OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface

Yun-Sheng Chang and Ren-Shuo Liu, ***National Tsing Hua University***

https://www.usenix.org/conference/atc19/presentation/chang

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface

Yun-Sheng Chang and Ren-Shuo Liu
Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan

Abstract

Consumer-grade solid-state drives (SSDs) guarantee very few things upon a crash. Lacking a strong disk-level crash guarantee forces programmers to equip applications and filesystems with safety nets using redundant writes and flushes, which in turn degrade the overall system performance. Although some prior works propose transactional SSDs with revolutionized disk interfaces to offer strong crash guarantees, adopting transactional SSDs inevitably incurs dramatic software stack changes. Therefore, most consumer-grade SSDs still keep using the standard block device interface.

This paper addresses the above issues by breaking the impression that increasing SSDs' crash guarantees are typically available at the cost of altering the standard block device interface. We propose Order-Preserving Translation and Recovery (OPTR), a collection of novel flash translation layer (FTL) and crash recovery techniques that are realized internal to block-interface SSDs to endow the SSDs with *strong request-level crash guarantees* defined as follows: 1) A write request is not made durable unless all its prior write requests are durable. 2) Each write request is atomic. 3) All write requests prior to a flush are guaranteed durable. We have realized OPTR in real SSD hardware and optimized applications and filesystems (SQLite and Ext4) to demonstrate OPTR's benefits. Experimental results show 1.27× (only Ext4 is optimized), 2.85× (both Ext4 and SQLite are optimized), and 6.03× (an OPTR-enabled no-barrier mode) performance improvement.

1 Introduction

Storage systems are constructed layerwise; thus the overall system performance depends on an appropriate division of labor between layers. For example, for applications and filesystems that run on top of flash-based solid-state drives (SSDs), if the underlying SSDs focus too much on optimizing their own performance and maintain too weak guarantees upon a crash or power loss (crash for short), programmers are forced to equip the applications and filesystems with safety nets using redundant writes and flushes [24, 26, 33, 41], which in turn complicate the overall system and degrade the overall performance.

That being said, widely used, consumer-grade SSDs (baseline SSDs for short) guarantee very few things upon a crash:

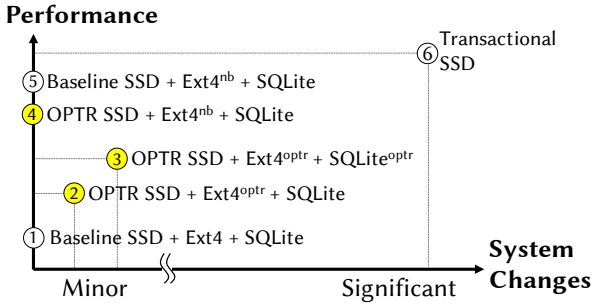


Figure 1: OPTR SSD vs baseline and transactional SSDs. $Ext4^{optr}$ and $SQLite^{optr}$ denote our optimized versions of Ext4 and SQLite, respectively. $Ext4^{nb}$ denotes Ext4 mounted with the -o nobarrier option.

each individual sector that is written since the last flush may either be done or left undone, which means there is an enormous set of post-crash states for applications and filesystems to handle. Lacking a strong disk-level crash guarantee has caused deep, long-term consequences. For example, the ordering constraint among user data, metadata, journal, and commit records should be enforced to prevent sensitive user data and inconsistent states from being accidentally exposed. However, disks are not obligated to preserve any write order, and filesystems are reluctant to use flushes to enforce a write order due to severe performance penalties. Therefore, Linux Ext3 even turned off flushes for many years at the risk of crash vulnerabilities [20]. Another alarming example is that since the crash guarantees of underlying disks are weak, various filesystems struggle to provide a standardized and strong crash guarantee at their level, and this struggle consequently makes applications have difficulty ensuring correct recovery from a crash. For example, LevelDB and Git were recently found to contain many crash vulnerabilities owing to this reason [39].

Despite many issues, SSDs with a weak crash guarantee are still widely used for two reasons. First, the weak crash guarantee has existed since the hard disk drive (HDD) era. This design choice was unavoidable for optimizing HDD performance and remains as a matter of course for optimizing SSD performance. For example, HDDs were permitted to freely reorder requests according to the position of pickup

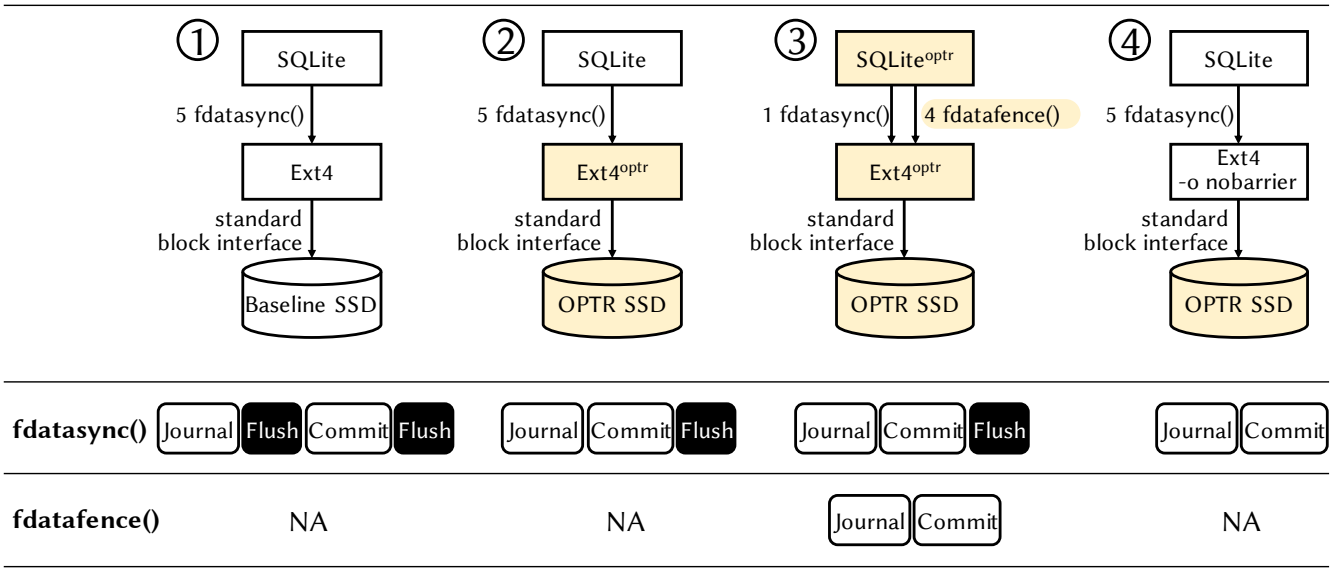


Figure 2: Usage modes of OPTR SSDs and their filesystem and disk interfaces. *fdatafence* is our newly proposed filesystem primitive that incurs no flush. More details on the original five *fdatasync* and the use of *fdatasync* are in Section 4.

heads to minimize seek and rotational latency, and thus, SSDs are also allowed to reorder or parallelize requests to maximize their internal channel- and chip-level parallelism. The second reason is that although novel SSDs with revolutionized interfaces that offer transaction-level crash guarantees (transactional SSDs for short) [28, 32, 37, 40, 44] have recently been studied, the fact is that there has been a vast body of software and systems developed based on a *standard block device interface* (block interface for short, e.g., SATA). Therefore, as shown in Figure 1, although transactional SSDs (⑥) can potentially offer much higher performance than baseline SSDs (①), only few systems can adopt such dramatic changes. Not surprisingly, most consumer-grade SSDs keep using the block interface, and thus, the benefits of transactional SSDs are not widely available.

This paper addresses the above issues by breaking the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the block interface. We propose Order-Preserving Translation and Recovery (OPTR), a collection of novel flash translation layer (FTL) and crash recovery techniques that are realized internal to a block-interface SSD to endow the SSD with ***strong request-level crash guarantees*** as defined as follows:

1. **Prefix Semantics**: The SSD does not make a write request durable unless all the write requests received previously by the SSD are durable (stronger than baseline SSDs).
2. **Request Atomicity**: Each write request received by the SSD is atomic regardless of the request size (i.e., number of sectors) (stronger than baseline SSDs).
3. **Flush Semantics**: The SSD guarantees durability to all write requests that are received prior to a flush (identical to baseline SSDs).

