Improving SSD Lifetime with Byte-Addressable Metadata

Yanqin Jin*
University of California, San Diego
y7jin@cs.ucsd.edu

Hung-Wei Tseng
North Carolina State University
hungwei_tseng@ncsu.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

ABSTRACT

Existing solid state drives (SSDs) provide flash-based out-of-band (OOB) data that can only be updated on a page write. Consequently, the metadata stored in their OOB region lack flexibility due to the idiosyncrasies of flash memory, incurring unnecessary flash write operations detrimental to device lifetime.

We propose PebbleSSD, an SSD with *byte-addressable metadata*, or BAM, as a mechanism exploiting the non-volatile, byte-addressable random access memory (NVRAM) inside the SSD. With BAM, PebbleSSD can support a range of useful features to improve its lifetime by reducing redundant flash writes. Specifically, PebbleSSD supports a write-optimized, BAM-based file block mapping to prevent excessive updates of file system index blocks. Furthermore, PebbleSSD allows log-structured file systems to perform fast and efficient log cleaning with minimal flash writes.

We have implemented a prototype of PebbleSSD on a commercial SSD development platform, and experimental results demonstrate that PebbleSSD can reduce the amount of data written by log-structured file systems during log cleaning by up to 99%, and PebbleSSD's BAM-based file block mapping can reduce flash writes by up to 33% for a number of workloads.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Software and its engineering → File systems management**; • **Computer systems organization →** *Heterogeneous (hybrid) systems*;

KEYWORDS

Flash memory, solid state drive, file system

1 INTRODUCTION

Due to the idiosyncrasies of flash, solid state drives (SSDs) implement complex flash translation layers (FTLs) to hide the details of flash, including its limited lifetime. To support this, NAND flash devices provide out-of-band (OOB) regions on each flash page that allows FTLs to store metadata. Storing the metadata in flash limits its utility since the OOB region suffers from the same idiosyncrasies as the primary "in-band" data.

Emerging non-volatile byte-addressable memories avoid the idiosyncrasies of flash memory and will eventually enable very fast SSDs. However, in the near future, the cost-per-bit of these non-volatile memories still cannot compete with flash. A more economical alternative is to use non-volatile, byte-addressable memories to store FTL metadata (i.e. the OOB data). Such a change would come at a modest price but could dramatically increase the flexibility of FTLs and enable a wide range of useful features.

Based on these observations, we have proposed *PebbleSSD* that allows us to explore the implications and applications of *byte-addressable metadata (BAM)*. PebbleSSD provides two additional features to modern SSDs. These features improve the efficiency of log-structured file systems.

The first new feature of PebbleSSD is the use of BAM to store the logical address to which each physical page has been mapped by the FTL. This BAM-based mapping is the inverse of the normal logical-to-physical address mapping that most FTLs already store in the in-SSD DRAM. Some existing SSDs also store this physical-to-logical mapping in the flash-based per-page OOB region. The second new feature of PebbleSSD is using BAM to temporarily store file system metadata.

To support these features, PebbleSSD provides two new commands. The first command is `remap` that allows the FTL to dynamically change the bi-directional mapping between logical and physical addresses, achieving flexible and fast movement of data in the logical address space without having to write to flash.

`remap` makes log cleaning more efficient. Original log-structured file systems have to read valid data from their previous locations and write them to new destinations. In contrast, our custom log-structured file systems running on PebbleSSD can "move" data without writing to flash, thus effectively reducing the amount of data written during log cleaning.

The second command is `fs_write` that persists not only file data blocks[1] in flash memory, but also their offset in the file and file inode number in BAM.

`fs_write` allows for more efficient flushing of file data blocks. This allows log-structured file systems to avoid the so-called "wandering tree problem" in which the file systems write data and index blocks to newly allocated space, thus writing a data block can cause

*Currently with Alluxio (alluxio.com).

MEMSYS 2017, October 2–5, 2017, Alexandria, VA, USA

ACM ISBN 978-1-4503-5335-9/17/10…$15.00
https://doi.org/10.1145/3132402.3132420

[1] Throughout this paper, we refer to the granularity of flash erase operation as "flash erase block", "flash block" or "erase block", while "block", "data block", "node block", "file block" etc. refer to the basic data unit in file systems.

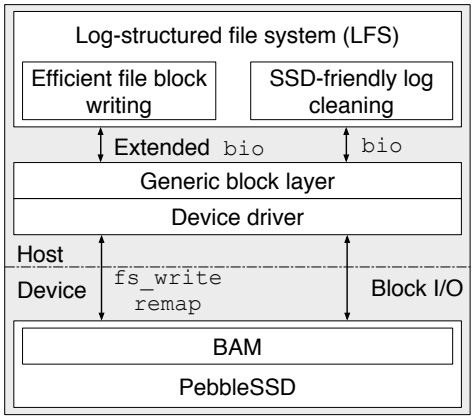

Figure 1: System architecture They system consists of the following components: PebbleSSD, log-structured file system, and block layer and device driver.

recursive flushing of its ancestor index blocks in the tree-based block mapping [15]. With `fs_write`, log-structured file systems write only file data blocks during `fsync`. The file systems do not have to write the index blocks during `fsync` since the information in BAM written by `fs_write` is already sufficient to guarantee the recoverability of file. Therefore BAM allows log-structured file system to write less data to flash, thus improving the efficiency and lifetime of PebbleSSD.

We implement PebbleSSD on a commercial, flash-based SSD reference design. We modify NILFS2 [10] and F2FS [24] to demonstrate the benefits of PebbleSSD. The experimental results show that PebbleSSD can reduce the amount of data written by log-structured file systems during log cleaning by up to 99% with the `remap` command. PebbleSSD's write-optimized file block mapping and `fs_write` reduce the flash write by up to 33% for a wide range of workloads.

The rest of the paper is organized as follows. Section 2 provides background of PebbleSSD. Section 3 presents an overview of the system. Section 4 and Section 5 describes the components of the system in detail. Section 6 presents our experimental results. Section 7 places this work in the context of existing research, while Section 8 concludes.

2 BACKGROUND AND MOTIVATION

PebbleSSD combines conventional flash memory, which is inexpensive, prone to wear-out, and idiosyncratic, with byte-addressable NVM that has lower latency, a cleaner interface, and better reliability. Combining these two types of memories allows PebbleSSD to significantly improve the reliability and efficiency of log-structured file systems at moderate cost.

Below, we briefly survey the characteristics of NAND flash, byte addressable NVM, and log-structured file systems.

2.1 Flash memory and SSD

The characteristics of flash memory has profound influence on SSD design and implementation. First, flash memory supports three basic operations, i.e. read, program and erase that operate on different granularities with different latencies. Specifically, read and program operate on 4–8 KB pages and have latencies of less than 100 μs and several hundred microseconds respectively depending on underlying flash devices. In contrast, erase operate on an entire flash block which typically consists of between 64 and 512 pages and takes several milliseconds to complete.

