Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer

Yun-Sheng Chang Yao Hsiao Tzu-Chi Lin Che-Wei Tsao Chun-Feng Wu
Yuan-Hao Chang Hsiang-Shang Ko Yu-Fang Chen

Institute of Information Science, Academia Sinica, Taiwan

Abstract

This paper introduces the design of a snapshot-consistent flash translation layer (SCFTL) for flash disks, which has a stronger guarantee about the possible behavior after a crash than conventional designs. More specifically, the flush operation of SCFTL also has the functionality of making a "disk snapshot." When a crash occurs, the flash disk is guaranteed to recover to the state right before the last flush. The major benefit of SCFTL is that it allows a more efficient design of upper layers in the storage stack. For example, the file system built on SCFTL does not require the use of a journal for crash recovery. Instead, it only needs to perform a flush operation of SCFTL at the end of each atomic transaction. We use a combination of a proof assistant, a symbolic executor, and an SMT solver, to formally verify the correctness of our SCFTL implementation. We modify the xv6 file system to support group commit and utilize SCFTL's stronger crash guarantee. Our evaluation using file system benchmarks shows that the modified xv6 on SCFTL is 3 to 30 times faster than xv6 with logging on conventional FTLs, and is in the worst case only two times slower than the state-of-the-art setting: the ext4 file system on the Physical Block Device (pblk) FTL.

1 Introduction

In modern computer systems, data storage usually needs to go through multiple layers, starting from a specific application, going through the file system, and eventually reaching the physical device. Usually, we refer to those layers as the *storage stack*. The design of a correct storage stack is non-trivial. For instance, in order to maintain efficiency, I/O operations sending from one layer can be reordered to reduce the overall waiting time. More importantly, under a sudden power loss or a system crash, a storage system should correctly recover the stored data. Due to the high complexity of the system design, in recent years, a significant amount of research effort is devoted to applying formal methods to provide rigorous correctness guarantees about storage stacks. Among those, one crucial direction is proving crash safety [3, 9, 10, 12, 13, 42].

Crash recovery is a critical issue; no one wants to lose important data after one accidental power loss. In the state-of-the-art system design, each component in the storage stack has its crash recovery mechanism and usually makes only minimal assumptions about its lower layers. For instance, the file system usually assumes the underlying physical device follows the *asynchronous disk model*. The model provides only minimal guarantees about its possible behavior when the system crashes. As a result, the file system needs to implement a heavyweight crash recovery mechanism.

The more recent design of physical devices offers much stronger guarantees while maintaining similar performance. For instance, the *prefix-preserving* disk model [11] guarantees to recover to some state after the last flush operation without operation reordering. The *snapshot-consistent* disk model we propose in this paper provides an even stronger guarantee. The advance in the physical device design provides us with an excellent opportunity to rethink the design of the entire storage stack. The stronger guarantees of the physical device enable a cleaner and more efficient design of file systems, database systems, and applications built on top of them. For instance, the upper layer can remove some write barriers and data replication to achieve higher performance.

In this paper, we introduce the *snapshot-consistent flash translation layer* (SCFTL). The *flash translation layer* (FTL) is the interface between flash memory and upper layers in the storage stack, providing operations such as write, read, and flush. As the name suggests, SCFTL implements the snapshot-consistent disk model, which ensures that a crashed disk will recover to the state right before the last flush operation.

Our snapshot-consistent disk model has the benefit that the flush operation can be used to take a "disk snapshot." The feature is particularly useful for upper layers to implement atomic operations/transactions—they only need to invoke a flush at the end of each operation/transaction. Upper-layer systems can utilize this feature to obtain a more efficient design, e.g., removing the journal from a file system.

Next, we compare the snapshot-consistent disk model with other disk models used in the literature. At first glance, one might think that the *synchronous disk model* can provide a similar crash guarantee, as it also confines the number of post-crash states to one. The synchronous disk model, however, can ensure only the atomicity of a single disk write, whereas our proposed snapshot-consistent disk model guarantees the atomicity of multiple writes between two consecutive flushes.

The asynchronous disk model guarantees only that writes before a flush are durable. For those after the last flush operation, even the order is not guaranteed. In the worst case,

it might have 2^n post-crash states, where n is the number of writes after the last flush. The *prefix-preserving disk model* guarantees that writes after the last flush will not be reordered, so the possible post-crash states reduce to, in the worst case, n. The post-crash states of the two disk models are "non-deterministic," and the upper layers have to consider all possible scenarios in their crash recovery mechanisms.

Our SCFTL allows an efficient implementation (§3) utilizing the *out-of-place* update feature of FTLs. An FTL usually maintains an in-memory *logical-to-physical* address translation table (or L2P for short). On a write of data d to a logical address l, due to the physical constraint of the flash memory, the FTL cannot just update the value pointed to by l to d. Instead, it finds a new physical location, puts d there, and updates L2P to remap l to that new location. The old data pointed to by l remains there. The main idea of our implementation is to remember the L2P right before the last flush operation, which we refer to as the *stable L2P*. When the system crashes, we use the stable L2P to recover to the state before the last flush. Writing the entire L2P to the flash memory is an expensive operation, so we design a mechanism to store only the changes to the last stored L2P table, and store the full L2P to the flash memory only occasionally. We also designed a mechanism to ensure that the garbage collector will not erase the data pointed by the stable L2P.

We have formally verified the correctness of our SCFTL implementation using a combination of a proof assistant, a symbolic executor, and an SMT solver, which achieves a good balance between the degree of automation and the expressive power for stating and proving desired properties. The formal framework is set up manually using an interactive proof assistant, while the proof obligations involving the detail of SCFTL are discharged automatically with an SMT solver.

In the formal framework, we start with a simple mathematical specification of the snapshot-consistent disk model, which we briefly illustrate here using Figure 1. The state of the specification has two sector arrays, **stable** and **volatile**. The flush() operation copies **volatile** to **stable**; then the operation write$(0, x_9)$ changes **volatile**$[0]$ to x_9 and nothing else, while the read(2) operation returns **volatile**$[2]$; finally, the recovery() operation overwrites **volatile** with **stable**. In contrast to the specification, the SCFTL implementation stores the two arrays using more sophisticated data structures to achieve better performance and to satisfy the constraints of the flash memory. For instance, the **stable** array is implemented as an in-flash L2P and a list of L2P changes. Using a proof assistant, we reduce a *behavioral correctness* property over multiple FTL operations—which asserts that the SCFTL implementation behaves as the specification describes—to simpler *per-operation correctness* properties about each FTL operation (§4). We also prove that the behavioral correctness of SCFTL implies its snapshot consistency.

We then use more *automatic* tools to prove per-operation correctness (§5). More specifically, the relationship between

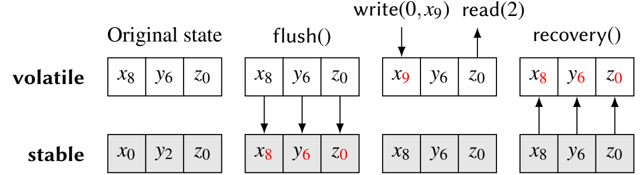

Figure 1. Illustration of the SCFTL operations.

the two arrays in the specification and the data structures in our implementation is described as a logical formula called the *abstraction relation*, and one type of the per-operation correctness formulae has the form "if the abstraction relation holds for the states before executing an operation, the relation will remain true for the states after executing the operation." We use a *symbolic executor* [8, 32] to translate the C program of our SCFTL implementation to logical formulae describing how the states change after executing an SCFTL operation. Then we use an SMT solver to ensure that our implementation does satisfy the per-operation correctness formulae.