Figure 3 uses a four-sector SSD to demonstrate the *strong request-level crash guarantees*. The four sectors initially store four version numbers, 0, 0, 0, and 0, respectively. The SSD receives four write requests and one flush request before a crash occurs. Each write request is specified by its *lba* and *size* in parentheses. We assume that write requests always increment the version number(s) of the written sector(s). For example, the first write request touches the first and second sectors, and thus the four version numbers become 1, 1, 0, and 0. As shown in the figure, each sector of a baseline SSD can exhibit two to three valid post-crash version numbers; therefore, the baseline SSD can exhibit $2\times2\times3\times3=36$ valid post-crash results. In comparison, an OPTR SSD guarantees to complete write requests in order and atomically, and there are only two write requests after the last flush; therefore, the number of valid results is significantly confined to three.

The *strong request-level crash guarantees* and our intentional choice to use a *standard block interface* bring several benefits. First, for programmers who want to develop new applications or filesystems, since the crash guarantees of OPTR SSDs are truly intuitive, the chance of making mistakes decreases. Additionally, since OPTR significantly confines the number of valid post-crash states, testing or verifying the correctness of a program becomes more manageable. Second, in addition to developing new programs, it is also simple for programmers to optimize existing applications and filesystems to benefit from OPTR. For example, Ext4 and SQLite resort to flushes to realize barriers because barriers are unavailable to most SSDs [45]. We refer to these flushes as ***unnecessary flushes***. With OPTR's *strong request-level crash guarantees*, we can optimize Ext4 and SQLite to remove

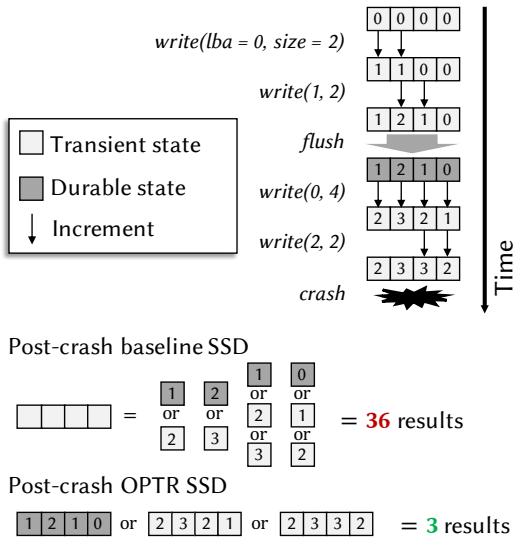


Figure 3: Post-crash states of OPTR SSDs vs. those of baseline SSDs.

unnecessary flushes (Section 4). The changes made to SQLite and Ext4 code are minor, and the achieved performance is significant, as illustrated by ② (optimizing Ext4 only) and ③ (optimizing both Ext4 and SQLite) in Figures 1 and 2. Finally, all existing, unmodified applications and filesystems can still run on OPTR SSDs, and this backward compatibility enables an *OPTR-enabled no-barrier mode* as described as follows.

In the ***OPTR-enabled no-barrier mode***, a filesystem is mounted with a no-barrier option (e.g., -o nobarrier for Ext4) and run on top of an OPTR SSD, as illustrated by ④ in Figures 1 and 2. In contrast to mounting the filesystem with the no-barrier option on a baseline SSD (⑤ in Figure 1), this mode achieves the best of both worlds, i.e., high performance (which is the reason why Ext3 used to disable flushes [20]) and consistency guarantees (because OPTR preserves order without the need for explicit flushes). This mode is practical and useful for smartphones, consumer-grade computers, and less-critical database systems such as SQLite [3] and some key-value stores [18][1].

This work makes the following contributions.

- This is the first SSD work with *strong request-level crash guarantees* and a standard block interface. We change the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the standard block device interface. We present the address translation, garbage collection (GC), and crash recovery algorithms internal to SSDs to achieve OPTR.
- We extend and restructure the FTL of an academic domain SSD project (OpenSSD [2]) to equip it with a write cache as our baseline. This FTL implementation is more sophisticated and modularized.
- We extend the FTL mentioned above to realize OPTR.
- We develop a functional simulator that can simulate an FTL given IO requests and crash events at high speed to test whether the FTL can recover from a crash and meet the OPTR requirement. We have validated this simulator and our implemented FTLs against each other. We anticipate that the simulator itself will be a useful tool for future FTL research.
- We exploit the *strong request-level crash guarantees* by optimizing the *fsync* primitive of Ext4 and newly proposing an *ffence* primitive for Ext4. These two primitives help to eliminate *unnecessary flushes*. We also demonstrate optimizing a database system, SQLite, to exploit the two primitives.

The rest of this paper is organized as follows. Section 2 describes the background of SSDs. Section 3 presents the detailed design and implementation of OPTR. Section 4 demonstrates the filesystem and application optimizations enabled by OPTR. Section 5 shows the results of validating OPTR using our developed functional simulator. Section 6 evaluates OPTR's gain and overhead. Section 7 analyzes the FTL in more detail. Section 8 surveys related work, and Section 9 concludes this work.

2 Background

2.1 Flash Translation Layers

In the core of an SSD lies an FTL. FTLs are responsible for translating a sequence of host-issued requests, including *write*, *read* and *flush*, into a sequence of flash operations, including *page-program*, *page-read*, and *block-erase*. Both read and write requests specify the address range of the data at the sector granularity; flush instructs an SSD not to acknowledge the host until the SSD persists all cached writes on stable media.

Upon receiving a write request, the FTL segments the write data into pages based on the given address range. A write request involves modifications to the logical-to-physical (L2P) mapping table. Many mapping schemes have been proposed (e.g., page-level, block-level, and hybrid) for various cost-performance tradeoffs. This work is based on page-level FTLs. Page-level FTLs divide the logical address space of SSDs into pages, which are indexed by a logical page number (LPN). Each entry in the L2P table maps an LPN to the location of a flash page indexed by a physical page number (PPN).

To handle a read request, the FTL translates the address range into LPNs and performs L2P table lookups to obtain the PPNs of the requested pages. If the cache happens to have a requested page, the contents are returned to the host; otherwise, a page-read is performed to retrieve the contents from the flash memory.

Flush requests are synchronous; thus, upon receiving a flush, the FTL must persist all dirty data in the write cache

[1] Note that this mode is not suitable for critical systems such as financial transaction processing systems because it relaxes durability.

and ensure no ongoing page-program operation exists before returning a success acknowledgment to the host.

As flash memory forbids in-place updates, overwriting data is done by writing the updated data to a free page and leaving the outdated data in the original page. A dedicated routine, called GC, is designed to reclaim these invalid pages, which store outdated data. The GC routine starts with choosing a victim block based on, e.g., the greedy policy [29, 42]. Then, this routine performs a series of page-reads and page-programs to relocate the valid data in the victim block. Finally, the victim block is erased and added to the free-block list.

2.2 High-Performance Schemes

Modern SSDs achieve significant performance mainly owing to the following three schemes: internal parallelism, request scheduling, and write caching. Note that these high-performance schemes all break the write order.

SSDs typically consist of multiple independent internal channels for transferring data and commands. Each channel connects to multiple flash chips. The **internal parallelism** of SSDs comes from these channels and chips, which can perform flash operations independently.

The goals of **request scheduling** are to increase the number of parallelized flash operations and to decrease the latency of each request. The former can be achieved by reordering requests to reduce resource conflicts [34]. The latter can be achieved by prioritizing requests with lower latency [22, 27].

Write caching removes slow storage accesses from the critical path. As the access speed of flash memory is often orders of magnitude slower than that of DRAM, most commercial SSDs employ a DRAM-based write-back cache for better performance. Another advantage of write caching is write coalescing. Multiple writes targeting the same LPN have the opportunity to be merged into one page-program operation, resulting in higher throughput and longer flash lifetimes.

2.3 SSD Recovery

The recovery mechanism of SSDs rebuilds the L2P table after a system crash. Some previous works have studied SSD recovery [7, 8, 11, 17, 30]. Leaving LPN information in the spare area of each written page during writes is the most common strategy [7], and the L2P information can be reconstructed after a crash by scanning this information. As multiple physical pages may contain the same LPN, a sequence number is often used to determine their validity.