Since pages are immutable once programmed until the next erase operation, and naive erase-before-program approach to in-place update is costly and harmful to SSD lifetime, SSDs utilize the flash translation layer (FTL) to emulate the block interface. The FTL maps logical block addresses (LBAs) to physical page numbers (PPNs), and directs incoming write requests evenly to erased pages. When all the pages in a flash block are programmed, the firmware inserts the block to a list for future erase operations.

Second, each flash block can endure only a limited number of program/erase (P/E) cycles before it becomes unreliable. Therefore, one of the most critical design goals of SSDs is to maximize their lifetime by making flash memory blocks wear slowly and evenly.

Flash pages include an OOB region in addition to the "in-band" 4 or 8 KB region. The FTL stores per-page metadata in the OOB region. This can include the LBA to which the physical page has been mapped. In this way, the FTL maintains an inverse mapping from physical to logical address space. Each time the host issues a write request to an LBA, the FTL writes data to an erased page and stores the LBA in the page's OOB region. To modify the OOB, the FTL has to write the content of a physical page to another location.

SSDs have to perform garbage collection to prepare erased flash blocks for incoming write requests. During garbage collection, the firmware sequentially scans the pages in a flash block and identifies which pages contain valid data. After copying valid pages to their new locations, the firmware can safely erase the entire flash block.

2.2 Log-structured file system

A log-structured file system [10, 24, 33] organizes its data in one or multiple append-only logs, a natural fit for flash-based devices. Due to the large-scale deployment of SSDs, recent flash-friendly file systems [24] have attracted close attention, yet they suffer from two limitations.

First, log-structured file systems have to perform host log cleaning to create free *segments* in the logical address space for incoming write requests from user space applications. Log cleaning is similar to, but independent from the SSD's internal garbage collection[2]. Current SSDs confine the log-structured file system to block-based I/O interface. The semantic gap and lack of coordination between SSD garbage collection and file system log cleaning can lead to inefficiencies. Due to their independent data movement, valid data may be written multiple times by each of them, consuming erased flash pages rapidly.

Second, log-structured file systems suffer from the "wandering tree problem" [15]. This occurs because log-structured file systems must perform out-of-place updates. A write to one data block in a file can cause a cascade of writes as the file systems update the

[2]In some context, log cleaning is also called "garbage collection" of log-structured file systems. In this paper, for the purpose of simplicity and clarification, we use "log cleaning" in the context of log-structured file system, and "garbage collection" in the context of SSDs.

pointer to that data block, the pointer to the direct node block that contains the pointer, the pointer to the indirect block that points to the direct node block, so on.

3 SYSTEM OVERVIEW

PebbleSSD has a heterogeneous architecture using flash memory as primary data storage and NVRAM to store BAM in addition to metadata in the flash's OOB.

Figure 1 illustrates the relationship between PebbleSSD and other components of the system. This section describes system components while Section 4 and Section 5 present their implementation in more detail.

3.1 PebbleSSD interface

PebbleSSD uses the BAM to enable a new interface that allows applications to manage their data more efficiently. Table 1 summarizes the commands included in this interface.

First, PebbleSSD stores the physical-to-logical address mapping in BAM. This is different from conventional designs that store such mapping in flash-based OOB region. The `remap` command can dynamically change both the normal LBA-to-PPN mapping and the BAM-based PPN-to-LBA mapping.

Second, the `fs_write` command not only writes data to flash memory, but also updates their corresponding metadata in BAM. This can be useful to file systems because they can avoid having to issue multiple `write` commands to flash memory. For example, file systems can issue a single `fs_write` command to write file data blocks as well as update the pointers pointing to them.

Third, PebbleSSD supports conventional `write`, `read` and `trim` commands to maintain compatibility.

3.2 PebbleSSD applications

System can use PebbleSSD's new commands in a variety of ways. We focus on two optimizations especially suited to log-structured file systems: using `remap` to reduce the cost of log cleaning and using `fs_write` to improve the efficiency of writing data blocks.

Log-structured file systems have to move clean, valid data from their original logical segments to newly allocated ones during log cleaning. To reduce data movement overhead, log-structured file systems prefer to select old segments for cleaning. Therefore, there is a high chance that the pages in this segment are already clean and persistent in flash memory.

Log structured file systems can use PebbleSSD's `remap` command to reduce the cost of moving clean data to new log segments. Rather than copying the data using `read` and `write` commands, the file system can `remap` the clean data into the new segment. Figure 2 shows this operation in action.

Log structured file system can use `fs_write` to avoid the wandering tree problem described in Section 2.2. The file system can use `fs_write` to store file inode numbers and data blocks' offsets within their files in the BAM, allowing the file system to defer the flushing of node blocks in the conventional tree-based block mapping index until next checkpoint.

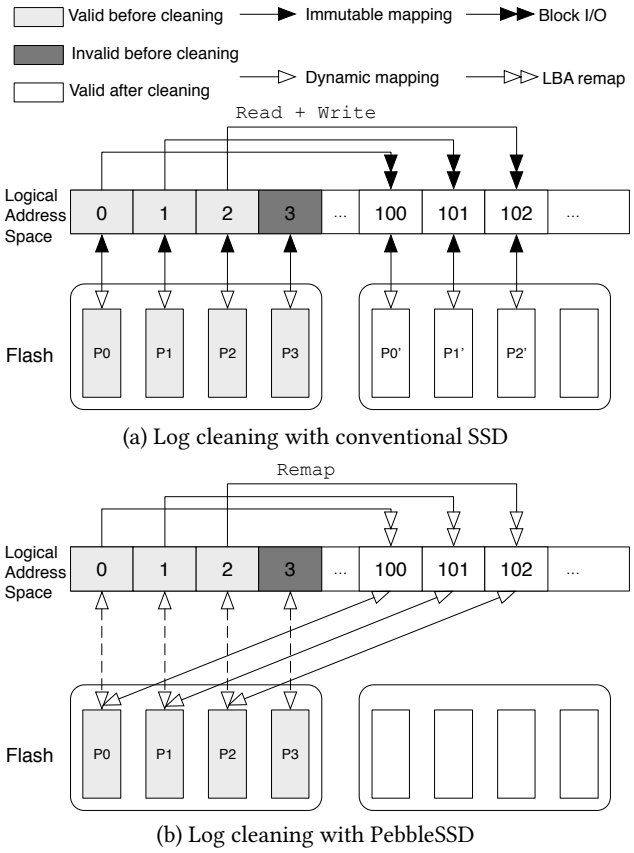

Figure 2: Log cleaning The `remap` command allows log-structured file systems to remap physical pages to new LBAs without writing flash memory.

4 PEBBLESSD

We have built PebbleSSD on a commercial SSD development board. Our implementation includes a customized firmware, a custom driver and modified Linux block layer to support BAM. The PebbleSSD firmware manages the flash memory and BAM area. The modified PebbleSSD block driver supports an extended interface that includes `remap` and `fs_write`. The block layer invokes the device driver to issue low-level commands to PebbleSSD. In this section we describe in detail the core features of PebbleSSD focusing on `remap` and `fs_write` commands.

4.1 Hardware architecture

Our commercial NVMe [11] SSD development platform connects to the host machine via four-lane PCIe 3.0. It comprises an array of SLC flash chips (375 GB in total) organized in 16 channels. The channels are connected to a multi-core controller. PebbleSSD firmware runs on the controller to manage flash memory and provide support for its commands.

