Refactored Design of I/O Architecture for Flash Storage

Sungjin Lee, Jihong Kim, *Member, IEEE*, and Arvind, *Fellow, IEEE*

Abstract—Flash storage devices behave quite differently from hard disk drives (HDDs); a page on flash has to be erased before it can be rewritten, and the erasure has to be performed on a block which consists of a large number of contiguous pages. It is also important to distribute writes evenly among flash blocks to avoid premature wearing. To achieve interoperability with existing block I/O subsystems for HDDs, NAND flash devices employ an intermediate software layer, called the flash translation layer (FTL), which hides these differences. Unfortunately, FTL implementations require powerful processors with a large amount of DRAM in flash controllers and also incur many unnecessary I/O operations which degrade flash storage performance and lifetime. In this paper, we present a refactored design of I/O architecture for flash storage which dramatically increases storage performance and lifetime while decreasing the cost of the flash controller. In comparison with page-level FTL, our preliminary experiments show a reduction of 19 percent in I/O operations, improvement of I/O performance by 9 percent and storage lifetime by 36 percent. In addition, our scheme uses only $\frac{1}{128}$ DRAM memory in the flash controller.

Index Terms—Storage systems, file systems, NAND flash memory, I/O architectures

1 Introduction

It is well-known that the physical properties of NAND flash are different from those of HDDs. To provide interoperability with existing block I/O subsystems, NAND flash-based devices employ an intermediate software layer, called a flash translation layer (FTL) [7]. Though interoperability is highly desirable, the usefulness of FTL-based storage is continuously being questioned. In FTL-based storage, NAND flash is managed by two different software layers, a file system and FTL, each of which has a different design goal. This duplicate management incurs high inefficiency in terms of performance, lifetime, and cost. FTL has to maintain a huge mapping table in DRAM and requires powerful embedded processors (e.g., three CPUs w/ 1 GB DRAM [10]) to run complicated firmware algorithms, including logical-to-physical mapping and garbage collection [7]. The duplicate storage management by two different layers also incurs lots of extra I/Os, degrading storage performance and lifetime.

In order to overcome the inefficiency of FTL-based storage, several alternative software architectures have been proposed. A host-based FTL solution like DFS [4], [12], [14] mitigates these problems by moving some key functions of FTL to a host device driver. Supporting FTL functions in the host, however, cannot eliminate the duplicate management problem of flash storage because it simply changes the system software layer where FTL is running. Thus, the host-based FTL still wastes considerable host resources and incurs many extra I/Os.

Another alternative solution is to use flash file systems (FFS) like JFFS2 and YAFFS. Using NAND-specific interface layers (e.g., MTD), FFS directly manages flash blocks of raw NAND chips without any helps from FTL. However, FFS have serious limitations for use in recent flash devices like SSDs and eMMCs. The internal architecture (e.g., channel organizations and I/O interleaving) of flash devices is both quite complex and different for each storage vendor [7]. Storage vendors are also reluctant to divulge the internal architecture of their devices and prefer hiding all those details behind the block I/O interface. FTL, provided by the storage vendor, performs internal storage management using rich proprietary information. In practice, FFS cannot work with most flash devices and is only used in limited embedded systems with few raw NAND chips.

In this paper, we propose a *RE*factored *D*esign of I/*O* architecture, called *REDO*, which solves the duplicate management problem while preserving the advantages of FTL-based storage. REDO refactors two main components of the I/O subsystem—the file system and the storage device. REDO removes logical-to-physical mapping and garbage collection from the storage device. Instead, a refactored file system (RFS) directly manages the storage address space, including the garbage collection. Unlike host-based FTL, all those functions are conducted by RFS without any helps from an intermediate host layer like a device driver. This eliminates the need for maintaining a large logical-to-physical page-map table, allowing us to perform garbage collection more efficiently at the file system level. A refactored storage device controller (RSD) becomes simpler because it runs a small number of essential flash management functions. RSD maintains a much smaller logical-to-physical segment-map table to manage wear-leveling and bad blocks. Unlike FFS, REDO provides interoperability with block I/O subsystems, allowing SSD vendors to hide all the details of their devices and NAND characteristics.