The process of rebuilding the L2P table may need to read the spare areas of an enormous number of pages, leading to long recovery times. For an SSD with 512 GB capacity, assuming that each page is 32 KB and reading 32 pages in parallel takes 100 μs, reading the spare area of all the flash pages can cost up to three minutes, which is unacceptable in most situations. Therefore, optimization to reduce the recovery time has been proposed. Birrell et al. store the abovementioned per page recovery information in the last page of a block when the block is fully written [8]. This design eliminates the need to read an entire block. Bjorling et al. take partial and full checkpoints of the L2P table and persist the images in a reserved area [9]. During recovery, these images are loaded as the initial L2P table, and pages written after the latest checkpoint are subsequently remapped.

3 OPTR SSD Design

We realize the following five components internal to an SSD to achieve *strong request-level crash guarantees* without changing the standard block interface: 1) tracking the completion of write requests to ensure request atomicity, 2) tracking the coalescing between write requests, 3) periodic mapping table checkpointing, 4) tracking the availability of blocks for GC, and 5) order-preserving recovery.

3.1 Write Completion Tracking

Request atomicity is one of OPTR's guarantees. To achieve this coarse-grained atomicity, OPTR determines the completion and incompletion of each individual write request using a simple strategy: a write request that involves N pages is completed if and only if those N pages do exist in flash after a crash.

More specifically, OPTR leaves the following three kinds of clues in the spare area of each written flash page to facilitate determining the completion of a write request afterward. The first is a unique sequence number of the write request (wid, an 8-byte integer) assigned by the FTL according to the order in which the request is received. The second is the size of the write request in number of pages ($size$, a 4-byte integer). The third is the logical page number (lpn, a 4-byte integer) of the written page.

To determine which writes are completed and which are not, the recovery procedure divides flash pages into groups according to their write request identifiers (wid), counts the appearance of each wid, and then compares this count with $size$ for each write request. The entire write request is completed if and only if the count matches the request size.

3.2 Write Coalescing Tracking

Two or more write requests may coalesce in the write cache of SSDs; we refer to the involved write requests as coalesced write requests. This situation reduces the in-flash appearance count of wid of the involved write requests.

OPTR allows write coalescing to happen instead of avoiding it. To this end, OPTR detects and records each coalesced page. Each page in the write cache is tagged with a dirty flag, a wid tag, and a $size$ tag. By doing so, whenever a dirty cache page is overwritten, OPTR detects that a coalesced page exists (and anticipates that the appearance of the corresponding wid in flash decrements). For each coalesced page, OPTR records (in a DRAM buffer) the request IDs of the two involved write requests and the size of the prior write request. For example, as shown in Figure 4, a coalescing record with

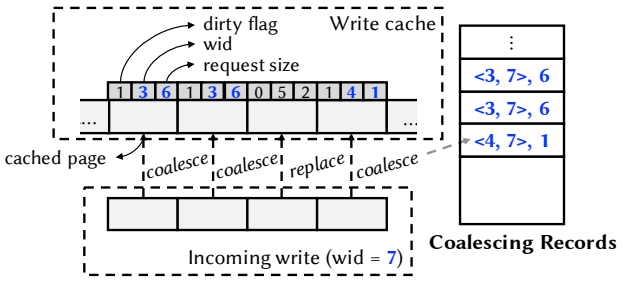


Figure 4: Write cache and generated coalescing records.

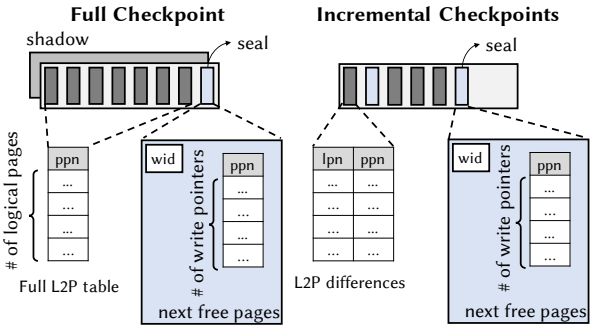


Figure 5: Full and incremental checkpoints.

"<3, 7>, 6" denotes that a prior write request whose ID is 3 coalesces with a later write request whose ID is 7, and the size of the prior write request is 6 pages. If a write involves multiple coalesced pages, multiple pieces of information are recorded. A batch of coalescing records are committed to flash when the buffer is full or when the SSD is requested to perform a flush (either issued externally by the host or internally by OPTR mechanisms).

It is hard to define the completion of each single write request that coalesces with other requests because the completion of multiple coalesced write requests needs to be atomic (which OPTR guarantees). In contrast, it is relatively simple to define incompletion as follows. Let P_i be the number of pages whose $wid = i$, D_i be the number of recorded $< x, y >$ pairs with $x = i$, and $Size_i$ be the size of the write request with $wid = i$. A coalesced write request with $wid = i$ is incomplete if $P_i + D_i < Size_i$.

3.3 Mapping Table Checkpointing

The L2P mapping table is checkpointed to flash to speed up recovery. More specifically, we reserve the first few flash blocks of each flash chip for checkpoints. OPTR keeps two types of checkpoints, a full checkpoint and several incremental checkpoints, as shown in Figure 5. The two types of checkpoints differ in that a full checkpoint snapshots the entire L2P table, while an incremental checkpoint records only the differences in the L2P table since the latest checkpoint (either full or incremental). In addition to the L2P table, both types of checkpoints record a seal page at the end of a checkpoint that includes 1) the latest wid at the time the checkpoint is made and 2) the PPNs of the next free flash pages at the time the checkpoint is made. OPTR employs incremental checkpoints by default. When the flash area for storing incremental checkpoints is full, OPTR creates a new full checkpoint and then clears the incremental checkpoints. OPTR employs a shadow for the full checkpoint to ensure its integrity, and the wid can be used to determine the recency between the full and incremental checkpoints after a crash. The PPNs of the next free flash pages allow L2P updates after the latest checkpoint to be retrievable. More specifically, for each PPN, subsequent written pages in the same block are available as pages within a block are sequentially written, and the summary page of the block can direct OPTR to the subsequent written block and so on. Then, the L2P updates can be derived from the LPN information in the spare area of these retrieved pages.

3.4 Garbage Collection

Flash pages that store outdated data are commonly considered invalid and useless, but it is these pages that OPTR leverages to rollback a disk from a crash to an order-preserved state. Thus, we enforce two constraints on the GC routine. The first constraint forbids GC from reclaiming pages programmed *after* the latest checkpoint. Since OPTR determines the completion of a write issued after the latest checkpoint by the number of pages owned by the write, reclaiming pages written prior to a flush but after the latest checkpoint would result in a flush semantics violation.

The second constraint forces an internal flush before performing GC. This flush ensures that every page being erased by GC has a stable counterpart that can always survive across a crash. To reduce the performance penalty of an internal flush, we amortize its cost by conducting GC to a batch of blocks (16 blocks in our case).

3.5 Order-Preserving Recovery

The recovery process is divided into the following phases: 1) recover the L2P table according to the latest full checkpoint, 2) sequentially incorporate the L2P differences recorded in the incremental checkpoints into the L2P table under recovery, 3) count the wid of flash pages written after the latest checkpoint according to the page pointers recorded in the latest checkpoint, and determine the completion/incompletion of each write request, 4) recover write requests after the latest checkpoint using a flow network, which we will describe shortly, and 5) sequentially incorporate the L2P changes of the write requests after the latest checkpoint.