Our experimental results (§7) show that when a workload does not flush the disk too frequently, SCFTL is as efficient as an FTL implementing the asynchronous disk model. To understand the usefulness of SCFTL, we modify the xv6 [16] file system to support group commit and utilize the stronger crash guarantee granted by SCFTL. By changing less than 30 lines of code, we show that the modified xv6 on SCFTL outperforms xv6 with logging on conventional FTLs by 3 to 30 times using our file system benchmarks; the performance improvement is less obvious for workloads that frequently flush the disk (e.g., smallfiles repeatedly creates a file, writes 100 bytes of data to it, and calls `fsync`), and more obvious for workloads with lower flush frequency (e.g., largefile writes 4 MB of data to a file and calls `fsync` for every 1 MB). This observation suggests an important guideline for building systems and applications on top of SCFTL: reducing the flush frequency to extract more benefits from SCFTL. Finally, we use the same file system benchmarks to compare the performance of the modified xv6 on SCFTL with the state-of-the-art setting: the ext4 file system on the pblk [6] FTL. The result is encouraging. Although xv6 is a file system known to be simple but slow, our xv6 on SCFTL is in the worst case only two times slower than ext4 on pblk. Moreover, xv6 on SCFTL has a stronger crash guarantee than that of ext4 on pblk.

In summary, the main contributions of this paper are the design, specification, and verification of SCFTL:

- The design exploits the out-of-place update feature of FTLs and uses an efficient checkpointing algorithm to provide a stronger crash guarantee at the disk level (§3). We validate its efficiency with disk and file system benchmarks (§7).
- The specification is simple and useful as it involves only the manipulation of two arrays and a counter, and it naturally ensures the atomicity of multiple disk writes. We formalize snapshot consistency and show that the specification satisfies the property with a proof assistant (§4).

- We verify that our implementation of SCFTL meets its specification using automatic verification tools. To scale the verification of SCFTL, we propose a novel approach to model crash behavior and describe some techniques to simplify the proof obligations, including using ghost variables to craft efficient SMT encodings, categorizing invariants to remove unnecessary conditions, and partitioning the proofs to avoid non-determinism (§5).

SCFTL has several limitations. First, SCFTL does not allow concurrent SCFTL operations, although it does allow flash operations to be executed concurrently (more detail in §6.2). Second, SCFTL assumes the underlying flash memory is free of error. Finally, SCFTL does not implement standard optimizations of FTLs, such as hot-cold data separation and wear leveling, although its design does not prohibit them.

2 Related Work

We first discuss the recent advance of disk models; in particular, we will focus on *transactional* and *order-preserving* models. To the best of our knowledge, our work is the first to address the verification of the crash safety issue at the physical device layer. Most previous work on crash safety verification assumes a correct asynchronous disk is given and put their focus on other layers in the storage stack, e.g., the file system. We will discuss some recent work in this direction.

The *transactional models* [15, 17, 25, 38, 41] guarantee the atomicity of multiple write operations. They provide a non-standard disk interface, which has two consequences: (i) their semantics (e.g., isolated concurrent transactions and transaction abortion) is hard to formally specify or verify, and (ii) system developers have to learn a new interface, increasing the burden of porting existing or developing new software.

The *order-preserving disk models* [11, 45] guarantee the preservation of operation orders across a crash. They expose the standard read-write-flush disk interface, but with fewer possible post-crash states than the asynchronous disk model. Upper layers in the storage stack can utilize this feature to reduce the number of flushes invoked in their crash recovery mechanisms (e.g., copy-on-write and journaling).

Compared with the transactional models, the snapshot-consistent model uses the standard read-write-flush interface. It, moreover, guarantees the atomicity of multiple writes between two consecutive flushes and thus provides a stronger guarantee than the order-preserving models.

Recent research work has discovered many crash vulnerabilities in widely used applications such as LevelDB and Git [36], as well as ACID violations in many relational database systems [46].These vulnerabilities mainly stem from the vague and weak crash guarantees provided by the underlying file systems. File systems themselves, even for mature ones such as ext4, btrfs [39] and F2FS [28], also contain bugs that may result in severe consequences [24, 27, 31].

In the past decades, a significant amount of research effort is devoted to the development of a verified crash-safe storage stack. To name a few, the verified file systems Yxv6 [42], FSCQ [13], and DFSCQ [12] assume an asynchronous disk model and use a log-based design to guarantee crash safety. Instead of the asynchronous disk model, both the BilbyFS [3] and Flashix [19] file systems assume the underlying layer is a raw flash device (without an FTL) and implement an atomic transaction mechanism to ensure crash safety.

SCFTL differs from previous work on verifying the storage stack in that it targets the physical device layer. This approach has three notable benefits: First, the code and the data structure of an FTL are usually simpler and more manageable than that of a file system; thus it is easier to develop an efficient yet verifiable layer. Second, providing the standard disk interface is more modular than directly building file systems on a raw flash device; developers can build their own systems with various features and optimizations, and leave crash safety to the underlying verified disk. Finally, this approach allows us to exploit useful device characteristics (e.g., the out-of-place update feature of FTLs); it enables SCFTL to provide a stronger crash guarantee without compromising the performance.

Regarding the verification methodology and framework, the closest work to ours is Yggdrasil [42], which establishes a forward simulation and discharges the proof obligations using an SMT solver, achieving a high degree of automation. Compared with Yggdrasil, we have additionally formalized our simulation argument, snapshot consistency, and relevant theorems using a proof assistant, providing more correctness guarantees. The proof obligations have to be manually massaged into a form that can be handled by an SMT solver; strictly speaking, this leaves a gap in our formal proof (but we bridge the gap by pen-and-paper reasoning).

By contrast, there has been work on storage system verification [9, 10, 13, 19] where the entire proof is formally verified with a proof assistant and does not have any gaps, although their verification cost is also significantly higher since the whole proof structure has to be carefully designed and constructed by programmers. Our framework is not as sophisticated as Argosy [9], which treats layered storage systems, or Perennial [10], which supports concurrency; extending our framework to support these features are interesting future directions. There are also differences in modeling decisions between our framework and the others—for example, in Argosy a crash is modeled as an individual event whereas we incorporate crashes into operations, and in Yggdrasil the effect of a lower-level operation may not be visible at a higher level whereas we always relate such an operation to a corresponding higher-level operation that is specified not to change the state. These differences in modeling decisions do not lead to vital differences in correctness guarantees, however.

3 SCFTL Design and Implementation

SCFTL is designed with *high performance*, *strong crash guarantees*, and *provable correctness* in mind. Below we give an

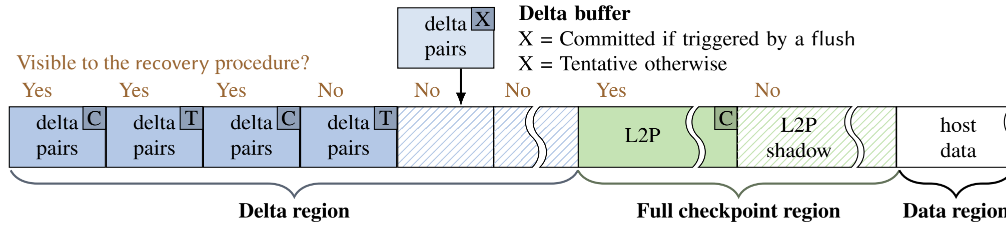

Figure 2. Flash memory layout of SCFTL.

overview of flash disks and describe the techniques we use that set SCFTL apart from traditional FTLs.