We have implemented RFS as a new file system in the Linux kernel and tested it using a flash storage device emulator for RSD. Our preliminary experiments show that REDO eliminates the well-known tradeoff between performance and cost in designing flash storage—it shows better I/O performance with smaller hardware resources than page-level FTL.

2 Refactored I/O Architecture for Flash

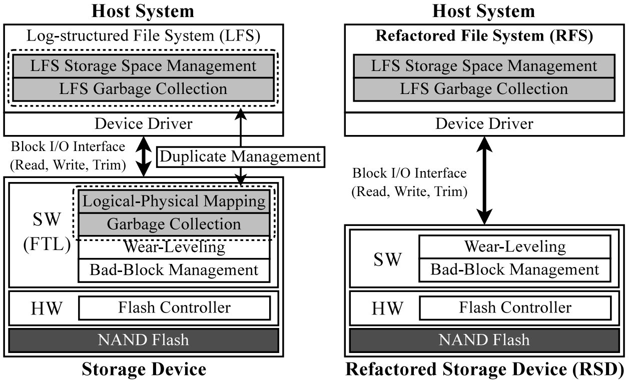
REDO is based on a log-structured file system (LFS) and, in the remainder of this paper, we assume LFS as a baseline file system. This is reasonable because many new file systems for SSDs are based on LFS. Usually, LFS works better than traditional file systems like EXT4, so it is regarded as a better file system solution to SSDs.

Fig. 1a shows the architecture of the FTL-based storage with LFS. Typical LFS (e.g., Sprite LFS [6], NILFS [9], and F2FS [3]) manages a logical address space as one huge log and FTL also manages a physical address space in an LFS-like manner with a logical-to-physical mapping table. LFS and FTL also run their own garbage collection algorithms. The same NAND flash is thus *doubly* managed by two different layers in a similar manner. REDO eliminates this duplicate storage management. As depicted in Fig. 1b, the storage space management and garbage collection modules of RFS directly manage NAND flash, removing logical-to-physical mapping and garbage collection from a storage device. Only simple wear-leveling and bad-blocks management remain in RSD. As the reliability of NAND substrate continuously degrades, those lifetime management functions can be effectively supported by the storage controller where detailed physical information for NAND devices is available. REDO maintains compatibility with the existing block I/O interface, enabling us to use existing block I/O subsystems. It is important to notice that the interface between the RFS and RSD does not interfere with exploiting device-level parallelism.

Nomenclature. We will use the following nomenclature for a page, a block and a segment in this paper. A file usually consists of 4 KB blocks, which we will refer to as *pages*. Flash storage typically

- *S. Lee and A. Mithal are with the Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: chamdoo@gmail.com, arvind@csail.mit.edu.*
- *J. Kim is with the School of Computer Science and Engineering, Seoul National University, Seoul, Korea. E-mail: jihong@davinci.snu.ac.kr.*

Manuscript received 20 Feb. 2014; revised 27 May 2014; accepted 3 June 2014. Date of publication 23 June 2014; date of current version 19 June 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2014.2329423



(a) FTL-based Storage (b) REDO

Fig. 1. A comparison between FTL-based storage and REDO.

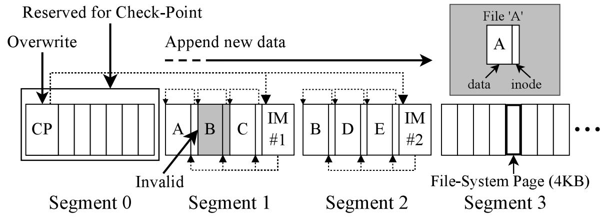


Fig. 2. An example of how LFS handles write requests. Five files are written to free space along with their inodes in the following order: `A`, `B`, `C`, `B`, `D`, and `E`. The file `B` is written twice. The pieces of the inode map, `IM#1` and `IM#2`, which point to the locations of inodes, are written together. The check-point `CP` is overwritten in the fixed location of the segment 0 which is reserved for the check-point.

consists of 4 KB flash pages and much larger *blocks* (typically 64 to 128 pages). In LFS, storage management is often done in much larger chunks, known as *logical segments* or simply *segments*. Typically, a segment is 2 MB which is two to four times larger than a flash block.