The fourth phase above needs awareness of the order, completion, and coalescing of write requests to recover the disk to a state that satisfies our claimed guarantees. To this end, we formulate the task as a flow network problem as follows. As shown in Figure 6, write requests are represented by vertices. The order of immediately successive write requests is

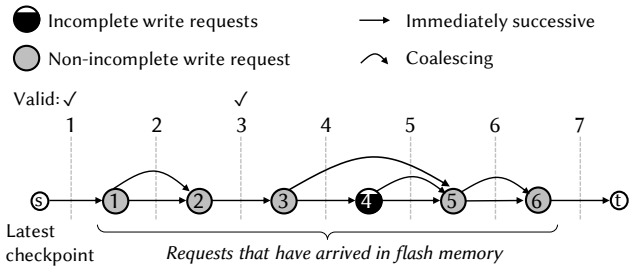


Figure 6: Determining the optimal recovery point by finding a cut in a flow network.

denoted by the straight directed edges pointing from request $wid = n$ to request wid=n+1, and coalescing between write requests is denoted by bent directed edges pointing from the earlier request to the later request. Each vertex is labeled by a wid and a Boolean value that denotes whether the write request is incomplete. Two additional vertices, s and t, are connected to the first and last vertices, respectively, to form a flow network from source (s) to sink (t), where s can be viewed as the latest checkpoint.

With such a formulation, finding a valid order-preserving recovery point is equivalent to finding an s-t cut such that 1) the cut size is equal to one (i.e., only a straight edge but not bent edges crosses the cut) and 2) the subgraph that contains vertex s (referred to as subgraph S for short) needs to contain no vertex denoting that the corresponding write request is incomplete. The rationales are as follows. The construction of the graph ensures that the write requests in subgraph S must preserve the order. A cut whose size is equal to one implies that coalesced write requests must be atomic. Since all write requests in subgraph S are not incomplete, it is valid for OPTR to recover them and drop others.

Let us take Figure 6 as an example. There are six write requests (1 to 6) present after the latest checkpoint, and some of their data pages have arrived in flash memory. The construction of the flow network reveals seven possible s-t cuts, 1 to 7, each representing a potential recovery point. Among the seven cuts, only 1 and 3 are valid. Cuts 5 to 7 are invalid recovery points because request 4 is incomplete, which breaks the order-preserving guarantee. Cuts 2 and 4 are invalid recovery points because they both cause coalesced requests to tear apart. Let us take Cut 4 as an example. Since some pages of request 3 are coalesced by request 5, request 3 cannot exist alone without request 5.

Note that OPTR can also handle multiple coalesces to the same page by creating multiple curved edges in the flow network. Again, take Figure 6 as an example. Consider a scenario where request 5 modifies a dirty cached page written by request 3, and then request 6 modifies the same page while this page is still dirty in the cache. This scenario would result in one curved edge from 3 to 5 and one from 5 to 6, as shown in Figure 6.

The optimal order-preserving recovery point should include as many write requests as possible. Therefore, finding this point is equivalent to finding the abovementioned s-t cut with the maximal subgraph S. A naive algorithm to find the optimal order-preserving recovery point is to start from a subgraph S containing only vertex s and gradually add vertices to subgraph S from left to right, one vertex at a time. Each time a vertex is added, the cut is checked to determine whether its size is equal to one and no request in S is incomplete. By doing so, the optimal recovery point is available.

In addition to the naive approach, we use a more efficient algorithm when implementing OPTR as follows. Let wid_{inc} be the wid of the earliest incomplete write and $\mathcal{C}$ be the set of all coalescing records generated after the latest checkpoint. As in Section 3.2, a coalescing record is in the form of an $< x, y >$ pair. The pseudocode is provided in Algorithm 1.

Algorithm 1 Find Optimal Recovery Point

Input: $wid_{inc}, \mathcal{C}$
Output: the optimal recovery point

```
1: wid_rec ← wid_inc;
2: Sort C by x in descending order;
3: for c ∈ C do
4:     if c.x < wid_rec ∧ c.y >= wid_rec then
5:         wid_rec ← c.x;
6:     end if
7: end for
8: return wid_rec;
```

4 Filesystem and Application Optimizations

We demonstrate Ext4 filesystem and SQLite database optimizations that exploit the benefits of OPTR SSDs as follows.

At the Ext4 level, we optimize an existing filesystem primitive, *fsync* (and its variant, *fdatasync*) and introduce a new filesystem primitive, *ffence* (and its variant, *fdatafence*). For brevity, we refer to *fdatasync* as *fsync* and *fdatafence* as *ffence* in this section. Conventional *fsync* issues two flush commands to a disk, one after *fsync* transfers a journal to the disk and the other after *fsync* transfers a commit record to the disk. Our optimized *fsync* uses the same order to transfer the journal and commit record to the disk but only issues the second flush request to the disk. Conventional *fsync* requires the first flush to prevent SSDs from persisting the commit record prior to the journal. With OPTR SSDs, which preserve order, the first flush becomes safely omissible. Note that the second flush of the optimized *fsync* still can guarantee the same durability semantics as the original *fsync* does.

The newly introduced *ffence* resembles the optimized *fsync*. This primitive also uses the same order to transfer the journal and commit record to the disk but omits both flush requests. Therefore, *ffence* is a pure barrier for applications to define

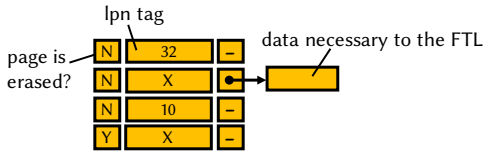


Figure 7: Emulated flash pages in VST.

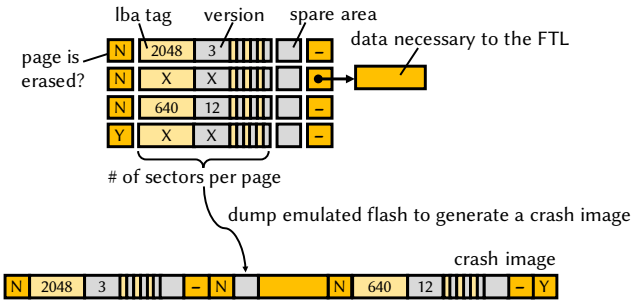


Figure 8: Extensions for crash-guarantee validation.

the required partial order of transferring write requests to disks. With OPTR SSDs and the newly introduced *ffence*, applications can use *fsync* sparingly only when immediate durability is needed.

At the SQLite level, we modify SQLite to use *ffence* whenever a barrier alone is sufficient and to use the optimized *fsync* when immediate durability is required. To commit a single INSERT transaction, SQLite (version 3.19) calls *fsync* four times in its default configuration (journal_mode=DELETE and synchronous=FULL) [1]. Since the default configuration fails to guarantee durability on Ext4 (because unlink is not synchronous [3]), we set the configuration to the synchronous=EXTRA mode, which incurs one additional *fsync* on the parent directory after unlink to persist the deletion [4]. The objectives of the first four *fsync*s are as barriers instead of requesting immediate durability. The first *fsync* is a barrier after writing the content of a rollback journal file. The second *fsync* is a barrier to enforce that the rollback journal file exists in the directory before the db file is modified. The third *fsync* is a barrier after writing the header of the rollback journal. The fourth *fsync* is a barrier after writing the db file. Therefore, we change these four *fsync*s to *ffence*s. For the fifth *fsync*, we change it to the optimized *fsync*.

Some applications and filesystems, such as SQLite and OptFS, do not strictly demand immediate durability. For example, SQLite developers deliberately choose to avoid the fifth *fsync* by default and to allow loss of durability following a power loss event [3]. In this case, the *OPTR-enabled no-barrier mode* is the best solution.

5 OPTR Design Validation

Validating the functional correctness and crash guarantees of an FTL poses two challenges to FTL designers. First, the validation process is inherently time consuming since it requires extensive stress tests to create a large number of crash points. FTL operations such as GC, which are not invoked until a sufficient number of writes occur, make the issue even worse. Second, one may lack necessary hardware support during development of a new FTL. For example, the OpenSSD platform [2] on which we implement OPTR does not allow access to the spare area of flash, which OPTR demands during recovery.

To address this issue, we extend an FTL testing framework named Virtual Stress Testing (VST) [31] and use it to validate our OPTR implementation. Please note that we still evaluate the performance on real OpenSSD hardware [2]. We use the same FTL code in the two experimental environments and make slight modifications to overcome hardware limitations (e.g., accessing the spare area).

5.1 VST Testing Framework

VST [31] is a simulation framework designed for validating the functional correctness of FTLs. This framework enables one to execute FTLs on PCs or servers that emulate the SRAM, DRAM and flash required by the FTLs. VST outperforms traditional FTL stress tests that use real SSDs by orders of magnitude. To validate an FTL, VST issues an extensive number of read and write requests to the FTL and reports a bug if the FTL violates any predefined rule such as incorrect page contents or a nonsequential program within a block.