3.1 Flash disk overview

Usually, a flash disk contains two main components, a fast *dynamic random-access memory* (DRAM) to store temporary data and a slower *flash memory* to store permanent data. In this paper, we use the keywords *in-memory* or *buffer* to mean the data is stored in the DRAM and *in-flash* to mean it is stored in the flash memory. Flash memory is usually structured as a list of *blocks*, each of which consists of multiple *pages*. A page further contains one or multiple *sectors*, the basic unit the upper layers (e.g., the file system) use to access data. One can access flash memory through commands such as READ and WRITE a page, ERASE a block, and SYNC to wait for the completion of all ongoing flash commands. Due to physical limitations, flash commands need to follow several intricate constraints. For instance, a *page* must be erased before being written. However, the basic unit for ERASE is a block, while that for WRITE is a page. It would be inefficient if we erase the entire block whenever we write to a page within it.

In order to free users from handling the intricate device characteristics of flash memory, a flash disk usually comes with a flash translation layer (FTL) to hide the complexity. Typical operations supported by an FTL include a write and a read operation to store and retrieve a sector of data, a flush operation to wait until all unprocessed changes is made to the flash memory, a recovery procedure that will be invoked after a crash, and a garbage collection (GC) procedure gc that will only be invoked by its internal garbage collector. Every FTL should at least support the main functionalities, namely *address translation*, *crash recovery*, and *garbage collection*. Below we introduce how SCFTL implements those functions.

3.2 Address translation

To comply with the *erase-before-write constraint* of flash memory, SCFTL maintains an in-memory logical-to-physical table (L2P) and writes data in a log-structured manner [40] to avoid in-place updates. Address translation can be done at the granularity of sectors, pages, blocks, or a mixture of them [26]. Often finer granularity leads to better performance, but at the cost of higher memory usage due to a larger L2P. SCFTL uses a sector-level L2P to achieve better performance.

SCFTL handles the request write(la, d), i.e., writing a sector of data d to the logical sector address la, as follows. It first stores d into a page-sized *merge buffer* employed to resolve the size mismatch between a sector and a page. Then SCFTL finds a new location pa for placing d using the triplet (blk, pg, sec), called an *active pointer*, where the first two together point to the next free page to be written, and the last one points to the next free slot in the merge buffer. We call the block blk the *active block*. Then SCFTL also updates the in-memory L2P with the entry $la \mapsto pa$.

When the merge buffer is full, SCFTL invokes the command WRITE(blk, pg, d) to write the buffered data to the flash memory, where d is the content of the merge buffer. Then SCFTL increases pg by one to follow the *sequential write constraint* (within one block) of flash memory, unless pg is already the last page in a block, in which case blk is assigned a new block address dequeued from the *free block queue* and pg is reset to 0. The free block queue is an in-memory data structure that SCFTL uses to track currently available blocks.

Finally, SCFTL would have to remember that the address storing old data (if any) is no longer valid, and the one for the new data is now valid. The garbage collector needs this information to relocate all valid data before erasing a block. In SCFTL, this is realized by an in-memory physical-to-logical table (P2L), which is the "reverse" mapping of L2P. Flash disks usually have more available physical locations than logical locations. So it can happen that a mapping $p \mapsto l$ is in P2L, but $l \mapsto p$ is not in L2P. In such a case, we know that this physical address p is invalid. We also use an in-memory table, called the *valid count table*, to remember the number of valid sectors in each block. The valid count table will be used by the garbage collector to select the *victim block* to recycle.

Handling a read(la) request is simpler: SCFTL first checks if the requested data is still stored in the merge buffer; if so, it directly returns the data in the merge buffer. Otherwise, SCFTL performs an L2P lookup to find the corresponding physical address p, followed by a READ(p) command to retrieve the requested data stored in the flash memory. For a flush() request, SCFTL first stores the buffered data in the flash memory and advances the active pointer in the same way as handling a write request. Then it issues a SYNC command to ensure all data written before this point is persistent.

SCFTL maintains another L2P (the *stable L2P*) to reflect the state of the in-memory L2P right before the last flush for ensuring snapshot consistency. To distinguish the two kinds of L2P, we will call the in-memory L2P the *volatile L2P*. The cost of physically storing the stable L2P in the flash memory can be prohibitively high. Thus, SCFTL logically maintains the stable L2P through *checkpointing* in an efficient way.

Figure 2 shows the flash memory layout of SCFTL, including a *delta region* that records L2P differences, and a *full*

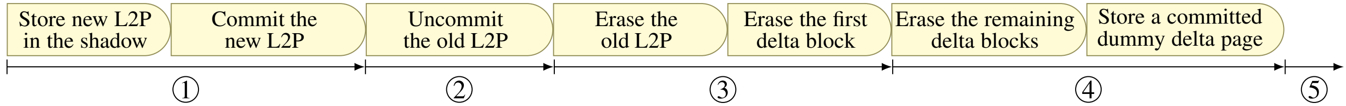

Figure 3. Full checkpoint protocol. The occurrence of crashes is partitioned to stages ① - ⑤.

checkpoint region that stores an entire L2P. SCFTL can create two kinds of checkpoints: a lightweight *delta checkpoint* and a heavyweight *full checkpoint*. SCFTL always creates a delta checkpoint when a flush is invoked. Under rare circumstances where the delta region nearly runs out of space, SCFTL creates a full checkpoint and clears the delta region.

Delta checkpoint Whenever a volatile L2P entry is modified due to a write or a gc operation, SCFTL also inserts a *delta pair*, which consists of a logical and a physical address, into the page-sized *delta buffer*. When the delta buffer is full, SCFTL invokes a WRITE command to store all buffered pairs along with a *tentative tag* into the in-flash delta region in a sequential manner and clears the buffer. We illustrate this operation in Figure 2.

On receiving a flush request, SCFTL first makes sure host data is safely stored in the flash memory. It then follows the same procedure above to store the buffered pairs in the flash memory, except here the delta page is tagged as *committed*. A committed delta page *activates* previous tentative delta pages, i.e., all delta pages before a committed one are treated as committed. When the committed delta page is safely kept in the flash memory, a delta checkpoint is successfully created.

Full checkpoint A full checkpoint of SCFTL consists of a complete L2P table and a commit flag. When making a full checkpoint, i.e., storing the volatile L2P together with a commit flag to the flash memory, we employ a *shadow* to prevent modification to the old L2P before the new one is settled. The detailed steps can be found in Figure 3. In short, we first store the new (volatile) L2P to the flash memory and start to erase the old L2P and the delta region only after the new L2P is committed. To ensure the correctness of our recovery procedure, we allow only flush operations to create a full checkpoint; write and gc operations are not allowed to create a full checkpoint. One potential issue is that the delta region might become full after a write or a gc operation. We address this issue by imposing upper bounds on the number of write and gc invocations (hence the number of created delta pairs) within an *epoch*, i.e., between two consecutive flushes. To ensure these bounds are respected, SCFTL uses a *write counter* and a *GC counter* to keep track of the number of write and gc invocations in the current epoch, respectively, and resets both counters on a flush or a recovery. If the upper layer calls a write after the write counter exceeds the bound, SCFTL simply treats that write as a no-op.