Command	Description
`read`(startLBA, num)	Read num sectors from startLBA
`write`(startLBA, num)	Write num sectors from startLBA
`trim`(startLBA, num)	Mark num sectors starting from startLBA as invalid
`fs_write`(startLBA, num, file, offstInFile)	Write num sectors starting from startLBA for a file.
`remap`(srcLBA, dstLBA, num)	Move num sectors starting from srcLBA to dstLBA.

Table 1: PebbleSSD commands The new `fs_write` and `remap` commands allow applications e.g. file systems to reduce the amount of data written to flash.

Figure 3 shows the hardware architecture of PebbleSSD. The most important components are the embedded cores, the flash interface and the in-SSD DRAM.

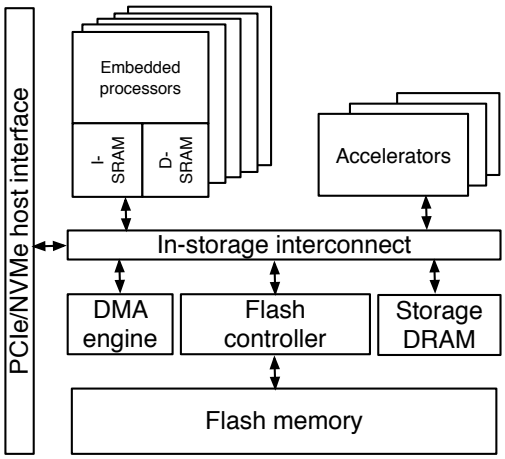

Figure 3: PebbleSSD architecture PebbleSSD exposes its internal DRAM and processing power to the host via PCIe/NVMe interface.

The SSD controller comprises multiple embedded processors running firmware at 500 MHz. Each of these cores has 64 KB private instruction and data memories. An on-chip network connects the cores, the flash interface and the DRAM so that they can communicate with each other. The embedded cores issue commands to the flash interface which reads or writes data from/to a buffer in the PebbleSSD's internal DRAM. The cores have to initiate explicit data movements between the DRAM and their private data memories to operate on the data stored in the DRAM.

The in-device DRAM (2 GB in total) holds PebbleSSD's BAM as well as other FTL's metadata. The DRAM also stores statistics and state information for each flash erase block so that the firmware can perform garbage collection and wear leveling efficiently.

In this paper, we assume that the DRAM is persistent. In practice, it would either be battery or capacitor-backed. New memory technologies e.g. Intel 3D-XPoint memory [1] could also replace the original DRAM.

4.2 BAM

The BAM is the core architectural component of PebbleSSD, and the firmware manages mapping information in the BAM to support `remap` and `fs_write`. Below we describe the implementation of `remap` and `fs_write` in more detail.

4.2.1 Physical-to-logical mapping and `remap`. PebbleSSD uses the BAM to track the bidirectional mapping between logical and physical address spaces, and the `remap` command exposes the mapping to software. The PPN-to-LBA table is an inverse of the normal LBA-to-PPN table that conventional FTLs use to emulate a block I/O interface.

PebbleSSD stores the PPN-to-LBA mapping table in the BAM. PebbleSSD uses a 64-bit integer to represent this information. The total size of the PPN-to-LBA table is no larger than that of the LBA-to-PPN table since the capacity of physical flash memory available in an SSD is usually smaller than the 64-bit logical address space.

The current PebbleSSD firmware implements this PPN-to-LBA mapping as an array. To keep the bidirectional mapping between logical and physical address space consistent and up-to-date, the `write` and `remap` commands always update the LBA-to-PPN and PPN-to-LBA tables together.

The `remap` command operates on both LBA-to-PPN and PPN-to-LBA mapping table. The `remap` command takes three parameters, `srcLBA`, `dstLBA` and `num`. The firmware copies the content of the `num` consecutive mapping table entries starting from `srcLBA` to the entries starting from `dstLBA`. Therefore, the actual data stored on those physical pages can be accessed from a new logical address `dstLBA` by future `read` operations. In this way, `remap` achieves the movement of data from `srcLBA` to `dstLBA` at low cost without incurring any flash write.

4.2.2 Efficient file block mapping and `fs_write`. PebbleSSD can also store the mapping from data blocks to corresponding file-system-specific information in BAM. Figure 4 illustrates the mapping tables used for this purpose. Each entry in the table has two components. The first component stores the data block's offset within the file, and the second component is a pointer that forms a linked list with other data blocks in the file.

The space required for each entry depends on file system data block size, the maximum size of an file and the capacity of the BAM. For example, if the file system's data block size is 4 KB, a file can reach 16 TB at most, and the BAM has a total capacity of 4 GB, then the firmware can use two 32-bit integers to store each entry.

For each opened file that supports `write` operation, PebbleSSD firmware maintains a linked list of the aforementioned mapping table entries. By traversing the per-file linked list, the firmware retrieves the metadata of all the data blocks that belong to the file

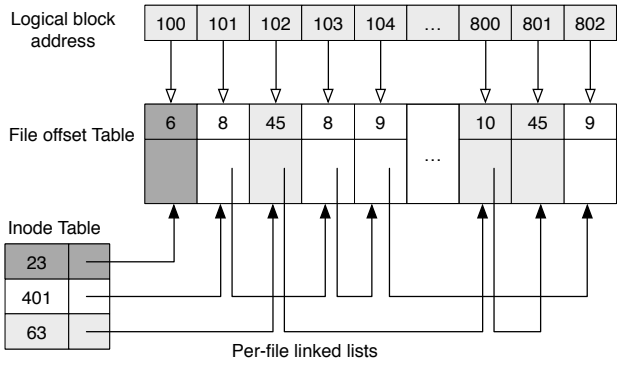

Figure 4: PebbleSSD write-optimized file block mapping PebbleSSD stores in the BAM an auxiliary file block mapping for files. This file block mapping comprises an inode table and a file offset table whose entries form per-file linked lists. The linked lists keep track of data blocks that have been written for each file since the latest checkpoint.

and that have been written back by the file system since the most recent file system checkpoint.

The linked lists are append-only. PebbleSSD is able to traverse the linked list and retrieve entries, but can only add new entries to its tail. The `fs_write` command causes the firmware to add mapping table entries to the linked list in addition to flushing a number of data blocks to flash memory. Once the firmware finishes the insertion into the linked list, the data blocks logically become part of the file so that the firmware can later locate them by traversing the linked list in BAM. Thus the file system no longer has to flush the index blocks of the file to flash memory frequently, effectively breaking the recursive updates in the wandering tree problem.

A separate hash table residing in BAM maps a file's inode number to the head of its linked list. PebbleSSD uses this hash table to locate the linked list that belongs to a particular file.

4.3 Block layer and device driver

The PebbleSSD requires a custom Linux block layer and device driver so that software can use the `fs_write` and `remap` commands. The block layer accepts extended `bio` requests from the file system and passes the `bio` structure to the SSD driver. In the extended `bio` struct, the field `bi_rw` encodes the type of the operation, and two new fields contain additional arguments to supply to the device driver, e.g. `offsetInfile`, `inodeNum` or `srcLBA`. Then the device driver translates the `bio` to appropriate NVMe commands and issues them to PebbleSSD.