The core data structure of VST is the emulated flash memory. As shown in Figure 7, an emulated flash page contains three fields: an $erased$ field that denotes whether a page has been erased, an lpn tag that records the LPN of the page, and a $data$ pointer that points to the FTL metadata stored in the page. The design of storing an lpn tag for host data rather than the actual contents dramatically decreases the memory requirement for simulation and speeds up the test. Thus, VST is particularly suitable for our validation purposes, as we want to generate a very large number of crash images.

5.2 Extending VST for Crash Recovery Tests

We largely extend VST to support crash recovery tests, which the original VST does not touch at all. For example, Figure 8 depicts the extensions we made to the emulated flash memory. We divide each emulated page into multiple 512-byte sectors to match the granularity of IO requests and further add a $version$ attribute to each emulated sector. The $version$ of a sector starts with 0 and increments by 1 when the sector is updated. For crash simulation, we fork a separate thread to periodically take snapshots of the emulated flash memory and store the snapshots as crash images.

The process of validating whether an FTL obeys prefix semantics and request atomicity is described as follows. First, we compile the FTL under test to a Linux shared object, execute the FTL on the extended VST, drive the FTL using

a trace file, and generate a number of crash images. Then, VST deserializes each crash image back into the form of the emulated flash memory and triggers the recovery procedure of the FTL. After recovery is done, VST queries the firmware for the last write it has recovered, which we call the *recovery point*, and replays the same trace separately until the recovery point to construct the *golden disk*, which represents the result of bug-free and crash-free execution. Finally, for each written sector in the golden disk, VST issues a read targeting the sector and checks if the LBA and version returned by the FTL match with those in the golden disk. If any inconsistency exists, our extended VST framework reports a violation.

We can also validate whether the FTL obeys flush semantics. We insert flushes into the request sequence and record the ID of the last flushed write in the header of a crash image. If the FTL fails to recover any write prior to the last flush, our VST framework also reports a violation.

5.3 Validation Results

We select 12 write-heavy traces from the MSRC I/O traces [35], which cover a variety of server-level access patterns [46], to drive the FTL tests. Our validation process consists of three runs. In the first run, we execute the firmware without crashes being simulated to validate its functional correctness. Similar to the validation approach in [31], we repeat each trace until the write amount reaches 1 TB. In the second run, we let the firmware recover from a total of 2,400 crash images created without any flush issued to validate whether the FTL correctly obeys prefix semantics and request atomicity. The final run is similar to the previous run, but we additionally insert a flush for every 1,000 writes to validate whether the FTL also obeys the flush semantics.

Table 1 shows the final validation results. Our implemented OPTR passes all three runs of tests. Please note that these results also suggest that our extended VST simulation framework passes the stress test, and this finding makes the simulator a much more reliable platform for future research.

Table 1: Validation results for our OPTR FTL.

	Functional Correctness	Prefix Semantics Preserved	Flush Semantics Obeyed
1-TB stress test	V		
2400 images w/o flushes		V	
2400 images w/ flushes		V	V

6 Evaluation

6.1 Experimental Setups

We implement the OPTR FTL on a real SSD (OpenSSD [2]) with an ARM7 core at 87.5 MHz, 96 KB on-chip SRAM, 64 MB DRAM, and 128 GB flash memory. We organize a total of 16 flash chips into four channels, each channel connects to four chips, each chip contains approximately 4,000 blocks, and each block contains 128 16-KB pages. The SSD adopts a static allocation strategy, which allocates a logical page to a certain flash chip based on modulo [22]. We allocate 8 MB of DRAM as the write cache of the SSD and adopt the LRU cache replacement policy. We adopt the greedy policy for GC [29, 42]. We store the per page recovery information (i.e., the wid, $size$ and lpn described in Section 3.1) in the last page of each flash block. For the OPTR-specific settings, we reserve 32 blocks for full checkpoints and 32 blocks for incremental checkpoints and coalescing records.

All the experiments are performed on a server with a 6-core Intel i7-8700 CPU and 32 GB DRAM running Ubuntu 16.04. We erase the entire SSD before conducting each experiment. More experimental setups are described in the captions of each experimental results figure.

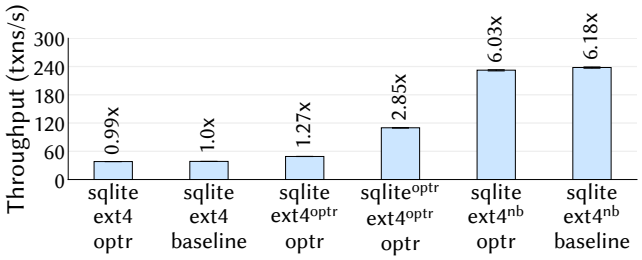


Figure 9: Performance of the three OPTR usage modes. In addition to the three usage modes and the baseline, we plot two additional bars: The leftmost bar represents running unmodified SQLite and Ext4 on OPTR to demonstrate OPTR's overhead for enforcing ordering. The rightmost bar represents running unmodified SQLite and Ext4 mounted with the no-barrier option; this configuration guarantees neither durability nor consistency and is for demonstrating the upper bound.

6.2 System-Level Performance

In this set of experiments, we evaluate three usage modes of OPTR as mentioned previously. The first mode (② in Figure 2) runs unmodified SQLite and the optimized version of Ext4; the second mode (③ in Figure 2) runs the optimized versions of SQLite and Ext4 (details of the optimizations are described in Section 4). The third mode is the OPTR-enabled no-barrier mode (④ in Figure 2). This mode guarantees consistency but achieves only eventual durability instead of immediate durability.

We use a microbenchmark that keeps generating transactions for a time interval of five minutes. Each transaction inserts a key-value pair into the SQLite database. Figure 9 shows the throughput performance of the three usage modes of OPTR. Removing the first flush of fdatasync (the third bar) improves the performance by 1.27×; invoking fdatafences (the fourth bar) for ordering constraints yields 2.85× im-

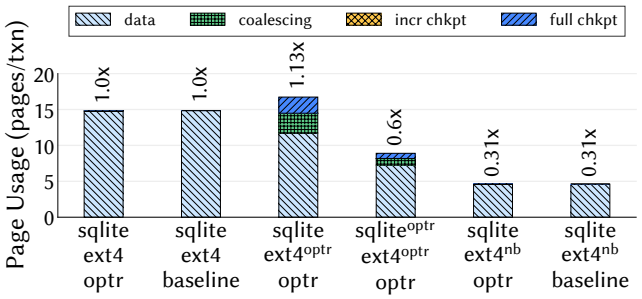


Figure 10: Endurance impact of the three OPTR usage modes.

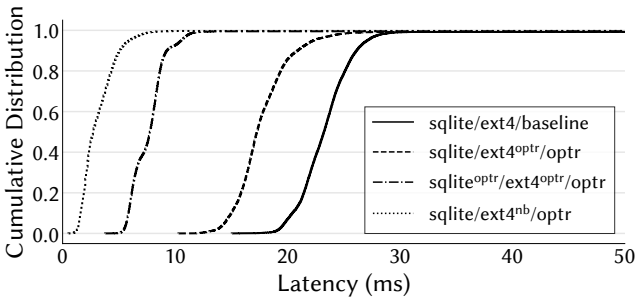


Figure 11: Latency of the three OPTR usage modes.

provement; finally, the OPTR-enabled no-barrier mode performs 6.03× better than the baseline, and this result is very close to the upper bound of 6.18×. We conduct the experiments ten times, and the errors are within ± 0.7%. To quantify the FTL overhead of enforcing the order, we compare unmodified SQLite and Ext4 atop OPTR SSDs (the first bar) with unmodified SQLite and Ext4 atop baseline SSDs (the second bar). The overhead is unnoticeable (hidden by the flushes of the workload). This result is consistent with our analysis in Section 7. However, we would like to emphasize that OPTR is not overhead-free (as shown in Section 7, OPTR can decrease the performance by up to 11.1% in some synthetic workloads).