2.1 Duplicate Storage Management Problem

To explain the problems associated with the duplicate storage management, we begin with a description of LFS (Fig. 2). Our explanation is based on Sprite LFS [6] because it is well-known and many LFSs follow its design concept. The problem of Sprite LFS is also observed in modern LFS like NILFS [9] and F2FS [3]. LFS first buffers file data and inodes in DRAM and it periodically performs an out-place update in segment size chunks to HDD or SSD. LFS needs to maintain an inode map which indicates the locations of inodes scattered across the storage space. An out-place update by its very nature invalidates some old pages in the file system which causes changes in the inode map. For crash recovery, LFS also maintains a check-point which points to pieces of the inode map. LFS stores the check-point in a fixed location for fast construction of the file system at mount time. Otherwise, LFS would have to scan the entire storage space to build the file system. The check-point is typically updated every 30 seconds or when an explicit sync command is issued. Once all the free space is exhausted, LFS performs garbage collection to reclaim free space.

Even though the overall architecture of LFS is well suited to the physical natures of NAND flash, the plain combination of LFS and FTL works inefficiently. Using the write request sequence used in Fig. 2, we show the problems that arise with such a simple combination in Fig. 3. We assume that the segment size is twice the flash block size, and the file-system page size is the same as the flash page size.

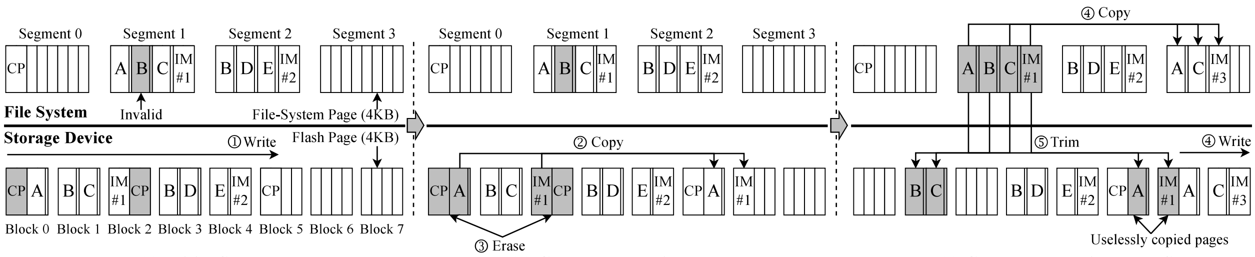
Using a logical-to-physical mapping table, FTL writes the incoming data to free space in a similar way that LFS does, overcoming the limitation of NAND flash to perform in-place updates. In Fig. 3a, the check-point `CP` is initially written and then is updated twice after the pieces of the inode map, `IM#1` and `IM#2`, are written. Since the check-point is updated in the fixed location with the same logical address, FTL writes the new check-point to the free space of NAND flash while invalidating the previous version. In LFS with FTL, both FTL and LFS perform garbage collection with their own policies. Suppose that FTL is the first one to trigger garbage collection. In Fig. 3b, the `blocks 0` and `2` are chosen as a victim and four valid pages for the file `A` and `IM#1` are copied to free blocks. The victim blocks are then erased. LFS must trigger garbage collection whenever the file system runs out of space. In Fig. 3c, the files `A` and `C` are copied to the free space of a file system, and `IM#1` is updated to `IM#3` to indicate the new locations of `A` and `C`. LFS finally informs the storage device that the victim segment has become garbage by issuing a TRIM command. Notice that the movement of the pages for `A` and `IM#1` by the FTL garbage collection turns out to be useless.

Useless page copies do not occur in traditional file systems like EXT4 and NTFS. However, such systems cause an inordinate amount of in-place updates which, in turn, triggers much more garbage collection at the flash device level. This is confirmed by our empirical results in Section 3 where for four out of five benchmarks REDO outperforms other systems.