5 FILE SYSTEMS CUSTOMIZATION

Log-structured file systems provide two opportunities to apply PebbleSSD's new capabilities. Leveraging the new commands allows these file systems to reduce the amount of data written to flash improving the SSD's lifetime.

The `remap` command of PebbleSSD allows log-structured file systems to perform fast and efficient log cleaning and reduce the amount of data written to the SSD. The `fs_write` command avoids the recursive updates on other interior node blocks in the tree-based block mapping index.

5.1 Log cleaning

Log-structured file systems perform log cleaning to create free, contiguous segments in the logical address space to service future write requests. During the cleaning process, log-structured file systems scan the logs, copy valid data to new locations, discard invalid data and reclaim the space originally occupied.

The `remap` command supports efficient movement of data inside the SSD without incurring flash writes and host-device data transfers. Log structured file systems can use `remap` to avoid copying data to a new segment as long as the data in the segment is clean (i.e., there does not exist a more recent update waiting in the operating system page cache).

Determining whether a block is clean is not always simple. The obvious approach is to use the Linux page descriptor's dirty bit, but this not always sufficient. For instance, F2FS [24] employs a "lazy migration" policy that marks valid data blocks in the page cache as dirty so the writeback thread will write them back later. To leverage `remap` in our version of F2FS, we used a new status bit to indicate whether this block is truly dirty or just marked dirty by the log cleaning thread.

The file system also has to track the old LBAs of all blocks before issuing the `remap` command. The original NILFS2 [10] fails to do so. After reading a file block into page cache, NILFS2 overwrites the original LBA. Consequently when the log cleaning thread later migrates this file block, it has no information about its source location. To address this issue, for each clean and up-to-date file block that the log cleaning thread plans to migrate, the `buffer_head` struct has an extra 64-bit field to store the old LBA.

5.2 Data block writing

Log-structured file systems can use PebbleSSD's new interface to improve the efficiency of writing file data blocks. The file system can issue one `fs_write` command to persist data blocks in flash memory and their metadata in the BAM area. The metadata include the blocks' offset within their file and the file inode number. This allows the file system to avoid the recursive updates of index blocks containing pointers pointing to the data blocks.

When writing back data blocks to a file with `fs_write`, the log-structured file systems send the following information to PebbleSSD via the block layer and device driver: `startLBA`, `length`, `inodeNum` and `offsetInFile`. The `fs_write` stores data blocks in flash memory and their associated metadata in the BAM area. Since PebbleSSD already allocates sufficient space for both inode table and file offset table, `fs_write` does not dynamically increase the space consumption of BAM.

The file system periodically flushes the original tree-based block mapping index to flash memory during checkpoint to limit the number of dirty blocks in the index. This helps reduce file system metadata loss and recovery overhead in the event of failures.

The file system maintains the conventional block mapping in page cache and continue to use it for normal file system operations unless there is a failure. The file system performs periodic

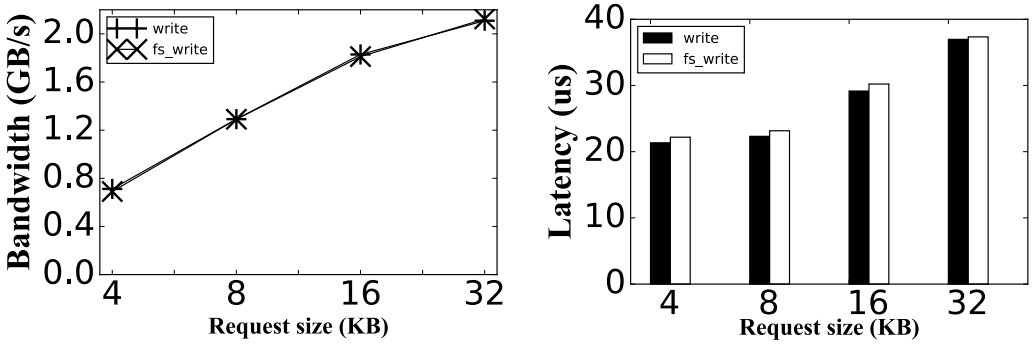

Figure 5: Performance comparison of `fs_write` and `write` (a) shows that `fs_write` achieves the same throughput as that of `write` since the overhead of linked list operations is minimal. (b) shows that `fs_write` and `write` have similar latency.

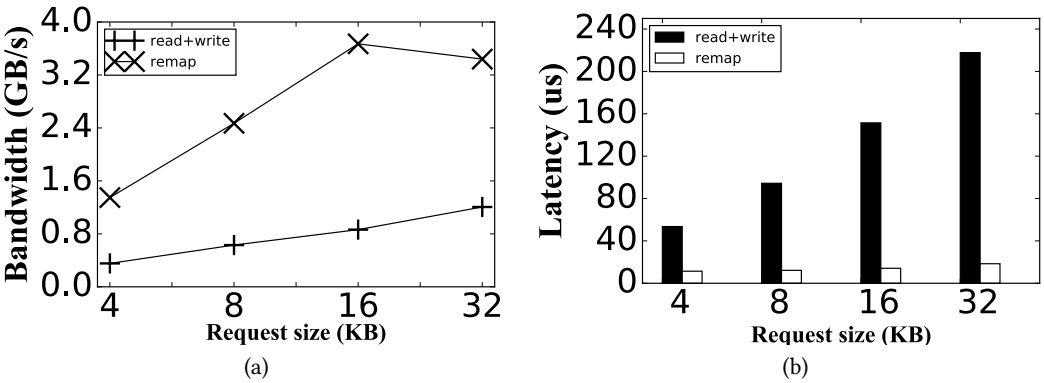

Figure 6: Performance of `MOVE` with conventional SSD and PebbleSSD (a) shows that the `remap`-based implementation of `MOVE` achieves 3.7 × improvement in throughput and (b) shows that it achieves 87% reduction in latency, compared with conventional implementation using `read` and `write`.

checkpointing and write back the node blocks of the conventional block mapping index. After the checkpointing task finishes, the file systems request that PebbleSSD reset the BAM-based block mapping via a vendor-specific NVMe command supported by our prototype. The function of this command is to clear a region in the NVRAM inside PebbleSSD.

The file systems use the BAM-based block mapping only for the purpose of recovery. Should an error occur, a kernel thread performs recovery on the data blocks of this file. The thread retrieves the `<inodeNum, offsetInFile>` metadata from the BAM by reading its content into host main memory and reconstruct the normal tree-based block mapping.

6 EVALUATION

We run microbenchmarks to measure the performance of PebbleSSD with a focus on the new commands it provides. Then we measure the performance and efficiency of log cleaning of both F2FS and NILFS2 with `remap` and compare with their baseline implementations. We refer to the customized versions as `F2FS-opt` and `NILFS2-opt`, while we refer to the baseline as `F2FS-baseline` and `NILFS2-baseline`. Finally we run workloads to quantify the benefit of `fs_write` on F2FS and NILFS2. Again, the customized versions are `F2FS-opt` and `NILFS2-opt`, while the baseline versions are `F2FS-baseline` and `NILFS2-baseline`. In this section, we describe our experimental setup and results in detail.