In terms of SSD endurance impact, OPTR may improve endurance because removing flushes results in a greater chance of write coalescing; OPTR may also hurt endurance because OPTR stores checkpoints and coalescing records into flash memory for the sake of crash recovery. Figure 10 shows the overall endurance impact of the three OPTR usage modes. The y-axis denotes the average number of flash pages written per transaction. We break down written pages into user data (data) and OPTR-related data (i.e., incremental checkpoints, full checkpoints, and coalescing records).

The first usage mode of OPTR (the third bar) slightly increases the number of written pages per transaction (1.13×). The second usage mode (the fourth bar) lowers the frequency of flush operations and thereby decreases the number of written pages per transaction to 0.6×. The third usage mode does not incur any flush, and thus, the number of written pages is 0.31× that of the baseline.

Figure 11 shows the cumulative latency distribution of transactions. Flushes are slow in nature, and thus, partially or fully removing them is expected to achieve shorter latency. The experimental results show that the average latencies of OPTR in the first, second and third usage modes are 0.78×, 0.35× and 0.17× that of the baseline, respectively.

7 FTL Analyses

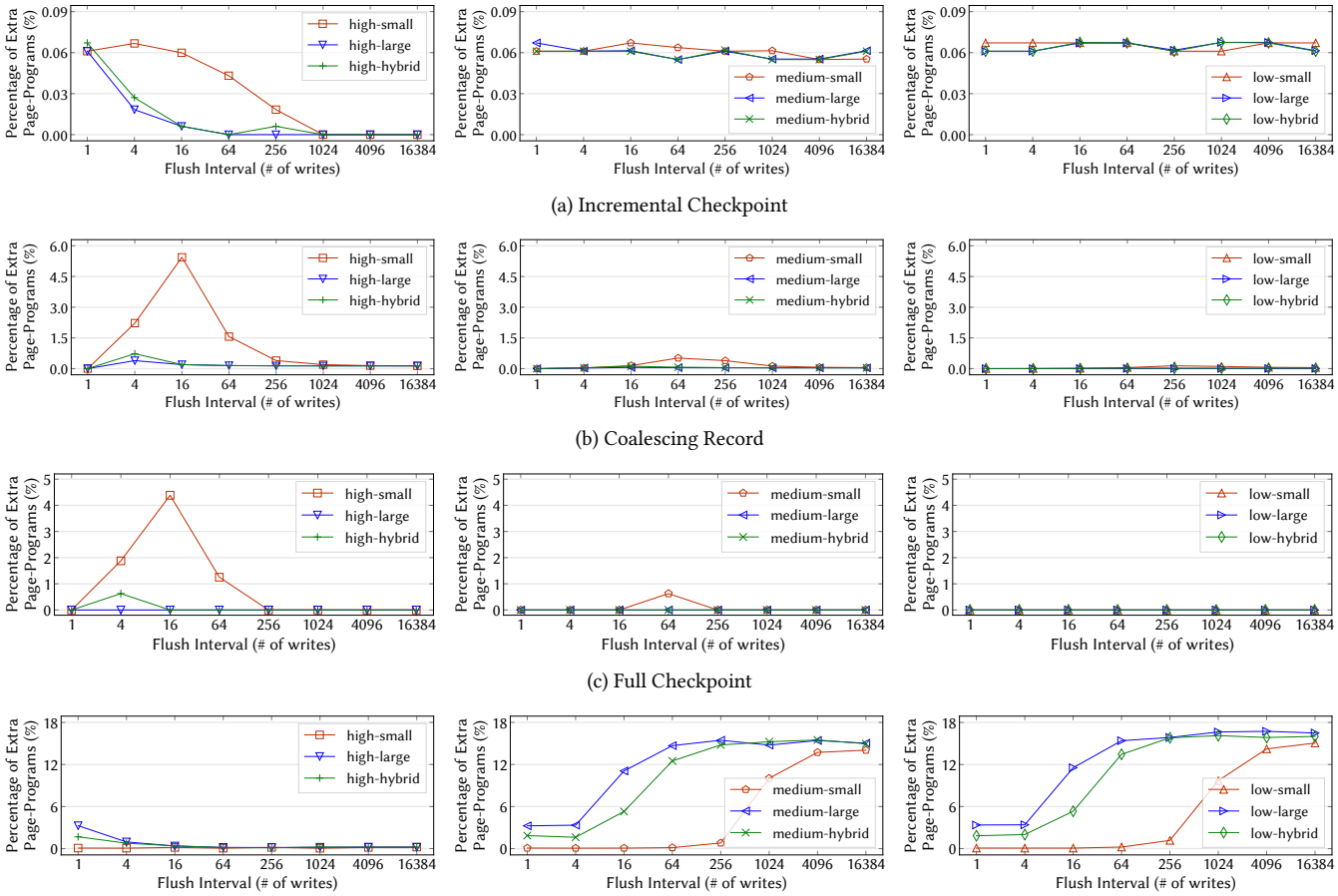
In this section, we analyze the FTL in more detail. We first report the extra flash page-programs caused by OPTR. Next, we analyze the extra GC constraints and report the overhead in terms of throughput performance. The above experiments are conducted using synthetic workloads that comprise random writes with different access localities (high: 1 MB footprint, medium: 80 MB footprint, and low: 1 GB footprint), different access granularities (large: 1 MB, small: 16 KB, and hybrid: half large and half small), and different flush intervals (1, 4, ..., to 16,384 writes). Note that these analyses are pessimistic because OPTR does not benefit from reducing flush requests in the synthetic workloads. Finally, we discuss the memory overhead (i.e., extra SRAM, DRAM, and flash space) caused by OPTR, and then, we estimate the recovery time.

7.1 Extra Flash Page-Programs

OPTR incurs extra flash page-programs because of incremental checkpoints, coalescing records, full checkpoints, and internally invoked flushes.

The overhead of **incremental checkpoints** is negligible (<0.075%), as shown in Figure 12a. The overhead is low because writing each user data page (e.g., 16 KB) incurs only an 8-byte L2P table change record, which constitutes incremental checkpoints. The overhead of incremental checkpoints is even low if write requests coalesce in the write cache. For example, the overhead is less than 0.01% for the workload with high access locality and a long flush interval (e.g., the *high-large* workload with a flush interval greater than 16 writes).

The overheads of **coalescing records** and **full checkpoints** are also negligible for most workloads. An exception is the workload with high locality and small access granularity (i.e., *high-small*) when the flush interval is between four and 64 writes (Figures 12b and 12c). High write locality tends to incur write coalescing. With more coalescing records, the checkpoint area is filled quickly, and OPTR invokes full checkpoints more frequently. If the flush interval is one write, flushes happen to suppress the occurrence of write coalescing. If the flush interval is long (e.g., >256 writes) or if the size of the write granularity is large, the overhead decreases because approximately 800 coalescing records result in an extra 16 KB page-program (each coalescing record is 20 bytes in our implementation).



(a) Incremental Checkpoint

(b) Coalescing Record

(c) Full Checkpoint

(d) OPTR-Incurred Flush

Figure 12: Page write overhead analysis using synthetic workloads. The prefix in the legends indicates the locality: *high*/*medium*/*low* access 1 MB/80 MB/1 GB of flash. The suffix indicates the access granularity: *small*/*large* write 16 KB/1 MB of data per request, and *hybrid* writes 16 KB or 1 MB of data per request, each with a 50% chance.

The percentages of page-programs caused by **internally invoked flushes** are shown in Figure 12d. In our OPTR implementation, the FTL internally flushes the write cache under the following four conditions: before an incremental checkpoint, before a full checkpoint, before GC, and every five seconds. The first three types of flushes are for correct recovery in case a crash happens, and the fourth type of flushes is for bounding the longest time duration for which dirty data can reside in the write cache. We observe a reasonable trend: if the flush interval is short (e.g., every four writes), the page-programs caused by internally invoked flushes only account for a small percentage (less than 4%). In comparison, if the flush interval is long (e.g., every 64 writes), the page-programs caused by internally invoked flushes account for a high percentage (up to 17%). Note that the page-programs analyzed here are for user data instead of OPTR's metadata, and baseline SSDs also need to flush their write caches at some points. Therefore, the above analysis is a conservative overestimation of overhead.