Systems that use SCFTL to implement atomic transactions should be aware of the write bound, to make sure an operation can fit into the current epoch before executing it. If an operation is too large to fit into an entire epoch (e.g., writing a large amount of data to a file), then it should be broken into multiple smaller ones. We believe that this requirement is not too restrictive, given that some systems also face a similar situation; for example, because of the log size limit, ext4 always checks that the current running transaction has sufficient capacity left before atomically updating its metadata [35, §3.1]. Similarly, systems built on SCFTL can record the number of writes issued to SCFTL since the last flush, and *before* executing an operation, calculate the number of writes required to complete the operation. If the operation does not fit into the current epoch, then the system should call a flush to form a new epoch. This avoids calling a flush in the middle of an operation, which may expose intermediate states on a crash.

3.3 Crash recovery

The recovery procedure first recovers the volatile L2P with full and delta checkpoints. To explain how to reconstruct the L2P, we begin by specifying what is *visible* to the recovery procedure. A full checkpoint is visible if and only if it is committed. For the delta region, we treat its first page specially to ensure we can erase the entire delta region atomically. The only important information on the first page is the commit flag. All delta pairs on the first page are *dummy delta pairs* that will be ignored in the recovery procedure. Visibility of delta pairs can be determined by examining the commit flag of the first page of the delta region and, if the commit flag is on, performing a sequential scan over the entire delta region to find the last committed page.

In most cases, the recovery procedure simply restores a base L2P from a full checkpoint and applies all visible delta pairs in sequential order. The only exception is when a crash occurs during a full checkpoint. Below we analyze the behavior of the recovery procedure against each crash point during a full checkpoint, as shown in Figure 3:

①: The old L2P and all delta pairs are visible. Restoring the old L2P and applying each delta pair yields the new L2P.

②: Both the old and new L2P, and all delta pairs are visible. Although we do not leave other information (e.g., a timestamp) to distinguish the old L2P from the new one, our checkpoint design ensures that starting from either L2P and applying each delta pair restores the new L2P.

③: The new L2P and all delta pairs are visible. For the same reasoning in ②, it restores the new L2P and applies each delta pair, yielding the new L2P.

④: The new L2P is visible and all delta pairs are invisible. It simply restores the new L2P.

⑤: The new L2P and the dummy pairs are visible. It restores the new L2P and ignores all dummy delta pairs.

Selective persistence The idea of selective persistence [34] is to persistently keep only *primary data*, which is the minimal set of data structures required for correct crash recovery,

and rebuild non-primary data from the primary one. This way, the protocol to correctly maintain consistency between multiple data structures can be greatly simplified. In SCFTL, the primary data is simply the L2P. After restoring the volatile L2P, the recovery procedure proceeds to rebuild the *block queues* (explained later), P2L table, and valid counts table.

3.4 Garbage collection

The GC procedure of standard FTLs consists of the following steps: (i) find a *victim block*, (ii) relocate all valid sectors (those pointed to by the volatile L2P) in the victim block similarly to the write operation (the *relocation phase*), and (iii) erase the victim block (the *erasure phase*). In SCFTL, the standard GC procedure cannot be used, as the data pointed to by the stable L2P might be erased by a garbage collector.

To prevent such an issue, one design option is to keep additional information on what is allowed to be erased. This approach is taken by TxFlash [38, §3.3] to prevent back pointers from being erased. The downside of this approach is that it can incur notable memory and performance overhead.

Another approach, adopted by OPTR [11, §3.4], is to invoke an internal flush (i.e., a flush operation issued by the FTL) before GC is activated. While an internal flush is allowed by the prefix-preserving guarantee that OPTR offers, it is not allowed by the snapshot consistency SCFTL is trying to achieve, as otherwise SCFTL might rollback to the state right before an internal flush on a recovery.

Two-phase garbage collection Instead, we use a simple protocol called *two-phase garbage collection* (2PGC), which delays the erasure phase until a flush is invoked. This is correct because after a flush operation, the old stable L2P will be discarded and hence all the previously selected victim blocks can be safely erased. To implement this idea, in the 2PGC mechanism, we use two functions $\mathsf{gc}_{\mathsf{rl}}$ and $\mathsf{gc}_{\mathsf{es}}$ to handle the relocation and erasure phases, respectively. We, moreover, maintain four queues to remember the status of blocks. Initially, all blocks are in the *free block queue*, except the active block, which does not belong to any of the state queues. When the active block is fully written, we add it to the *used block queue* and pick another block from the *free block queue* as the new active block.

When the garbage collector invokes the $\mathsf{gc}_{\mathsf{rl}}$ function, it first removes a victim block from the used queue and performs the relocation of valid sectors. After all valid sectors are removed from the victim block, the $\mathsf{gc}_{\mathsf{rl}}$ function adds the victim block to the *invalid block queue*. As the name suggests, all blocks in the invalid queue do not have any valid sector. Nevertheless, those blocks might still contain sectors pointed to by the stable L2P and hence cannot be immediately erased. All blocks in the *invalid* queue will be moved to the *erasable block queue* at the end of a flush operation. After a flush, the stable L2P will be updated, and all invalid blocks are no longer pointed to by the new stable L2P and are now erasable. All blocks in the erasable block queue can be safely erased. When the garbage collector invokes the $\mathsf{gc}_{\mathsf{es}}$ function, it finds a block b in the erasable queue, performs an ERASE(b) command, and puts b in the free block queue.

The remaining problem of 2PGC is that garbage collected blocks cannot be immediately reused in the current epoch. To address this issue, SCFTL exploits the upper bounds on the number of write and $\mathsf{gc}_{\mathsf{rl}}$ allowed in one epoch to ensure that there is sufficient space for newly written data and relocated data. These two bounds already exist to avoid overflowing the delta region (§3.2). Below we describe the constraints that need to be satisfied when picking the values for the two bounds, W (for write, in terms of sectors) and K (for $\mathsf{gc}_{\mathsf{rl}}$, in terms of blocks).

The first constraint ensures that write and $\mathsf{gc}_{\mathsf{rl}}$ do not consume space more than what $\mathsf{gc}_{\mathsf{rl}}$ can produce in one epoch:

$$\underbrace{W}_{\text{consumed by write}} + \underbrace{KN}_{\text{consumed by } \mathsf{gc}_{\mathsf{rl}}} \leq \underbrace{KS}_{\text{produced by } \mathsf{gc}_{\mathsf{rl}}} \quad (1)$$

where N is the maximum number of valid sectors in every victim block (we will explain how to obtain this bound later); S is the number of sectors per block. Each write occupies one sector and each $\mathsf{gc}_{\mathsf{rl}}$ relocates at most N sectors; thus at most $W+KN$ sectors will be consumed in one epoch. Each $\mathsf{gc}_{\mathsf{rl}}$ also turns one used block into one invalid block, which becomes an erasable block in the next epoch; thus at most KS sectors can be produced in one epoch.

To obtain the bound N, we introduce a GC threshold U: $\mathsf{gc}_{\mathsf{rl}}$ can be activated only when the number of used blocks is larger than or equal to U. Let L be the number of entries of L2P (i.e., the number of logical sectors). Recall that a valid sector is a physical sector mapped by the volatile L2P. The *pigeonhole principle* states that there exists a used block (i.e., the hole) whose number of valid sectors (i.e., the pigeons) is no more than $\lfloor L/U \rfloor$. One design choice is to force $\mathsf{gc}_{\mathsf{rl}}$ to always pick the block with the least number of valid sectors as the victim block (i.e., the greedy policy). A more flexible alternative allows $\mathsf{gc}_{\mathsf{rl}}$ to pick any block providing the block has no more than $\lfloor L/U \rfloor$ valid sectors; the pigeonhole principle ensures the existence of such block. Either way, we obtain $N = \lfloor L/U \rfloor$.