6.1 Experimental setup

The specification of our SSD hardware platform is covered in Section 4.1. We implemented PebbleSSD by modifying its original reference firmware provided by the SSD manufacturer. The original firmware supports only conventional block-based I/O, while PebbleSSD extends the NVMe command set to provide additional features, e.g. `fs_write` and `remap`. To allow host programs to interact with PebbleSSD, we also modified Linux NVMe driver and block layer. Thus PebbleSSD supports not only block-based I/O, but also `fs_write` and `remap`. Before running the experiments, we preconditioned the SSD by filling it with random data multiple times.

The host machine that we use has a quad-core Intel Xeon E3-1230 CPUs on a single-socket motherboard. This machine contains 32 GB of DRAM as main memory and runs a customized Linux 3.16.3 kernel. We collected all results with hyper-threading disabled.

Microbenchmarks measure the bandwidth and latency of the new commands `fs_write` and `remap` provided by PebbleSSD. To measure PebbleSSD's bandwidth, four threads running on the host machine issue commands to PebbleSSD continuously. To measure latency, only a single thread issues commands to the SSD.

We apply `remap` and `fs_write` respectively to both F2FS and NILFS to study the effects of each feature independently. To measure the performance and efficiency of `remap` on log-structured file systems, we first run a subset of Filebench [4] workloads. We run the workloads for equal amount of time on each baseline file system and its customized version. Then we manually trigger file-system-specific log cleaning subroutines which keeps executing until there is no data to clean.

We run a subset of Filebench, an open-source implementation [13] of TPC-C [14] and LinkBench [7] to quantify the benefits of `fs_write` on F2FS and NILFS2, comparing with their baselines.

6.2 Microbenchmarks

In comparison with original `write` command, the `fs_write` command not only writes the block to SSD, but also updates its BAM. Since we are concerned about the single command performance here, we eliminated the overhead of Linux virtual file system (VFS) by allowing applications to send `fs_write` commands to PebbleSSD directly via `ioctl`. Figure 5 presents the throughput and latency of `fs_write` and compares it with the original `write` command. The extra overhead caused by `fs_write`'s linked list operations is minimal and `fs_write` achieves almost the same performance as `write`. Although as a single command, `fs_write` does not provide better performance than `write`, but as we will demonstrate in Section 6.4, log-structured file systems can utilize `fs_write` to improve the performance and lifetime of SSDs.

Another microbenchmark named `MOVE` measures the performance of `remap` command. `MOVE` requires host program to move data from source LBAs to destination LBAs. The baseline version of `MOVE` implementation first reads data from PebbleSSD into host main memory and then writes them to their destination locations in the SSD. In contrast, `remap`-based implementation uses the `remap` command to remap data from source LBAs to destinations. Apparently the baseline incurs multiple data transfer between the host and PebbleSSD while `remap`-based version does not suffer from this overhead. As a result, `remap`-based version can achieve 3.7 × improvement in throughput and reduces the latency by 87% on average, as shown in Figure 6. Both implementation sends commands from the host to PebbleSSD via `ioctl` directly.

6.3 Efficient log cleaning

Figure 7 depicts the effect of using `remap` in the log cleaning of F2FS and NILFS2. In most cases, a kernel thread executes the log cleaning subroutine for F2FS and the kernel thread wakes up periodically, typically between every 30 and 60 seconds. In order to measure how much time it takes to clean all used segments in F2FS, we add an `ioctl` to F2FS to trigger F2FS log cleaning mannually and let the log cleaning subroutine run continuously until there is nothing left to clean. For NILFS2, we use its own `nilfs-clean` utility program [9] that performs log cleaning and space reclamation after a certain period of time specified by the user. In our experiments, we set the period to 600 seconds.

Before starting log cleaning subroutines, we first run the corresponding workload to emulate a used SSD. Figure 7 (a) shows the time required to finish log cleaning of F2FS, and Figure 7 (b) shows the amount of data written by F2FS log cleaning. `F2FS-opt` spends 33% less time than `F2FS-baseline` to finish cleaning the segments in average. Figure 7 (b) can account for the reason of this improvement. `F2FS-baseline` always causes flash write operations during log cleaning, while `F2FS-opt` can use `remap` to clean the majority of data. In the case of `F2FS-opt`, the log cleaning subroutine writes up to 99% less data than `F2FS-baseline` in average. The rest of data are cleaned using the `remap` command which does not incurr flash write at all.

Figure 7 (c) and (d) depict the result for NILFS2 log cleaning. Similarly, `nilfs-clean` writes up to 97% less data in the case of `NILFS2-opt` than `NILFS2-baseline` because the former can use `remap` to move data. Since NILFS2 is not optimized for parallel storage devices e.g. SSDs, `nilfs-clean` does not fully utilize the I/O bandwidth. Consequently, the saving in log cleaning time is not as big as in F2FS. `NILFS2-opt` improves the SSD lifetime because it incurs less flash write than `NILFS2-baseline`.

6.4 Write-optimized file block index

We measure the impact of `fs_write` on F2FS and NILFS2. `F2FS-baseline` already writes only direct node blocks to the SSD during `fsync` [24] to improve the performance of `fsync`. `F2FS-opt` uses `fs_write` to further avoid writing direct node blocks. To compare the effectiveness in reducing the amount of data written to the SSD, we use a metric called *file write amplification* similar to the concept in [28] which is defined as the quotient of the total amount data written (data and index blocks) over the amount of file data blocks.

Figure 8 illustrates the advantage of `F2FS-opt` over `F2FS-baseline` and `NILFS2-opt` over `NILFS2-baseline` in a simple benchmark. The benchmark executes four threads each of which writes data to its dedicated file synchronously. We assign each thread to its own file to avoid the contention for the lock on the file's inode. Figure 8 (a) shows that with `F2FS-opt`, the aggregate throughput of the threads achieves 1.28 × improvement comparing with `F2FS-baseline`. Figure 8 (b) shows that the file write amplification of `F2FS-opt` is 28% lower than that of `F2FS-baseline`. Figure 8 (c) similarly shows that the benchmark throughput on `NILFS2-opt` is 1.13 × the throughput on `NILFS2-baseline`. Figure 8 (d) shows that file write amplification of `NILFS2-opt` is 24% less than that of `NILFS2-baseline`. The performance improvement and reduction in file write amplification result from `fs_write` that writes only data blocks for both `F2FS-opt` and `NILFS2-opt`.

