ because the unit of FTL garbage collection becomes larger. For `RISA`, the WAF value does not significantly increase up to 8 channels and 8 ways (i.e., a segment size is 32 MB). However, beyond that, the value of WAF jumps when the segment size is larger than 64 MB (i.e. 16 x 8). As pointed out before, during garbage collection, the FTL can select the cheapest victim flash blocks in individual channels/ways, but `RISA` has to select the cheapest logical segment as victim (which is larger than a flash block) and copy all valid data to free locations. Therefore, it is inevitable that `RISA` incurs many more data copies than $\texttt{CISA}_{\text{EXT4}}$ when segment sizes become huge.

This problem can be addressed by using a smaller physical segment than the actual RSD organization. For example, a physical segment can be composed of flash blocks on the same way. If RSD has 16 channels and 16 ways, a physical segment is a group of 16 flash blocks on different channels belonging to the same way. A physical/logical segment size is thus reduced to 8 MB. RISA still exploits way-level I/O parallelism by writing data to logical segments sequentially (from $0th$ ways to $15th$ ways). Whenever segment cleaning is invoked, RISA should reclaim 16 free segments on different ways (i.e., total 128 MB) to fully exploit I/O parallelism when writing user data later. Since RISA has more freedom to select smaller victim segments with fewer valid data, overall segment cleaning costs are lowered.

`RISA(Impr)` in Figure 14 shows preliminary experimental results when RISA uses a smaller physical segment size. For 4 channels and 2 ways, a logical/physical segment size is set to 2 MB and, for 16 channels and 16 ways, a 8 MB segment is used. `RISA(Impr)` shows much smaller WAF values than `RISA` as expected. Currently, a physical segment size is the same as the group of blocks belonging to the same way, but its proper size should be carefully decided by considering I/O bandwidth, garbage collection efficiency and file-system design complexity. We do not study it in detail here because `RISA` performs well with relatively large segments (e.g., 32 MB). We will further investigate the effects of a small physical segment in several aspects.

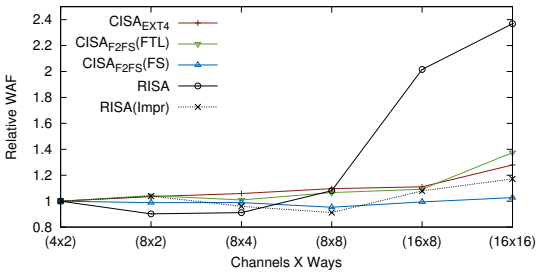


Figure 14: Effect of a segment size

Finally, $\texttt{CISA}_{\text{F2FS}}$ always uses the same segment size, so the file-system WAF values are kept around 1.0. However, the number of extra copies in the FTL increases with more channels/ways because of the increase of the FTL garbage collection unit. This shows that using a small/fixed segment is not always beneficial because of interference by the FTL.

5. Conclusion

In this paper, we proposed a new reduced I/O system architecture (RISA) to overcome the problems of existing complex I/O system architectures (CISA). By leveraging the architectural advantage of LFS, which was well-suited to the physical nature of NAND flash, the RISA file system directly handled the storage device using rich system-level information, removing the needs for logical-to-physical mapping and garbage collection on the storage side. The RISA storage device was designed to be as simple as possible, to only provide error-free storage access and high I/O throughput. Our evaluation showed that RISA performed much better than CISA with using significantly less resources.

In the near future, we plan to extend RISA for high-performance database systems. Many modern database systems manage storage space in an LFS-like manner. Thus, the data management techniques presented in RISA can be easily adapted to database systems using flash storage. This allows us to directly access flash storage, bypassing a deep I/O stack (the file system and the FTL), to achieve better I/O latency and high I/O throughput. RISA will also be extended to support efficient in-storage processing with reconfigurable hardware logic for data analytic applications. We expect that the simple architecture of RSD will enable us to easily integrate hardware accelerators into existing storage management functions in a straightforward manner, with minimum intervention from firmware.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, 2008.

[2] A. Ban. Flash file system, 1995. US Patent 5,404,485.

[3] M. Bjørling, J. Madsen, P. Bonnet, A. Zuck, Z. Bandic, and Q. Wang. Lightnvm: Lightning fast evaluation platform for non-volatile memories. In *Proceedings of Non-Volatile Memory Workshop*, 2014.