7.2 GC Constraints

Although OPTR constrains GC from selecting recently written flash blocks to guarantee recoverability, this constraint does not cause significantly adverse effects. The reasons are two-fold. First, OPTR constrains GC from selecting flash blocks written after the latest full or incremental checkpoint. According to our analysis, this type of blocks accounts for only 0.2% of the total blocks on average. Second, sophisticated GC policies such as age-aware GC policies avoid selecting recently written blocks in the expectation of data invalidation due to write locality.

7.3 Performance Overhead

Figures 13a, 13b, and 13c show the throughput performance (MB/s). Dotted lines denote the throughput of the baseline, and solid lines denote that of OPTR. Overall, OPTR incurs less than 2% performance overhead on average and up to 11.1% performance overhead for the synthetic workloads. Note that OPTR can slightly outperform the baseline for some workloads when OPTR's mechanisms happen to match the

workload characteristics. For example, for workloads with a high locality, age-aware GC is better than greedy GC. OPTR happens to achieve some effects of age-aware GC because OPTR constrains GC from selecting recently written flash blocks.

7.4 Memory Overhead

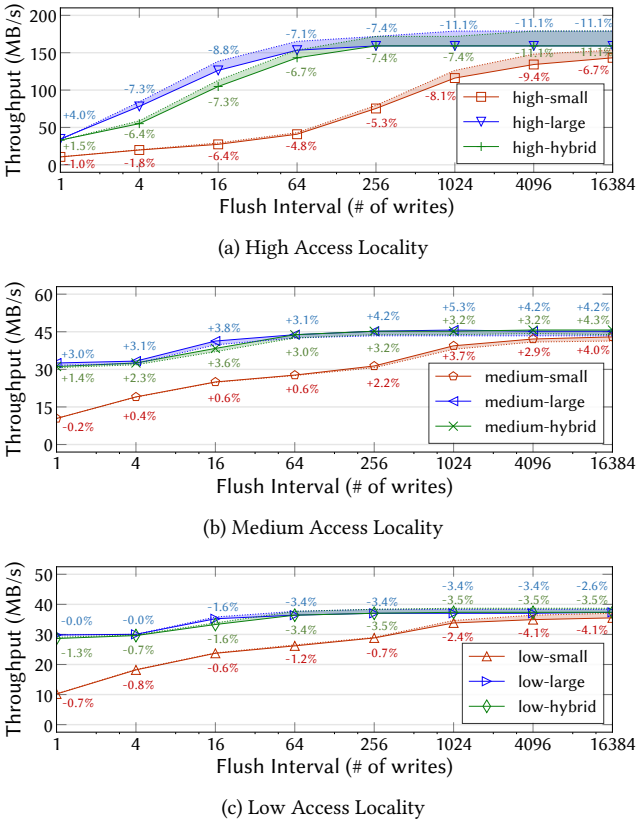
The DRAM and SRAM overheads for implementing OPTR are analyzed as follows. First, OPTR associates each entry in the write cache with two extra attributes, wid (8 bytes) and $size$ (4 bytes). In our implementation, the write cache is 8 MB (16 KB $\times$ 512 entries). Therefore, the space for storing wid is 8 $\times$ 512 = 4 KB (in DRAM in our implementation), and the space for storing $size$ is 4 $\times$ 512 = 2 KB (in SRAM in our implementation). Second, we employ a 256 KB DRAM area to buffer incremental checkpoints. Third, OPTR stores coalescing records in a DRAM buffer, whose size is equal to a flash page, e.g., 16 KB. Finally, the memory space for crash recovery can overlay that for regular FTL operation, so we do not consider this space as additional overhead.

The flash memory overhead for implementing OPTR is described as follows. First, OPTR additionally stores wid and $size$ (8 bytes and 4 bytes, respectively) in the spare area of flash memory. Modern flash memory with 16 KB page size equips each page with a 2,208-byte spare area (i.e., page size = 16,384 + 2,208 = 18,592 bytes) [6]. Since OPTR only consumes 12 bytes out of the 2,208 bytes, the overhead is only 12/2,208=0.5%, and the impact on the ratio of the error correction code rate is only 16,384/18,580 - 16,384/18,592 = 0.06%. Second, OPTR stores the lpn of each flash page in the spare area and summarizes the lpn of a block of pages in the last page of the block. We anticipate that baseline SSDs also do so. Third, OPTR keeps incremental checkpoints, coalescing records, and full checkpoints in flash memory. We set the area for storing incremental checkpoints and coalescing records to 64 MB. The size of a full checkpoint is equivalent to that of the L2P table. Given an SSD with 128 GB flash memory and 16 KB pages, the size of a full checkpoint and its shadow is at most 128 GB/16 KB $\times$ 4 B $\times$ 2 = 64 MB.

7.5 Recovery Time

OpenSSD does not provide an access approach to the spare area of flash pages, so we are not able to measure recovery time using OpenSSD. Instead, we conduct a worst-case estimation. We anticipate that accessing flash memory, which is slower than SRAM, DRAM, and computation, dominates the recovery time.

Recovery begins with reading the full checkpoint, whose size is approximately 32 MB (2,048 pages) for a 128 GB SSD. Second, OPTR sequentially scans the 64 MB area (4,096 pages) for storing incremental checkpoints and coalescing records. Third, OPTR scans the summary pages programmed after the latest checkpoint. Given a 256 KB buffer for incremental checkpoints, the number of summary pages after the latest checkpoint can be up to 256. Finally, each flash chip can have a partially written block, and OPTR needs to access these pages during recovery. We assume that the number of flash pages of partially written blocks is 4,096. Overall, there are 10,496 pages. We conservatively assume that reading each flash page costs 100 μs and no internal parallelism is available; then, the worse-case recovery time is 10,496 $\times$ 100 μs = 1 s, which is acceptable in general.



(a) High Access Locality

(b) Medium Access Locality

(c) Low Access Locality

Figure 13: Performance overhead for preserving write order.

8 Related Work

OptFS [15], BarrierFS [45], and Featherstitch [23] are related filesystem and IO stack works that propose to decouple ordering from durability. Unlike this work, they do not focus on the FTL design and recovery process of SSDs. OptFS presents a filesystem that requires disks to support asynchronous durability notification, which notifies a host when certain blocks become persistent. BarrierFS presents a filesystem and an IO stack that require disks to support the cache barrier command, which is only available in a few eMMC (embedded multimedia card) products [45] but unavailable in the standard block device interfaces of off-the-shelf SATA, SAS, and NVMe SSDs. BarrierFS [45] implements an FTL to support the barrier write command in a commercial eMMC product. OPTR is complementary to both OptFS and BarrierFS and can simplify their designs. For instance, OptFS requires checksum encoding/decoding to enforce the ordering con-

straint between journal metadata and the commit record, and BarrierFS uses explicit cache barrier primitives to declare order; with OPTR, both can be simplified out because OPTR implicitly preserves the write order.

Mime [13], the Logical Disk [21], Stasis [43], TxFlash [40], Beyond Block IO [37], Mars [19], LightTx [32], X-FTL [28], and Isotope [44] propose transactional storage with full or partial supports to ACID at the disk level. The notion of transaction is stronger than OPTR's guarantees, but these works all propose changing the standard block device interface, which inevitably incurs significant software stack changes. Among these works, Mime [13] also advocates the benefits of request atomicity and ordering, which are in line with OPTR's design. Compared with OPTR, Mime [13] is fundamentally different because it is HDD design instead of SSD design. In addition, Mime does not allow write coalescing in a write cache, which OPTR can handle.

Increasing the disk-level crash guarantee not only can improve system performance but also, more importantly, helps to reduce crash vulnerabilities. In this sense, just replacing baseline SSDs with OPTR SSDs is beneficial. Crash vulnerabilities are serious problems. Pillai et al. [39] find 60 application-level crash vulnerabilities in widely used applications such as LevelDB and Git. Zheng et al. [47] also find ACID violations in many database systems. These vulnerabilities are mainly caused by the weak and vague crash guarantees provided by the underlying filesystems. Thus, Bornholt et al. present crash-consistency models [10], and Pillai et al. [39] specify a set of persistent properties. These approaches aim to confine and standardize the crash behaviors of filesystems, and OPTR can help to achieve these aims.