Figure 9 and Figure 10 further compare the effect of running more complex benchmarks on `F2FS-baseline`, `F2FS-opt`, `NILFS2-baseline` and `NILFS2-opt`. Figure 9 (a) presents the throughput of Filebench OLTP benchmark, and `F2FS-opt` outperforms `F2FS-baseline` by 3% while `NILFS2-opt` achieves 9% improvement. Figure 9 (b) shows the throughput of TPC-C varying the size of data set. The total

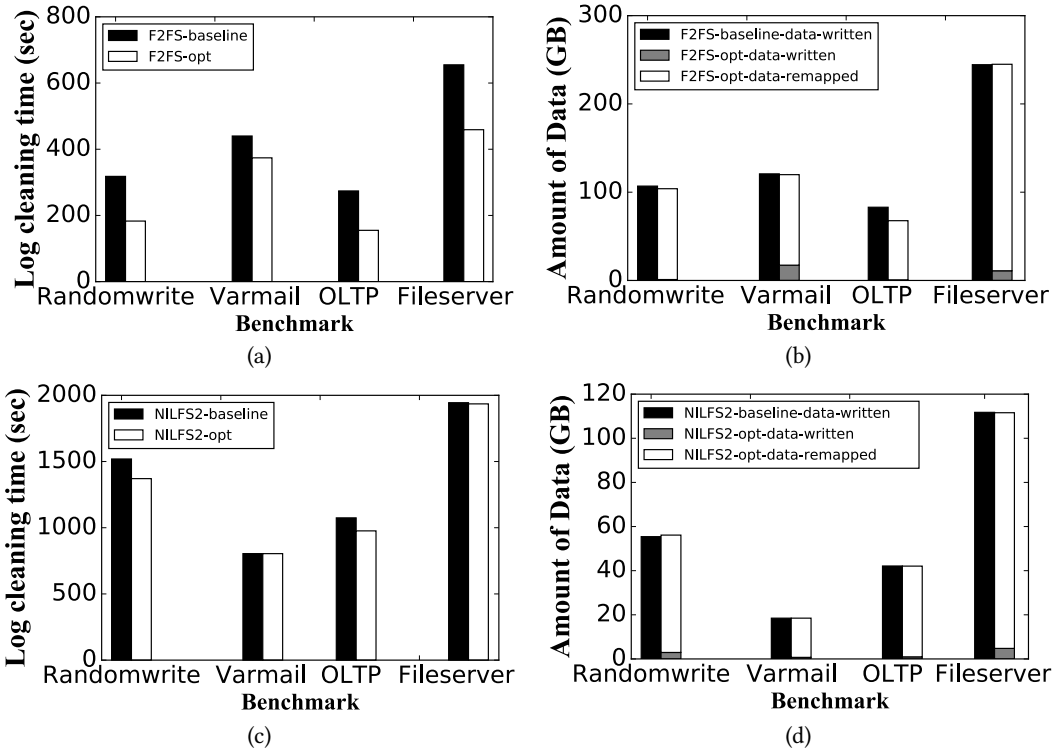

Figure 7: Improvement on log cleaning efficiency due to `remap` In (a) and (b), `F2FS-opt` with `remap` requires 29% less time to clean the segments than `F2FS-baseline`, and writes 96% less data in average (99% in the best case) during log cleaning. In (c) and (d), `NILFS2-opt` also requires less time to clean the old segments than `NILFS2-baseline`, and saves write traffic by 97% in average (99% in the best case) during log cleaning.

data set size of TPCC-10 and TPCC-100 are approximately 800 MB and 8 GB respectively. Both benchmarks run on top of MySQL [8] 5.5 with InnoDB [6] whose buffer pool size is 16 GB. In both cases, `F2FS-opt` outperforms `F2FS-baseline` by a small margin and the same holds for `NILFS2-opt` and `NILFS2-baseline`. Figure 9 (c) shows the result for LinkBench with different sized data sets. Similarly, `F2FS-opt` and `NILFS2-opt` lead to larger throughput than `F2FS-baseline` and `NILFS2-baseline` respectively. The throughput improvement is not high because even `F2FS-baseline` and `NILFS2-baseline` cannot fully saturate the IO bandwidth of PebbleSSD, thus writing less data does not lead to larger throughput.

Figure 10 shows that for Filebench OLTP, TPCC-10 and LinkBench-10GB, `F2FS-opt` can use `fs_write` to reduce the file write amplification from close to 1.5 to nearly 1.0. Similarly, `NILFS2-opt` reduces file write amplification from 2.7 to 2.0. When the data set size increases, especially in the case of LinkBench-100GB, the reduction in file write amplification diminishes since MySQL will have to move data between main memory buffer pool and the SSD, leading to lower write throughput. Consequently the workloads issue fewer `fsync` to the underlying log-structured file systems.

6.5 BAM space utilization

With `remap` and `fs_write`, PebbleSSD can make more efficient use of its internal NVRAM than conventional SSDs.

As described in Section 4.1, PebbleSSD exposes 375 GB flash to the host and has 2 GB NVRAM to store BAM. Assume the total amount of user-visible flash is 512 GB for simpler calculation.

According to the implementation of original firmware, a conventional SSD based on this platform uses 4 bytes for each address mapping table entry, and each entry maps 4 KB in logical address space to the physical address space. This configuration leads to 2^{27} entries, consuming 512 MB of the BAM. Since the original SSD keeps the PPN-to-LBA mapping in flash memory, the space utilization of BAM is about 35% due to the logical-to-physical mapping and some other system data structures.

To support `remap`, PebbleSSD maintains both logical-to-physical and physical-to-logical mappings in BAM. The entries in both tables are 4 bytes in size and describes the mapping for 4 KB regions. Therefore, the total space consumption will be 1 GB, leading to a 60% utilization of BAM.

For `fs_write`, the original address mapping table also occupies 512 MB of BAM. Both the file offset table and inode table use 8-byte entries. The file offset table has 2^{27} entries, consuming 1 GB BAM. The inode table keeps an entry for each file that is currently open