[4] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the Symposium on Applied Computing*, pages 1126–1130, 2007.

[5] Y.-B. Chang and L.-P. Chang. A self-balancing striping scheme for nand-flash storage systems. In *Proceedings of the Symposium on Applied Computing*, pages 1715–1719, 2008.

[6] M.-L. Chiang and R.-C. Chang. Cleaning policies in mobile computers using flash memory. *Journal of System Software*, 48(3):213–231, 1999.

[7] H. J. Choi, S.-H. Lim, and K. H. Park. Jftl: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4(4):14:1–14:22, 2009.

[8] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System software for flash memory: A survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 394–404, 2006.

[9] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2012.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2009.

[11] K. Ha and J. Kim. A program context-aware data separation technique for reducing garbage collection overhead in nand flash memory. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O*, 2011.

[12] S. Hahn, S. Lee, and J. Kim. Sos: Software-based out-of-order scheduling for high-performance nand flash-based ssds. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies*, pages 1–5, 2013.

[13] S. S. Hahn, J. Jeong, and J. Kim. To collect or not to collect: Just-in-time garbage collection for high-performance ssds with long lifetimes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implemenation (Poster)*, 2014.

[14] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl: Database systems on ftl-less flash storage. In *Proceedings of the VLDB Endowmen*, pages 1278–1281, 2013.

[15] Hitachi. Hitachi accelerated flash, 2015.

[16] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the International Workshop on Data Engineering*, pages 41–51, 2010.

[17] A. Hunter. A brief introduction to the design of ubifs, 2008.

[18] JEDEC. Onfi 4.0: Open nand flash interface 4.0 specification, 2014.

[19] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. Dfs: A file system for virtualized flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2010.

[20] M. Jung, E. H. Wilson, III, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 75:1–75:11, 2013.

[21] S.-M. Jung, J. Jang, W. Cho, H. Cho, J. Jeong, Y. Chang, J. Kim, Y. Rah, Y. Son, J. Park, M.-S. Song, K.-H. Kim, J.-S. Lim, and K. Kim. Three dimensionally stacked nand flash memory technology using stacking single crystal si layers on ild and tanos structure for beyond 30nm node. In *Proceedings of the International Electron Devices Meeting*, pages 1–4, 2006.

[22] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the International Conference on Embedded Software*, pages 161–170, 2006.

[23] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2014.

[24] Y. Kang, J. Yang, and E. L. Miller. Efficient storage management for object-based flash memory. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[25] H.-J. Kim and S.-G. Lee. A new flash memory management for flash storage system. In *Proceedings of the International Computer Software and Applications Conference*, 1999.

[26] 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*, 48(2): 366–375, 2002.

[27] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[28] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2015.

[29] S. Lee, J. Kim, and Arvind. Refactored design of i/o architecture for flash storage. *IEEE Computer Architecture Letters*, Preprint, 2014.

[30] 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 Embedded Computing Systems*, 6(3), 2007.

[31] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, and S. Maeng. Ossd: A case for object-based solid state drives. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies*, pages 1–13, 2013.

[32] C. Manning. Yaffs: Yet another flash file system, 2004.

[33] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 238–251, 1997.

[34] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2012.

[35] S. Mylavarapu, S. Choudhuri, A. Shrivastava, J. Lee, and T. Givargis. Fsaf: File system aware flash translation layer for nand flash memories. In *Proceedings of the Design Automation Test in Europe Conference*, pages 399–404, 2009.

[36] E. H. Nam, B. Kim, H. Eom, and S. L. Min. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, 60(5):653–666, 2011.

[37] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 301–311, 2011.

[38] Y. Pan, G. Dong, and T. Zhang. Error rate-based wear-leveling for nand flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(7):1350–1354, July 2013.

[39] S. Park and K. Shen. Fios: a fair, efficient flash i/o scheduler. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2012.

[40] Phison. Ps3110 controller, 2014.

[41] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1991.

[42] Samsung. Samsung ssd 840 evo data sheet, rev. 1.1, 2013.

[43] D. Woodhouse. Jffs2: The journalling flash file system, version 2, 2008.

[44] 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 USENIX Conference on File and Storage Technologies*, 2012.