2.2 Refactored File System

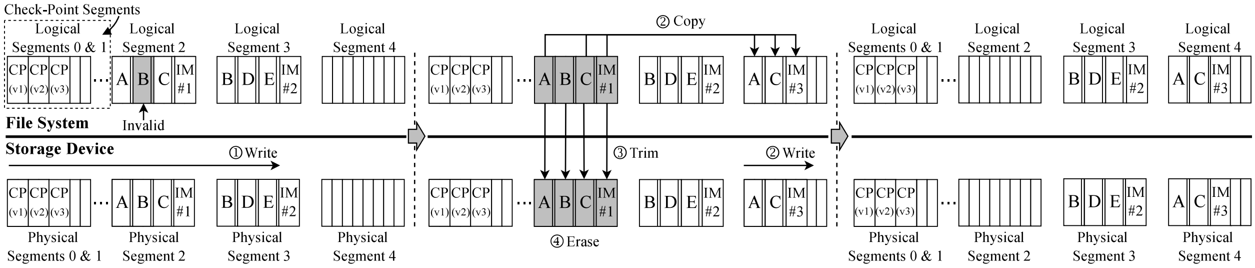
To solve the problem of the duplicate storage management, RFS is designed differently from the conventional LFS in two ways; it only issues out-place update commands and informs a storage device about which blocks have become erasable via TRIM commands. This frees the flash controller from the task of garbage collection all together. The question of unnecessary copies in the flash storage never arises in REDO.

A logical segment in RFS corresponds directly to a "*physical segment*", which is the group of flash blocks. This eliminates the need of logical-to-physical page-map table, enabling us to access NAND flash directly. In the next section, we will see that RSD maintains a segment-map table that maps segment addresses supplied by RFS to real physical segments. This remapping requires



(a) Initial State **(b) Garbage collection at FTL** **(c) Garbage collection at LFS**

Fig. 3. An example of how LFS and FTL manage NAND flash. The file `A` and `IM#1` are copied unnecessarily.



(a) Initial State **(b) Garbage collection at RFS** **(c) After garbage collection**

Fig. 4. An example of how RFS in REDO manages NAND flash.

much smaller tables and is for bad-block management and wear-leveling only.

Fig. 4 shows how RFS manages NAND flash when the write requests shown in Fig. 2 are issued. As depicted in Fig. 4 a, RFS writes file data, inodes, and the pieces of the inode map in an out-place update manner. Unlike LFS, the incoming data are written to a physical segment corresponding to a logical segment, and their relative offsets in the logical segment are preserved in the physical one. For check-pointing, RFS reserves two *fixed* logical segments, called check-point segments. In Fig. 4, the `logical segments 0` and `1` are check-point segments (the `segment 1` is not shown). RFS then appends new check-points with different version numbers, so that the overwrites never happen. RFS manages all the obsolete data at the level of a file system and triggers garbage collection when free space is exhausted. In Fig. 4b, RFS chooses the `logical segment 2` as a victim and copies live data to free space. The victim segment becomes free for future use. To inform that the physical segment for the victim has obsolete data, RFS delivers a TRIM command to RSD. Finally, RSD marks the physical segment out-of-date and erases flash blocks.

At mount time, RFS finds a check-point with the latest version number by scanning two check-point segments and then uses it to build a file system. Considering that the segment size is several megabytes, the time taken to find the latest check-point is negligible. If the free space in two segments is exhausted, RFS reclaims a free segment by garbage collection. The garbage collection for check-point segments is simple. The latest version of a check-point is always stored in one of two check-point segments. If one segment has the latest check-point, the other segment has only the old versions. RFS frees this old segment and issues a TRIM command to inform the storage device that the corresponding physical segment has obsolete data. Then, RFS reuses the freed logical segment to write new check-points. Even if RFS uses two logical segments repeatedly, it does not imply a rapid wear-out of those segments because the storage device manages the wear-leveling.

Note that RFS does not change the existing storage space management and garbage collection modules of LFS. It merely modifies the check-point management module to avoid in-place updates. This simplicity enables RFS to be easily adapted to other LFSs. The crash recovery of RFS is also not changed greatly; in fact, the recovery process is exactly the same as that of LFS except for finding the latest check-point.