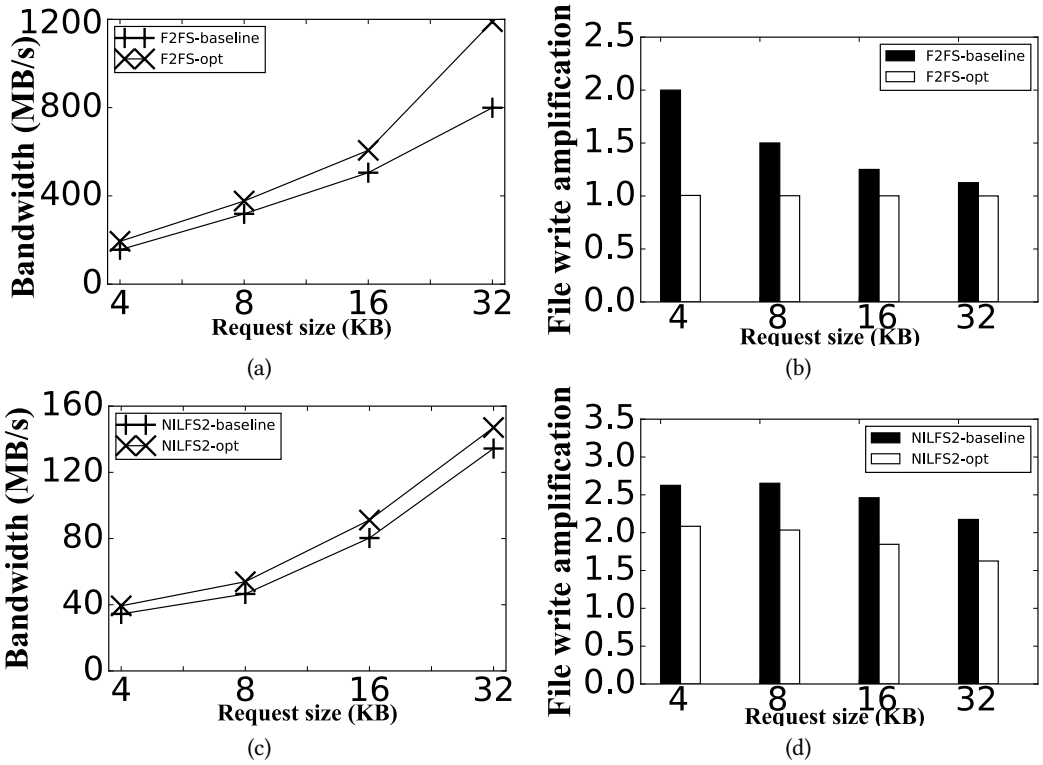

Figure 8: Improvement on writing to files with `fs_write` (a) shows that with `fs_write`, `F2FS-opt` achieves 1.28 × improvement in throughput while (b) shows that `F2FS-opt` reduces file system write amplification by 28%. (c) and (d) show similary results. With `fs_write`, `NILFS2-opt` improves throughput by 1.13 × and reduces file write amplification by 24%, compared with `NILFS2-baseline`.

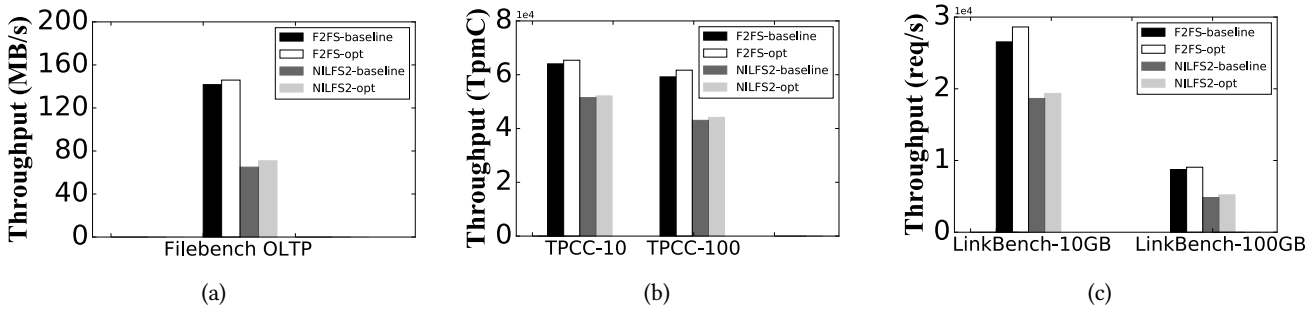

Figure 9: Performance comparison between `F2FS-opt`, `NILFS2-opt` and their baseline implementations In (a), Filebench OLTP running on `F2FS-opt` and `NILFS2-opt` achieve 1.02 × and 1.09 × improvement in throughput. In (b), TPCC running on `F2FS-opt` and `NILFS2-opt` can achieve slightly better performance than on `F2FS-baseline` and `NILFS2-baseline` respectively. In (c) LinkBench can also achieve slightly larger throughput on `F2FS-opt` `NILFS2-opt` than on `F2FS-baseline` and `NILFS2-baseline`.

with read-write access. In our current system, we support 1 million files that can be open for write at the same time. Considering the load factor of the hash-based inode table, this requires approximately 16 MB space in BAM. The space utilization of BAM can reach 80%.

7 RELATED WORK

Many prior research efforts have explored various techniques aiming to extend the lifetime of flash-based SSDs by writing less data. In this section, we present a review of these efforts and place PebbleSSD in the context with them.

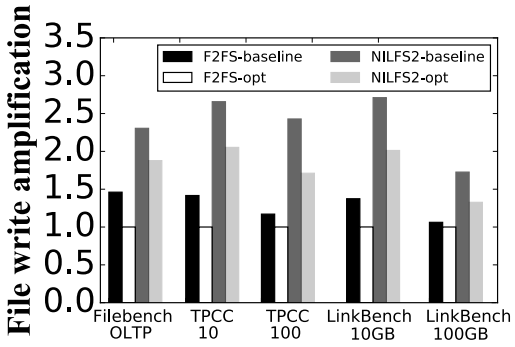

Figure 10: Improvement on file write amplification for F2FS and NILFS2 `F2FS-opt` can reduce file write amplification from 1.5 to nearly 1.0 for synchronous write-intensive workloads e.g. database applications. `NILFS2-opt` can reduce file write amplification from 2.7 to 2.0 for the same workloads.

7.1 Backward pointer

The mapping information stored by PebbleSSD in the BAM is similar to the backward pointers in some previous work. Many file systems and FTLs have adopted the backward pointer mechanism to reduce the overhead of persistent data migration and synchronization. File systems, e.g. Pilot [32] file system, Pangaea [34], NoFS [17] use backward pointers to ensure system consistency. BtrFS [2] and Backlog [29] maintain back references to support dynamic relocation of data blocks without flushing index updates to persistent storage. An object-based FTL, i.e. OFTL [28] stores for each page the information of the object to which the page belongs. Such backward reference resides in the flash-based OOB region of each page, thus does not support in-place update and requires flash write operations. In contrast, PebbleSSD stores the mapping in dedicated OOB region as BAM to support dynamic modification. In addition, PebbleSSD allows for easy and efficient retrieval of each file's mapping by maintaining per-file linked list of mapping table entries.

7.2 Address indirection

Most flash-based SSDs introduce a layer of indirection due to the FTL's address map [19]. Researchers have proposed a number of FTLs [5, 16, 20, 21, 23, 27, 35]. CAFTL [16] exploits the hashed signature of data chunks to detect duplicates and reduce flash writes, leading to improvement in SSD's lifetime. FTL2 [35] implements atomicty at the level of FTL so as to reduce the overhead of writing database logs to flash. DFTL [21] selectively caches page-level mapping. These approaches still focus on the normal logical-to-physical address mapping table, while PebbleSSD goes a step further to use BAM to enhance the flexibility of FTLs.

Several systems have also researched the possibility of achieving efficient flash-write-free data movement by manipulating this map directly. JFTL [18] remaps addresses of journal pages to their original locations in the logical address space without writing the same data to flash memory. ANViL [36] allows host system to access the address map and supports snapshot, deduplication and single-write journaling. Researchers evaluated JFTL on a simulation platform, while ANVil's address map resides in the main memory of host machine. Furthermore, neither of them discussed how to extend the idea of address map manipulation to an SSD with flash-based OOB. If the FTL stores the LBA in the spare space of each flash page, merely modifying the LBA-to-PPN address map does not fulfil the goal of moving data in the logical address space. Instead flash write is inevitable to modify the metadata in the OOB region. In contrast, PebbleSSD goes a step further to address this issue and presents an evaluation of the effectiveness of its solution.