The threshold may disable $\mathsf{gc}_{\mathsf{rl}}$ before there is enough space for the next epoch; SCFTL would fail to proceed to the next epoch in this situation. To avoid such a situation, we can choose a proper U, W, and K such that $\mathsf{gc}_{\mathsf{rl}}$ must be enabled when there is not enough space for the next epoch:

$$\underbrace{U}_{\text{GC threshold}} \leq \underbrace{P - 1 - \lceil (W+KN)/S \rceil}_{\text{lower bound of used blocks when not enough space}} \quad (2)$$

where P is the number of data blocks. First observe that free, erasable, and invalid blocks are all available to the next epoch as all invalid blocks become erasable after a flush. The condition of "not enough space" essentially means that the total number of these three kinds of blocks is less than $\lceil (W+KN)/S \rceil$. Given that the total number of free, erasable, invalid, and used blocks is equal to $P-1$ (we always have

one active block), we know that the number of used blocks is more than $P-1-\lceil (W+KN)/S \rceil$ when there is not enough space for the next epoch.

We demonstrate a configuration that satisfies the above constraints: A 4-GB flash disk has 2^{20} logical sectors ($L = 2^{20}$); suppose each block has 512 sectors ($S = 512$) and we have 6 GB of flash memory for storing data, then the number of data blocks will be 3072 ($P = 3072$). We can set the GC threshold at 2500 ($U = 2500$) and we know the number of valid sectors in every victim block would not exceed 419 ($\lfloor L/U \rfloor = \lfloor 2^{20}/2500 \rfloor = 419$). Next we pick suitable values for W and K. Let $W = 4000$ and $K = 50$ and check whether they satisfy the above two constraints:

$$24950 = W + KN \leq KS = 25600$$
$$2500 = U \leq P-1-\lceil (W+KN)/S \rceil = 3022$$

Both constraints are satisfied.

Recovery of block queues After the recovery procedure reconstructs the volatile L2P, it proceeds to reconstruct other data structures, including the aforementioned block queues. The recovery procedure simply scans through the blocks. If a block contains some sectors pointed to by the L2P, then it is a used block; otherwise, the recovery procedure treats it as an erasable block. Note that after the reconstruction of the volatile L2P, the stable and volatile L2Ps are identical and hence we do not have any invalid blocks. We do not have any free blocks after the recovery procedure; we do not have the information whether a block was in the erasable or free block queue before the crash. So we have to play it safe and put them in the erasable queue to follow the erase-before-write flash constraint.

4 Formal Verification Framework

The design of SCFTL (§3) is fairly sophisticated; to provide a strong guarantee of its reliability, we have formally verified its correctness. Our formal verification framework starts with the definition of a disk model $\mathbb{S}$ (similar to Figure 1) that specifies the intended disk behavior. $\mathbb{S}$ is defined as a particular kind of state transition system (§4.1) on which snapshot consistency can be formulated and proven (§4.2). As opposed to $\mathbb{S}$, which is merely an abstract, mathematical specification that is meant to be understood easily, the SCFTL implementation constitutes a more realistic transition system $\mathbb{P}$. We prove that $\mathbb{P}$ is behaviorally correct with respect to $\mathbb{S}$, and moreover, this behavioral correctness is strong enough to imply the snapshot consistency of $\mathbb{P}$ (§4.3). Behavioral correctness is a complicated property about sequences of state transitions, which cannot be easily verified with automatic verification tools. We can, however, reduce its proof to one about the behavior of individual operations (§4.4), the latter of which is more amenable to automatic verification (§5). The content of this section is formally verified with the Agda proof assistant [33], but here we provide only a high-level sketch.

4.1 Specification of disk behavior

To model the behavior of a disk as a state transition system, we should define the possible states of a disk and the operations that can be performed on the disk states. In $\mathbb{S}$, which is our abstract disk model that acts as a definition of intended behavior, a state t of a disk is a pair of arrays $t\,.volatile$ and $t\,.stable$ representing the volatile and stable copies of disk data respectively and a number $t\,.wcnt$ that counts the number of writes since the last flush. There is a set $\mathcal{N}_{\mathbb{S}}$ of states that represent the possible contents of a new disk, where only the *stable* array is initialized to some default value. Reading a disk state is just retrieving the data at a given address in the volatile part; since the operation does not change the disk state, we simply define it as a function $read(t,a) \triangleq t\,.volatile[a]$ rather than a kind of state transition.

Mirroring the FTL operations except read, the operations of $\mathbb{S}$ are shown in Figure 4: they are classified as *regular*, *flush*, and *recovery* operations, and have a successfully executed version and a crashed version. Writing $t \xrightarrow{op} t'$ to mean that there is a transition from t to t' through the operation *op*, that is, t' is the state resulting from applying the operation *op* to the state t, we define the effect of an operation by specifying how t and t' are related: A write operation $\mathsf{w}_{a,d}$, which writes the piece of data d to the address a in the volatile part, is defined by saying that $t \xrightarrow{\mathsf{w}_{a,d}} t'$ amounts to

- $t'\,.volatile = t\,.volatile[a \mapsto d]$, where the right-hand side is the array whose values are the same as $t\,.volatile$ except at the address a, where the value is d,
- $t'\,.stable = t\,.stable$, meaning that the stable data is not modified, and
- $t'\,.wcnt = t\,.wcnt + 1$, meaning that *wcnt*, the write counter mentioned at the end of §3.2, is incremented by one

when a and $t\,.wcnt$ are within bounds, or otherwise $t' = t$. The garbage-collecting operations rl and es do not change the (abstract) disk state. The flush operation f copies the volatile data to the stable part, and the recovery operation r does the opposite; both operations reset the write counter to 0. The crashed operations $\mathsf{w}^{\mathsf{c}}_{a,d}$, rl^{c}, es^{c}, and r^{c} may disrupt the volatile data arbitrarily, and thus their definitions only specify that the stable data remains the same (and the transitions become non-deterministic); for f^{c} there are two kinds of post-crash state because the update to the flash disk may have finished, in which case the system behaves as if the flush operation is successfully executed.

4.2 Snapshot consistency

Snapshot consistency is essentially a property about *execution fragments* (or *fragments* for short), which are consecutive sequences of transitions $t_1 \xrightarrow{op_1} t_2 \xrightarrow{op_2} \cdots \xrightarrow{op_n} t_{n+1}$; we often omit unimportant intermediate states and write $t_1 \xrightarrow{op_1, op_2, \ldots, op_n} t_{n+1}$. Informally, snapshot consistency says that when recovered from a crash, reading the state after the recovery operation will be the same as reading the state right

	Regular		Flush	Recovery
	Write	**Relocate/Erase (GC)**	**Flush**	**Recover**
Successful	$t \xrightarrow{\mathsf{w}_{a,d}} t' \triangleq$ $(\mathit{InBounds}(a, t.wcnt) \wedge t'.volatile = t.volatile[a \mapsto d]$ $\wedge\, t'.stable = t.stable \wedge t'.wcnt = t.wcnt + 1)$ $\vee\, (\neg \mathit{InBounds}(a, t.wcnt) \wedge t' = t)$	$t \xrightarrow{\mathsf{rl}/\mathsf{es}} t' \triangleq$ $t' = t$	$t \xrightarrow{\mathsf{f}} t' \triangleq$ $t'.volatile = t.volatile$ $\wedge\, t'.stable = t.volatile$ $\wedge\, t'.wcnt = 0$	$t \xrightarrow{\mathsf{r}} t' \triangleq$ $t'.volatile = t.stable$ $\wedge\, t'.stable = t.stable$ $\wedge\, t'.wcnt = 0$
Crashed	$t \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a,d}} t' \triangleq$ $t'.stable = t.stable$	$t \xrightarrow{\mathsf{rl}^{\mathsf{c}}/\mathsf{es}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.stable$	$t \xrightarrow{\mathsf{f}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.volatile$ $\vee\, t'.stable = t.stable$	$t \xrightarrow{\mathsf{r}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.stable$