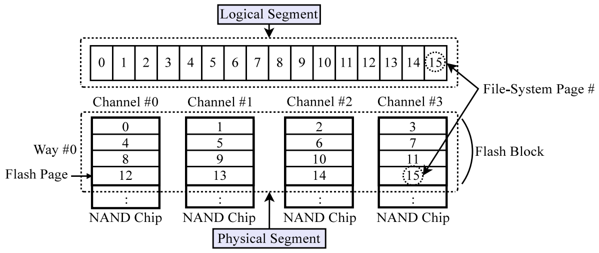


Fig. 5. Logical and physical segment mapping.

2.3 Refactored Storage Device

The direct storage management of RFS greatly simplifies the architecture of RSD, lowering the cost of building a storage device close to that of SSDs with simple block-level FTL. The flash blocks belonging to a physical segment are mapped to different channels and ways that can be operated in parallel. File-system pages and flash pages in the logical and physical segments are statically mapped to maximize device-level parallelism. Fig. 5 shows an example of how a logical segment is mapped to a physical segment when there are four channels and one way. The number of file-system pages per logical segment is assumed to be 16. RFS transfers the bulk of data to RSD in the ascending order of their logical addresses after buffering them in DRAM. Thus, this simple static mapping can maximally exploit device-level I/O parallelism.

The handling of write requests in RSD is depicted in Fig. 6. RSD maintains the segment-map table, and each entry of the table points to physical blocks that are mapped to a logical segment. When write requests come, RSD calculates a logical segment number (i.e., 100) using the logical file-system page number (i.e., 1,600). Then, it looks up the remapping table to find the physical blocks mapped to the logical segment. If physical blocks are not mapped yet, RSD builds the physical segment by allocating new flash blocks. RSD picks up free blocks with the smallest P/E cycles in the corresponding channel/way. A bad block is ignored. If there are flash blocks already mapped, RSD writes the data to the fixed location in the physical segment as depicted in Fig. 5. Block erasure commands are not explicitly issued from RFS. But, RSD easily figures out which blocks are out-of-date and are ready for erasure because RFS informs RSD of physical segments only with obsolete data via a TRIM command. RSD handles overwrites like block-level FTL. This is inefficient, but since RFS only issues out-place updates, it works efficiently with RFS.

RSD has a simple architecture compared with other flash devices. Thus, we expect that existing flash devices can be easily modified to support RSD. It is also worth noting the differences between REDO and nameless write [15]. The nameless write removes duplicate storage management by moving file-system functions to a storage device. This makes a storage device more complicated, and, furthermore, requires many custom I/O interfaces for the interactions between the host and the device. REDO not only reduces

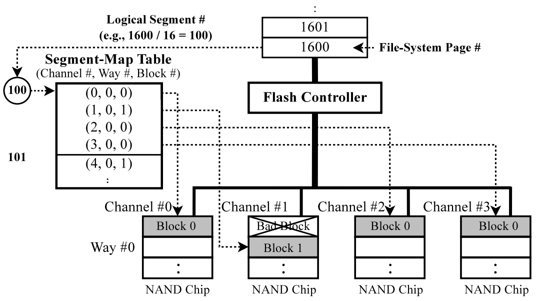


Fig. 6. Handling of write requests in RSD.

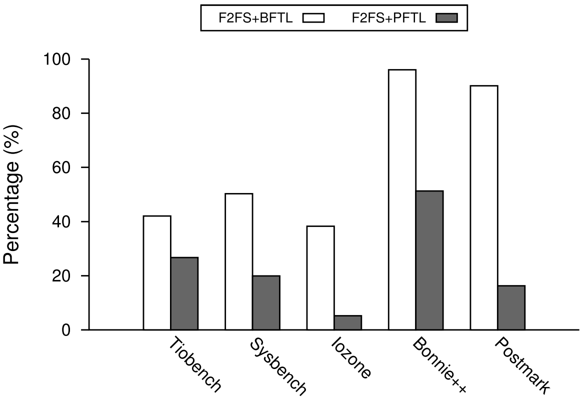


Fig. 7. Reduction in I/O operations.