SHARE [30] is also able to remap addresses inside the SSD and achieve write atomicity. SHARE caches a subset of the PPN-to-LBA mapping entries in the SDRAM inside the OpenSSD [12], therefore, the firmware determines which entries to keep and which to evict to flash. This approach is able to handle large amount of flash with relatively small SDRAM, but requires extra logic at the firmware level to implement cache eviction policies. Furthermore, system performance can also be affected by the size of the cache. In contrast, PebbleSSD keeps the entire PPN-to-LBA mapping in BAM for the functionality of `remap`. PebbleSSD needs larger BAM size than SHARE, but does not require a cache management policy. SHARE and PebbleSSD represent different design trade-offs and considerations. Depending on the optimization goal, either can be more preferable than the other.

7.3 Coordinated garbage collection

A storage system with an LFS running on a flash-based SSD has multiple layers of log structures, and the gaps between them lead to inefficiencies such as unnecessary data migration [37].

Researchers have proposed coordinating file system log cleaning with SSD's garbage collection to improve flash device lifetime. Application-Managed Flash (AMF) [26] and ParaFS [38] both employ a coordinated garbage collection approach with supports from both the FTL and file system. During garbage collection, the host file system migrates data to its proper new locations by consulting domain-specific knowledge about data placement, while the FTL shoulders the responsibility of erasing flash blocks for future use. Both file systems need to accommodate the physical characteristics of flash memory. For example, they need to ensure that a flash block is erased first before writing to its flash pages. This potentially requires a non-trivial re-engineering of the I/O path because general file systems do not have this limitation. Furthermore, host CPUs are involved in the GC of flash memory. In comparison, PebbleSSD does not require the file system to be aware of the physical layout and characteristics of underlying flash. Thus modification of legacy file system is moderate, and host CPUs can also be freed from GC.

Coordinated GC has also been discussed in other conventional file systems running on SSDs. EXT3 [3] with nameless write [39] allows the FTL to move valid flash pages and inform the host file system of the new physical addresses of data pages via *migration callbacks*. PebbleSSD, in contrast, lets the file system determine the new logical addresses for its data.

7.4 Metadata caching

Some efforts use NVRAM-based storage as a durable cache for database or file system metadata to reduce synchronous writes to flash memory. NVMFS [31] stores file system metadata in NVM DIMMs

attached to the memory bus. DuraSSD [22] uses non-volatile in-device cache to support atomicity and durability. Cooperative Data Management [25] considers NVM as a cache and allows the NVM-resident copy to invalidate flash-resident copy. In contrast, PebbleSSD does not cache identical copies of data in NVRAM. Instead, PebbleSSD stores metadata optimized for the characteristics of BAM, saving host interface bandwidth and NVRAM space.

8 CONCLUSION

By providing BAM in the NVRAM-based OOB region, PebbleSSD supports a range of useful features to improve device lifetime, including write-optimized file block mapping and fast, efficient log-cleaning. PebbleSSD exposes two new commands, `fs_write` and `remap` allowing file systems to access BAM. Log-structured file systems can use `fs_write` command to avoid recursive updates on file index blocks, and use `remap` command to perform fast and efficient movement of valid data during log cleaning. Log-structured file systems achieve better performance while reducing flash writes. Therefore BAM is effective in improving SSDs' lifetime.

ACKNOWLEDGMENT

We would like to thank the reviewers for their helpful suggestions. This work was supported in part by CFAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also sponsored in part by Samsung, NSF award 1219125 and NSF BIGDATA 1447943.

REFERENCES

[1] *3D-XPoint.* https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.
[2] *BtrFS official site.* https://btrfs.wiki.kernel.org/index.php/Main_Page.
[3] *EXT3 file system.* http://lxr.free-electrons.com/source/fs/ext3/ext3.h?v=3.7.
[4] *Filebench.* https://github.com/filebench/filebench/wiki.
[5] *FusionIO VSL.* http://www.fusionio.com/products/vsl.
[6] *InnoDB.* https://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.
[7] *LinkBench.* https://github.com/facebookarchive/linkbench.
[8] *MySQL.* https://www.mysql.com/.
[9] *NILFS2 clean utility.* http://nilfs.sourceforge.net/en/man8/nilfs-clean.8.html.
[10] *NILFS2 official site.* http://nilfs.sourceforge.net/en/.
[11] *NVM Express official site.* http://www.nvmexpress.org.
[12] *OpenSSD.* http://www.openssd-project.org/wiki/The_OpenSSD_Project.
[13] *Percona-Lab implementation of TPC-C.* https://github.com/Percona-Lab/tpcc-mysql.
[14] *TPC-C.* http://www.tpc.org/tpcc/.
[15] A. Bityutskiy. 2005. *JFFS3 design issues.* www.linux-mtd.infradead.org/tech/JFFS3design.pdf.
[16] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives.. In *FAST*, Vol. 11.
[17] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. Consistency without Ordering.. In *FAST*. 9.
[18] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. 2009. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *Trans. Storage* 4, 4, Article 14 (Feb. 2009), 22 pages. https://doi.org/10.1145/1480439.1480443
[19] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of flash translation layer. *Journal of Systems Architecture* 55, 5 (2009), 332–343.
[20] Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 138–163.
[21] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/1508244.1508271
[22] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 529–540. https://doi.org/10.1145/2588555.2595632
[23] Yangwook Kang, Rekha Pitchumani, Thomas Marlette, and Ethan L Miller. 2014. Muninn: A versioning flash key-value store using an object-based storage model. In *Proceedings of International Conference on Systems and Storage*. ACM, 1–11.
[24] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 273–286. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee
[25] E. Lee, J. Kim, H. Bahn, and S. H. Noh. 2016. Reducing Write Amplification of Flash Storage through Cooperative Data Management with NVM.. In *MSST*. 1.
[26] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 339–353. http://usenix.org/conference/fast16/technical-sessions/presentation/lee
[27] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review* 42, 6 (2008), 36–42.
[28] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 257–270. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou
[29] Peter Macko, Margo I Seltzer, and Keith A Smith. 2010. Tracking Back References in a Write-Anywhere File System.. In *FAST*. 15–28.
[30] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. 2016. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 343–354.
[31] S. Qiu and A. L. Narasimha Reddy. 2013. NVMFS: A Hybrid File System for Improving Random Write in Nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–5. https://doi.org/10.1109/MSST.2013.6558434
[32] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. 1980. Pilot: An Operating System for a Personal Computer. *Commun. ACM* 23, 2 (Feb. 1980), 81–92. https://doi.org/10.1145/358818.358822
[33] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/121132.121137
[34] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. 2002. Taming Aggressive Replication in the Pangaea Wide-area File System. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 15–30. https://doi.org/10.1145/844128.844131
[35] Tianzheng Wang, Duo Liu, Yi Wang, and Zili Shao. 2013. FTL2: a hybrid flash translation layer with logging for write reduction in flash memory. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 91–100.
[36] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2015. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 111–118. https://www.usenix.org/conference/fast15/technical-sessions/presentation/weiss
[37] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang
[38] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 87–100. https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang
[39] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with Nameless Writes.. In *FAST*. 1.