Figure 4. Definitions of operations in $\mathbb{S}$. GC stands for Garbage Collection. The definition of the predicate $\mathit{InBounds}(a, wcnt)$ is $a \leq a_{max} \wedge wcnt \leq wcnt_{max}$ where a_{max} and $wcnt_{max}$ are the upper bounds on the addresses and the write counter respectively. Note that in the definitions a write operation has no effect ($t' = t$) when the write counter exceeds the bound ($wcnt > wcnt_{max}$).

before the last flush operation prior to the crash. More precisely, the disk may have operated normally for some time before the crash happens, and the recovery may fail several times before it succeeds. This whole behavior is described as a *one-recovery fragment* of the form

$$t_1 \xrightarrow{a_1,\ldots,a_{k-1}} t_2 \xrightarrow{a_k(=\mathsf{f}),b_1,\ldots,b_\ell} t_3 \xrightarrow{c,(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} t_4$$

where $a_1, \ldots, a_k$ are a sequence of successful regular or flush operations ending with f (that is, $a_k = \mathsf{f}$), $b_1, \ldots, b_\ell$ are successful regular operations, c is a crashed regular or flush operation, and $(\mathsf{r}^{\mathsf{c}})^m$ is the crashed recovery operation repeated m times; this is abbreviated to $t_1 \rightsquigarrow t_4$ later on. Writing $t \approx t'$ to mean that $read(t,a) = read(t',a)$ for all addresses a within bounds, snapshot consistency of a one-recovery fragment of the above form is defined as follows:

- if c is a crashed regular operation, then $t_2 \approx t_4$;
- if c is the crashed flush operation f^{c}, then either $t_2 \approx t_4$ or $t_3 \approx t_4$.

Note that in the definition k can be 0, in which case no flush is performed before the crash c, and the definition requires, for instance in the first case, that the disk be reverted to the first state t_1 (which equals t_2) observationally.

We can now formulate snapshot consistency of a disk model. The typical way of using a disk can be represented as a *multi-recovery fragment* of the form

$$t_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^\ell,\mathsf{r}} t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots \rightsquigarrow t_n \rightsquigarrow t_{n+1} \xrightarrow{a_1,\ldots,a_m} t_{n+2}$$

which starts with performing the recovery operation on a state $t_0 \in \mathcal{N}_{\mathbb{S}}$ (until it succeeds) to bring the disk to a usable state, and continues with an arbitrary number of one-recovery fragments and some trailing regular and flush transitions representing uses of the disk. We say that a multi-recovery fragment of this form is snapshot-consistent if all the one-recovery sub-fragments $t_1 \rightsquigarrow t_2, \ldots, t_n \rightsquigarrow t_{n+1}$ are snapshot-consistent, and that a disk model is snapshot-consistent if its every multi-recovery fragment is snapshot-consistent. With the definitions in place, we can now state our first result (whose proof is straightforward and omitted here).

Lemma 1. *$\mathbb{S}$ is snapshot-consistent.*

Note that the definitions of snapshot consistency make sense for any disk model that has the same structure as $\mathbb{S}$, in particular for the model $\mathbb{P}$ that we will describe next.

4.3 Behavioral correctness and snapshot consistency of SCFTL

$\mathbb{S}$ is a simplistic transition system: it gives a concise definition of the intended disk behavior, but is unsuitable for direct implementation. In SCFTL (§3), we use more practical states that consist of various in-memory and in-flash data structures, and sophisticated operations implemented in C. All these give rise to another transition system $\mathbb{P}$, which has the same set of operations as $\mathbb{S}$ (as well as a *read* function for reading the states of $\mathbb{P}$) and also a set $\mathcal{N}_{\mathbb{P}}$ of possible states of a new disk (where only the in-flash part is initialized). The definition of $\mathbb{P}$ is a formal version of what has been presented in §3; the exact definition is not needed in this section though, and will be described later in §5.

We have proven that $\mathbb{P}$ is *behaviorally correct* with respect to $\mathbb{S}$: if we perform a legitimate sequence of operations in $\mathbb{P}$ to obtain a fragment and read the *normal* states, which are states that immediately follow a successful operation, the results will be the same as performing the same sequence of operations in $\mathbb{S}$ and reading the corresponding states. This property allows the behavior of $\mathbb{P}$ to be understood in terms of $\mathbb{S}$. More formally, we have the following theorem.

Theorem 1 (behavioral correctness of $\mathbb{P}$)**.** *For every multi-recovery fragment $s_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} s_1 \xrightarrow{op_1} \ldots \xrightarrow{op_n} s_{n+1}$ in $\mathbb{P}$, there exists a fragment $t_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} t_1 \xrightarrow{op_1} \ldots \xrightarrow{op_n} t_{n+1}$ in $\mathbb{S}$ (which has the same sequence of operations) such that $s_i \approx t_i$ for every corresponding pair of normal states s_i and t_i.*

We will see how Theorem 1 is proven in §4.4 and §5. Before we do so, we show that the behavioral correctness of $\mathbb{P}$ is strong enough to allow $\mathbb{P}$ to inherit snapshot consistency from $\mathbb{S}$.

Theorem 2. *$\mathbb{P}$ is snapshot-consistent.*

Proof. We must show that any multi-recovery fragment in $\mathbb{P}$ is snapshot-consistent, that is, the results of reading the states mentioned by the definitions of snapshot consistency are the same. Observe that all these states are normal, so reading these states in the fragment (in $\mathbb{P}$) is the same as reading the

corresponding states in the fragment in $\mathbb{S}$ that is guaranteed to exist by the behavioral correctness of $\mathbb{P}$. This means that the $\mathbb{P}$-fragment is snapshot-consistent if the $\mathbb{S}$-fragment is, and the latter is indeed the case because $\mathbb{S}$ is snapshot-consistent (Lemma 1). □

4.4 Per-operation correctness

Theorem 1 is proven with a *forward simulation* argument [30], which, though fairly standard, is described below for the sake of completeness. Given a multi-recovery fragment in $\mathbb{P}$, we construct a fragment in $\mathbb{S}$ with the same sequence of operations while ensuring that every corresponding pair of normal states in the two fragments is related by an *abstraction relation AR* (to be described below) such that $AR(s,t)$ implies $s \approx t$ for all s and t, and moreover, any corresponding pair of abnormal states is related by a weaker abstraction relation CR. This forms a stepwise relationship between the two fragments, as illustrated, for example, in

$$\begin{array}{cccccccccccccccc} & s_0 & \xrightarrow{\mathsf{r}} & s_1 & \xrightarrow{\mathsf{w}_{a,d}} & s_2 & \xrightarrow{\mathsf{f}} & s_3 & \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a',d'}} & s_4 & \xrightarrow{\mathsf{r}^{\mathsf{c}}} & s_5 & \xrightarrow{\mathsf{r}} & s_6 & \longrightarrow & \\ CR & \vdots & AR & \vdots & AR & \vdots & AR & \vdots & CR & \vdots & CR & \vdots & AR & \vdots & & \cdots \\ & t_0 & \xrightarrow{\mathsf{r}} & t_1 & \xrightarrow{\mathsf{w}_{a,d}} & t_2 & \xrightarrow{\mathsf{f}} & t_3 & \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a',d'}} & t_4 & \xrightarrow{\mathsf{r}^{\mathsf{c}}} & t_5 & \xrightarrow{\mathsf{r}} & t_6 & \longrightarrow & \end{array} \quad (3)$$