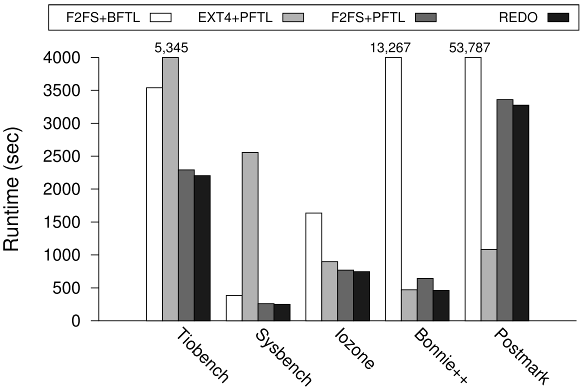


Fig. 8. Runtime (second).

complexity of a flash controller, but also does not require any custom interfaces.

3 Experiments

We implemented RFS in F2FS [3]. RSD was implemented in an SSD emulator, called FlashBench [11], which emulated the array of NAND flash using host DRAM. FlashBench also ran several firmware algorithms, including address mapping and garbage collection. FlashBench was organized with two channels and two ways. A flash block consisted of 4 KB 128 pages. The total SSD capacity was 512 MB.

We compared REDO with F2FS running on top of two different FTL designs: block-level and page-level FTLs, which are denoted by F2FS+BFTL and F2FS+PFTL. RFS in REDO was based on F2FS, so RFS used the exactly same module as F2FS for address management and garbage collection. To eliminate in-place updates, we modified the check-point module of F2FS. Since F2FS allowed in-place updates to the inode map, we modified F2FS so that it wrote the inode map in an out-place-update manner. For page-level FTL, a greedy policy was used [7]. We also compared REDO with EXT4 running with page-level FTL, which is denoted by EXT4+PFTL.

For our evaluation, we used five benchmarks: Postmark [2], Bonnie++ [8], Tiobench [5], Sysbench [1], and Iozone [13], and set parameter values so that the maximum file size created did not exceed the storage capacity. The benchmarks were configured to generate sufficient I/Os for meaningful evaluation. For other parameters, default values were used.

Fig. 7 shows the impact of the elimination of useless copies on the reduction of I/O operations. Since useless page copies do not occur in EXT4+PFTL, we compare REDO with F2FS+BFTL and F2FS+PFTL. On average, REDO reduces the number of I/O operations by 61 and 19 percent over F2FS+BFTL and F2FS+PFTL. The benefit of eliminating useless copies increases in proportion to garbage collection overheads at the FTL. Bonnie++ exhibits high garbage collection overheads, incurring lots of page copies. By eliminating unnecessary copies, REDO reduces the number of I/O operations by 51 percent over F2FS+PFTL. For Iozone with low garbage collection overheads, the number of I/O operations decreases by 5 percent.

TABLE 1
The Number of Block Erasure Operations

Benchmark	F2FS+BFTL	EXT4+PFTL	F2FS+PFTL	REDO
Tiobench	115,168	478,561	40,861	21,564
Sysbench	7,085	27,742	5,318	3,197
Iozone	18,325	9,686	10,602	7,900
Bonnie++	1,237,200	4,275	6,662	3,776
Postmark	2,011,481	16,342	46,426	37,085

TABLE 2
The Size of a Mapping Table

Capacity	F2FS+BFTL	EXT4+PFTL	F2FS+PFTL	REDO
512 MB	4 KB	512 KB	512 KB	4 KB
1 TB	8 MB	1 GB	1 GB	8 MB

Fig. 8 shows the runtime of F2FS+BFTL, EXT4+PFTL, F2FS+PFTL, and REDO. REDO reduces the runtime by 50.1, 40.1, and 9.2 percent over F2FS+BFTL, EXT4+PFTL, and F2FS+PFTL. Except for F2FS+BFTL, for Tiobench, Sysbench, and Iozone, EXT4+PFTL shows the worst performance because of many extra I/Os at the FTL level. In case of Bonnie++, EXT4+PFTL works better than F2FS+PFTL because of higher garbage collection overheads of F2FS+PFTL at the FTL. By removing useless copies, REDO reduces the runtime by 3 and 28 percent over EXT4+PFTL and F2FS+PFTL. For Postmark, both F2FS+PFTL and REDO perform worse than EXT4+PFTL. Postmark is a small-file-oriented benchmark. F2FS does not efficiently handle lots of small files, issuing 1.9× more I/O requests to the SSD over EXT4+PFTL, which in turn increases the overall runtime of F2FS+PFTL and REDO.