Several studies have attempted to reduce the usage or overhead of flush operations at the application or filesystem level. BarrierFS [45] and [5] replace *unnecessary flushes* with cache barrier commands. Our previous work-in-progress report [12] proposes to directly omit *unnecessary flushes* by considering *order-preserving SSDs* that achieve *strong request-level crash guarantees*. iJournaling [38] performs fine-grained journaling per file to mitigate the interference between fsync-intensive threads. NoFS [16] proposes backpointer-based consistency to fully eliminate ordering constraints but at the cost of not being able to implement atomic operations (e.g., rename). Xsyncfs [36] introduces external synchrony, in which a write is not immediately persisted (i.e., asynchronous writes) unless an external observer sees the write; thereby, this method provides the simplicity of synchronous writes and approaches the performance of asynchronous writes. Chen et al. identify the sync amplification issue in virtualized environments and propose solutions using journaling techniques at the virtual-disk level [14].

One can equip SSDs with supercapacitors, which may help to preserve write order. However, they are not widely adopted. In previous work [48], power interrupt tests are performed on 15 SSDs, and only four out of the 15 tested SSDs are equipped with supercaps. In addition, supercapacitors may be sufficient to protect an FTL from corruption, but they may not be sufficient to preserve write order. Among the four tested supercapped SSDs in [48], two still exhibit shorn or unserializable writes under power faults. Finally, capacitors are sensitive to temperature- and aging-related degradation and failures [25].

9 Conclusion

In common practice, consumer-grade SSDs (whose write cache is not battery-backed) cannot guarantee the order and atomicity of write requests upon a crash because SSD performance optimization strategies including write caching, write coalescing, request scheduling, and parallel flash programming all tend to break the guarantee. The lack of a strong crash guarantee at the disk level complicates the design of applications and filesystems and degrades the overall system performance. By exploiting the fact that SSDs adopt out-of-place updates, this work proposes order-preserving translation and recovery design (OPTR) that maintains SSD performance while preserving an illusion that write requests are completed in order and atomically. We realize the required address translation, garbage collection, and crash recovery techniques internal to a real SSD to achieve OPTR. We also develop a functional simulator to validate the correctness of OPTR.

Three usage modes of OPTR SSDs are identified and evaluated: 1) optimizing filesystems to remove the *unnecessary flushes* of *fdatasync*, 2) optimizing both applications and filesystems to further replace *fdatasync* with *fdatafence* primitives, and 3) combining the performance advantages of the no-barrier mode of filesystems and the *strong request-level crash guarantees* of OPTR SSDs. Real system experiments based on SQLite, Ext4, and a real SSD show that these three modes achieve 1.27×, 2.85×, and 6.03× performance improvement, respectively.

This work is the first SSD work with *strong request-level crash guarantees* and the standard block device interface. In comparison with previous works on transactional SSDs, we change the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the standard block interface. We anticipate that OPTR can inspire more future application and filesystem designs.

Acknowledgments

We thank our shepherd, Youjip Won, and the anonymous reviewers for their valuable feedback. This research is supported in part by NOVATEK Fellowship and in part by the Ministry of Science and Technology (MOST) of Taiwan under grants 108-2218-E-007-023, 108-2218-E-007-021, and 107-2218-E-007-001. We also thank the National Center for High-performance Computing (NCHC) of Taiwan for computer time and facilities.

References

[1] Atomic commit in SQLite. https://www.sqlite.org/atomiccommit.html.

[2] OpenSSD (www.openssd-project.org/).

[3] Potential bug in crash-recovery code: unlink() and friends are not synchronous. http://sqlite.1065341.n5.nabble.com/Potential-bug-in-crash-recovery-code-unlink-and-friends-are-not-synchronous-td68885.html.

[4] Pragma statements supported by SQLite. https://www.sqlite.org/pragma.html#pragma_synchronous.

[5] Using cache barriers in lieu of REQ_FLUSH | REQ_FUA for eMMC 5.1. https://www.spinics.net/lists/linux-ext4/msg48992.html.

[6] SpecTek NAND flash detail: TLC 8192Gb. https://www.spectek.com/menus/flash_detail.aspx?memType=TLC+8192Gb, 2019.

[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC '08)*, Boston, Massachusetts, USA, 2008.

[8] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, April 2007.

[9] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, California, USA, 2017.

[10] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, Georgia, USA, 2016.

[11] Yu-Ming Chang, Ping-Hsien Lin, Ye-Jyun Lin, Tai-Chun Kuo, Yuan-Hao Chang, Yung-Chun Li, Hsiang-Pang Li, and KC Wang. An efficient sudden-power-off-recovery design with guaranteed booting time for solid state drives. In *Proceedings of the 8th IEEE International Memory Workshop (IMW '16)*, Paris, France, 2016.

[12] Yun-Sheng Chang and Ren-Shuo Liu. Improving the performance of SQLite and Ext4 using order-preserving SSDs. In *17th USENIX Conference on File and Storage Technologies Work-in-Progress Reports (FAST WiPs '19)*, Boston, Massachusetts, USA, 2019.

[13] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, Er Stepanov, John Wilkes, Richard Wagner, and Scene I. Mime: A high performance parallel storage device with strong recovery guarantees. Technical Report HPL–CSP–92–9 rev 1, Hewlett-Packard Laboratories, 1992.

[14] Qingshu Chen, Liang Liang, Yubin Xia, Haibo Chen, and Hyunsoo Kim. Mitigating sync amplification for copy-on-write virtual disk. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST '16)*, Santa Clara, California, USA, 2016.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, Pennsylvania, USA, 2013.

[16] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, California, USA, 2012.

[17] Jin-Yong Choi, Eyee Hyun Nam, Yoon Jae Seong, Jin Hyuk Yoon, Sookwan Lee, Hong Seok Kim, Jeongsu Park, Yeong-Jae Woo, Sheayun Lee, and Sang Lyul Min. Hil: A framework for compositional FTL development and provably-correct crash recovery. *ACM Trans. Storage*, 14(4):36:1–36:29, December 2018.

[18] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, 2012.

[19] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, Pennsylvania, USA, 2013.

[20] Jonathan Corbet. Barriers and journaling filesystems. https://lwn.net/Articles/283161/, 2008.

[21] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, North Carolina, USA, 1993.

[22] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and

Myoungsoo Jung. Exploiting intra-request slack to improve SSD performance. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, 2017.

[23] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007.

[24] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2), May 2000.

[25] Feng Gao and Dave Verburg. SSD's reliability failure mode and it's supercapacitor failure. In *Proceedings of South East Asia Technical Conference on Electronics Assembly Technologies*, Penang, Malaysia, 2016.

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13)*, San Jose, California, USA, 2013.

[27] Myoungsoo Jung, Ellis H. Wilson, III, and Mahmut Kandemir. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA '12)*, Portland, Oregon, USA, 2012.

[28] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD '13)*, New York, New York, USA, 2013.

[29] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the 1995 USENIX Technical Conference (TCON '95)*, New Orleans, Louisiana, USA, 1995.

[30] Dongwook Kim, Youjip Won, Jaehyuk Cha, Sungroh Yoon, Jongmoo Choi, and Sooyong Kang. Exploiting compression-induced internal fragmentation for power-off recovery in SSD. *IEEE Transactions on Computers*, 65(6):1720–1733, June 2016.

[31] Ren-Shuo Liu, Yun-Sheng Chang, and Chih-Wen Hung. VST: A virtual stress testing framework for discovering bugs in SSD flash-translation layers. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '17)*, Irvine, California, USA, 2017.

[32] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD '13)*, Asheville, North Carolina, USA, 2013.

[33] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[34] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Trans. Comput.*, 60(5):653–666, May 2011.

[35] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, November 2008.

[36] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3):6:1–6:26, September 2008.

[37] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA '11)*, Washington, DC, USA, 2011.

[38] Daejun Park and Dongkun Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, California, USA, 2017.

[39] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, USA, 2014.

[40] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, USA, 2008.

[41] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[43] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, USA, 2006.

[44] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: ACID transactions for block storage. *ACM Trans. Storage*, 13(1):4:1–4:25, February 2017.

[45] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, California, USA, 2018.

[46] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, California, USA, 2013.

[47] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, USA, 2014.

[48] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, California, USA, 2013.