Intuitively, the abstraction relation AR captures how a normal state in $\mathbb{P}$ is interpreted as a state in $\mathbb{S}$. For example, one part of AR describes where the current data at a logical address—i.e., an entry in the *volatile* array of an $\mathbb{S}$-state—can be found in a $\mathbb{P}$-state. On the other hand, for abnormal states in $\mathbb{P}$, only the in-flash data are reliable, and the crash abstraction relation CR describes only how the in-flash part of a $\mathbb{P}$-state is interpreted as the stable part of an $\mathbb{S}$-state. Back to diagram (3): The in-memory part of s_0 ($\in \mathcal{N}_{\mathbb{P}}$) is not yet initialized, and thus s_0 only satisfies CR with some t_0 ($\in \mathcal{N}_{\mathbb{S}}$). A successful recovery operation brings the disk to a normal $\mathbb{P}$-state that satisfies AR with an $\mathbb{S}$-state. This AR relationship is preserved by successful regular and flush operations, but deteriorates to CR when a regular or flush operation crashes. Recovery attempts may fail but CR is preserved, and the relationship is restored to AR after the recovery succeeds, from which point we can resume using the disk.

The stepwise relationship is established inductively by showing (i) that initially CR holds for all $s \in \mathcal{N}_{\mathbb{P}}$ and $t \in \mathcal{N}_{\mathbb{S}}$ (to establish, for example, the leftmost $CR(s_0,t_0)$ in diagram (3)), and (ii) that each operation preserves AR or CR, or transforms AR into CR or vice versa (giving rise to each of the squares in diagram (3)). The inductive cases (ii) are called *type A per-operation correctness*. For example, the type A per-operation correctness formula for the crashed flush operation f^{c} is

$$\forall s,t,s'.\, AR(s,t) \wedge RI(s) \wedge s \xrightarrow{\mathsf{f}^{\mathsf{c}}} s' \implies \exists t'.\, t \xrightarrow{\mathsf{f}^{\mathsf{c}}} t' \wedge CR(s',t') \quad (4)$$

where RI is the *representation invariant* describing properties that should be satisfied by the various data structures in a $\mathbb{P}$-state, and is needed in the antecedent to make per-operation correctness provable. For example, a part of RI states that the in-memory L2P should agree with the in-flash L2P in addition to all the delta pairs. Like AR and CR, there is also a weaker version of RI called CI that describes only the properties about the in-flash part of a $\mathbb{P}$-state, and is used in the relevant per-operation correctness formulae. Note that per-operation correctness is about the behavior of an operation in general, not about its effect on particular states. We have thus reduced reasoning about fragments, of which there is a myriad possibilities, to reasoning about operations, of which there is only a handful.

Finally, to make the induction go through, we need to establish RI on all normal states and CI on all abnormal states so that the RI and CI premises in the type A per-operation correctness formulae are satisfied. This is done by showing that the invariants are suitably preserved or transformed by each operation, for example,

$$\forall s,s'.\, RI(s) \wedge s \xrightarrow{\mathsf{f}} s' \implies RI(s') \quad (5)$$

These formulae are called *type B per-operation correctness*.

Up to this point, what we have proven is that $\mathbb{P}$ is behaviorally correct if (i) $AR(s,t)$ implies $s \approx t$ for all s and t, (ii) $CR(s,t)$ for all $s \in \mathcal{N}_{\mathbb{P}}$ and $t \in \mathcal{N}_{\mathbb{S}}$ and $CI(s)$ for all $s \in \mathcal{N}_{\mathbb{P}}$, and (iii) (type A and type B) per-operation correctness holds for each operation. The three conditions are discharged using automatic verification techniques described next in §5.

5 Verifying the SCFTL Implementation

We use the SMT solver Z3 [18] to prove the correctness of the aforementioned three conditions. The first two conditions are easy to check: once we have the formulae describing the invariants RI and CI and the abstraction relations AR and CR, we can easily construct the corresponding formulae and let Z3 prove their validity automatically. For the third condition, i.e., the per-operation correctness, we need to construct the formulae $s \xrightarrow{op} s'$ in $\mathbb{P}$ from the C implementation for all operations op. We use the symbolic executor Serval [32] to build these formulae (§5.1). If we naively construct the per-operation correctness formulae, often the generated formulae would be too difficult for Z3 to solve. We explain in §5.2–§5.4 how to simplify the formulae so that Z3 can handle them.

5.1 Modeling flash states and crashes

To perform symbolic execution for SCFTL, we need to translate C statements into formulae describing how they update the $\mathbb{P}$-states. A $\mathbb{P}$-state includes a *memory state* and a *flash state*. The memory state is a mapping from variable names to their in-memory value. For example, it maps L2P to an in-memory table. Serval can handle the update of memory states and produce the corresponding formula automatically. We model a flash state as a function $content(bk,pg)$ that maps a *flash location*, which is a pair (bk,pg) where bk is a block address and pg is a page address, to the data stored in that

page, and modify Serval to support the flash commands SYNC, ERASE, READ, and WRITE. More concretely, we need to tell Serval how those commands update flash states.

The WRITE(bk, pg, d) command updates the flash state $content(b, p)$ to $ite(bk = b \wedge pg = p,\ d,\ content(b, p))$, where $ite(g,\ e_1,\ e_2)$ is a shorthand for "if g then e_1 else e_2." The ERASE(bk) command updates the flash state $content(b, p)$ to $ite(bk = b,\ empty,\ content(b, p))$, where *empty* is a page (which can be modeled as, e.g., an array) with all cells valued -1. Neither the READ(bk, pg) nor the SYNC commands change $content(b, p)$ in the flash state.

Handling asynchronous flash operations The flash commands are *asynchronous*, i.e., the invoked commands first wait in a queue and start to update the flash memory only when the scheduler selects them. Updates to the same page will be executed in the same order in which they come into the queue, but there is no restriction regarding when the updates happen for different pages. If the system crashes, it will lose all commands in the queue.

The flash command SYNC blocks the system until the queue becomes empty. If the system crashes right after a SYNC command, there will be only one possible flash state. However, when it happens between two SYNC commands, there can be multiple possibilities, because we do not know which of those queued commands are processed.

Example 1. Suppose the content at the location (b_1, p_1) is *empty* before invoking the sequence of flash commands WRITE(b_1, p_1, d_1), ERASE(b_1, p_1), WRITE(b_1, p_1, d_2), SYNC. If the system crashes right before SYNC, the content at (b_1, p_1) can be either *empty*, d_1, or d_2.

One way to model crash behavior is to use *crash schedules* [42, §3.1], which are a set of boolean variables representing the occurrence of crash events during the execution of an operation. A special case where all the boolean variables are true indicates a successful execution, in which all the WRITEs invoked by the operation are synchronized. If we adopted this approach, then SCFTL would have to issue a SYNC at the end of each operation, limiting concurrency within a single operation. However, in our SCFTL implementation, it often happens that a SYNC is invoked only after multiple operations. Thus, this modeling would reduce performance significantly.

An alternative approach represents each flash page as a *history of values* [13, §3.2], which are the values written asynchronously to the flash page since the last SYNC. The history can be implemented as a *list*, and a crash non-deterministically chooses a value from the list. This modeling does not require synchronization at the end of an operation, and therefore does not limit concurrency. However, lists are not well supported by SMT solvers.