Table 1 shows that REDO reduces the number of block erasures by 36, 62, and 94 percent over F2FS+PFTL, EXT4+PFTL, and F2FS+BFTL. This implies that REDO improves the SSD lifetime by the same amount. Table 2 lists the mapping table sizes. When the SSD capacity is 512 MB, the mapping table sizes for F2FS+BFTL, EXT4+PFTL, F2FS+PFTL, and REDO are 4, 512, 512, and 4 KB, respectively. If the SSD capacity is 1 TB, the mapping table increases to 1 GB in EXT4+PFTL and F2FS+PFTL. Even if only a small fraction of DRAM is required (which is the same as F2FS+BFTL), REDO outperforms F2FS+PFTL and EXT4+PFTL.

In conclusion, our results showed that REDO eliminated the tradeoff between performance and cost in designing SSDs, realizing a high-performance and low-cost storage solution.

4 Conclusion

We presented a new I/O architecture, called REDO, for flash-based SSDs. Experimental results showed that refactoring the file system and the storage device software was very effective—reducing the number of I/O operations by 19 percent while improving the storage performance and lifetime by 9 and 36 percent, respectively, and simultaneously lowering the amount of DRAM by $\frac{1}{128}$ over page-level FTL with F2FS.

Acknowledgments

This research was supported by the National Research Foundation (NRF) of Korea (NRF-2013R1A6A3A03063762, NRF-2013R1A2A2A01068260, No. 2010-0020724). At MIT, CSAIL Samsung Electronics (#692509) supported this work.

References

[1] A. Kopytov, "SysBench: A system performance benchmark," http://sysbench.sourceforge.net/, 2004.

[2] J. Katcher, "PostMark: A new filesystem benchmark," NetApp, Inc., Sunnyvale, CA, USA, Tech. Rep. TR3022, 1997.

[3] J. Kim, "F2FS: Introduce flash-friendly file system," [Online]. Available: http://lwn.net/Articles/518718/, 2012.

[4] M. Jung, E.-H. Wilson, 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. pp. 75:1–75:11.

[5] M. Kuoppala, "Tiobench: Threaded I/O Bench for Linux," [Online]. Available: http://sourceforge.net/projects/tiobench/, 2002.

[6] M. Rosenblum and J.-K. Ousterhout, "The Design and implementation of a log-structured file system," in *Proc. 13th ACM Symp. Operating Syst. Principles*, 1991, pp. 1–15.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J.-D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.

[8] R. Coker, "The Bonnie++ benchmark," [Online]. Available: http://www.coker.com.au/bonnie++/, 2001.

[9] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The Linux implementation of a log-structured file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, 2006.

[10] Samsung Corp., "Samsung SSD 840 EVO data sheet, Rev. 1.1," [Online]. Available: http://samsung.also.ch/fileadmin/Dateien/pdf/Samsung_SSD_840_EVO_mSATA_ Data_Sheet_Rev_1.0.pdf, 2013.

[11] S. Lee, J. Park, and J. Kim, "FlashBench: A workbench for a rapid development of flash-based storage devices," in *Proc. Int. Symp. Rapid Syst. Prototyping*, 2012, pp.163–169.

[12] W.-K. Josephson, L.-A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," in *Proc. USENIX Conf. File Storage Technol.*, 2010, pp. 85–100.

[13] D. Capps and W. Norcott, "Iozone filesystem benchmark," [Online]. Available: http://www.iozone.org/, 2003.

[14] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D.-K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 301–311.

[15] Y. Zhang, L.-P. Arulraj, A.-C. Arpaci-Dusseau, R.-H. Arpaci-Dusseau, "De-indirection for flash-based SSDs with nameless writes," in *Proc. USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.