We propose a novel approach to model crash behavior, which does not limit concurrency and is amenable to SMT reasoning. The main idea is to "over-approximate" possible flash states when they are affected by asynchronous updates. We implement the idea by adding to the flash state a mapping $sync(b, p)$ that maps a flash location (b, p) to a boolean value denoting whether the page is synchronized, i.e., it is not affected by asynchronous updates since the last SYNC. The WRITE(bk, pg, d) command updates $sync(b, p)$ to $ite(bk = b \wedge pg = p,\ false,\ sync(b, p))$ and the ERASE(bk) command updates it to $ite(bk = b,\ false,\ sync(b, p))$. The READ$(b, p)$ command does not change $sync(b, p)$ and always returns $content(b, p)$ no matter $sync(b, p)$ is true or not. The SYNC command remaps all locations of *sync* to *true*.

If *op* is a successfully executed operation, we collect all $\mathbb{P}$-states produced after executing *op* symbolically and use them to construct the formula for $s \xrightarrow{op} s'$.

If *op* is a crashed operation, we collect all possible crash states in two steps. We first collect all flash states (i) right before every SYNC command and (ii) after executing *op*. We then update the $content(b, p)$ function of the collected states to $ite(sync(b, p),\ content(b, p),\ any)$, where *any* means the content can be any value. We model *any* with *fresh variables*, whose values can be arbitrarily assigned. The formula $s \xrightarrow{op} s'$ can then be constructed from all the $content(b, p)$ functions of the collected states. We also designed a suitable crash representation invariant *CI* for SCFTL operations that records flash disk information that is just sufficient for the recovery operation. For instance, it states that "at least one of the two full checkpoints is committed." The *CI* can be guaranteed even with the over-approximated flash state we just introduced.

As said, the formulae produced with the approach we just described may be too difficult for SMT solvers to solve. Below (§5.2–§5.4) we introduce the techniques we use to simplify the formulae and make automatic verification feasible. These techniques are very general and should be usable by other automatic verification projects.

5.2 Crafting the abstraction relations and representation invariants

To avoid overwhelming the SMT solvers, care must be taken to put the abstraction relations and representation invariants in a suitable form. Below we look at a concrete example. A part of the abstraction relations asserts that the stable L2P in $\mathbb{S}$ should agree with its concrete representation in $\mathbb{P}$, which is the in-flash L2P stored in a committed full checkpoint, in addition to all the committed delta pairs stored in the delta region (Figure 2). As opposed to representing the assertion as a relation, we could define a *function* that computes the physical address for a given logical address *la* from a $\mathbb{P}$-state by starting with the in-flash L2P and then sequentially applying the delta pairs, and assert that the stable L2P in $\mathbb{S}$ agrees with the results of the function. Serval compiles the assertion to the following constraint (assuming for simplicity that there are only two delta pairs (la_0, pa_0) and (la_1, pa_1)):

$$\forall la.\ L2P_{stable}[la] = ite(la = la_1,\ pa_1,\ ite(la = la_0,\ pa_0,\ L2P_{flash}[la]))$$

The number of *ite*s is the same as the number of delta pairs, which in the implementation is set to be large enough to avoid frequent full checkpointing. The resultant formula turns out to be too large for the SMT solvers to handle, though.

We thus choose to represent the assertion as a relation, which is divided into two disjoint parts. The first part considers logical addresses that appear in the delta region, and the second part considers those that do not. In more detail:

- For every ($\forall$) logical address *la* that appears in the region, there exists ($\exists$) a pair (la, pa) in the region such that $L2P_{stable}[la] = pa$. Moreover, this pair should be the last one about *la* in the sense that for any ($\forall$) subsequent pair (la', pa') we have $la' \neq la$.
- For every ($\forall$) logical address *la* that does not appear in the region, we have $L2P_{stable}[la] = L2P_{flash}[la]$.

The first part of the relation is still not amenable to efficient SMT solving as it contains *quantifier alternation* of the form $\forall x. \exists y. \forall z. \ldots$, which is often too hard for the SMT solvers [4, 44] to deal with. The key observation is that the existential quantifier $\exists$ can be avoided with the *ghost variable* technique [20]: because *pa* is determined by *la* and the $\mathbb{P}$-state, we can extend the $\mathbb{S}$-states with an auxiliary array *aux* (along with some other ghost variables required to determine *pa*) and modify the transition relation of $\mathbb{S}$ to keep track of the last *pa* associated with each *la*; the formula can then simply use $aux[la]$ (instead of an existentially quantified variable) wherever it needs to refer to the last *pa* associated with *la*. Note that the transitions of the non-ghost variables (i.e., *volatile*, *stable*, and *wcnt*) must not depend on the ghost variables (e.g., *aux*), so that the specification of interest (Figure 4) is essentially a *projection* of the $\mathbb{S}$-states, in which ghost variables are removed from the state space.

5.3 Categorizing the invariants

We use the observation that, usually, the invariants and abstraction relations can be grouped into different categories and handled separately in verification. For example, the *RI* may include constraints for different components of SCFTL, e.g., checkpoint and garbage collection. To simplify the presentation, here we assume $RI(s) = RI_{chk}(s) \wedge RI_{gc}(s) \wedge RI_{other}(s)$, where $RI_{chk}(s)$ are invariants related to checkpoints, $RI_{gc}(s)$ are those related to garbage collection, and $RI_{other}(s)$ are other invariants. Now we can divide Formula 5 into three simpler formulae: $\forall s, s'.\, RI(s) \wedge s \xrightarrow{\mathsf{op}} s' \implies RI_x(s')$, where $x \in \{chk, gc, other\}$, and prove their correctness separately. We can further simplify the formulae by using only a subset of the constraints in *RI* to show the preservation of invariants. The reason is that to show $RI_x(s')$ holds after the execution of op, we usually do not need the starting state s to also satisfy the invariants related to other components. Hence we can use the formula $\forall s, s'.\, RI_x(s) \wedge s \xrightarrow{\mathsf{op}} s' \implies RI_x(s')$ instead.

5.4 Partitioning the proofs

Both the implementation and specification of SCFTL involve "non-determinism" when a flush operation crashes. In the implementation, we collect multiple flash states to produce the formula $s \xrightarrow{\mathsf{f^c}} s'$. In the specification (Figure 4), when a flash operation crashes, the stable data may remain unchanged ($t'.stable = t.stable$) or change to the volatile data ($t'.stable = t.volatile$). We found that such non-determinism induces verification bottlenecks. We tried to prove the correctness of flush using Z3 with the presence of such non-determinism, but the solver failed to solve it given a few days of time budget. In our experience, this usually means the problem is too difficult for Z3 and needs to be simplified. For SCFTL, such non-determinism can be avoided by *partitioning the proofs*. We need to figure out (i) which flash states of the implementation correspond to the specification $t'.stable = t.stable$ and (ii) which correspond to $t'.stable = t.volatile$. More concretely, assuming that we collect two flash states, represented as $f_1(s)$ and $f_2(s)$, during the execution of the crashed flush operation, we can substitute the transition relations in Formula 4 properly to obtain:

$$\forall s, t, s'.\, AR(s,t) \wedge RI(s) \wedge (s' = f_1(s) \vee s' = f_2(s)) \\ \implies \exists t'.\, \begin{pmatrix} t'.stable = t.stable \\ \vee\, t'.stable = t.volatile \end{pmatrix} \wedge CR(s', t') \quad (6)$$

Instead of directly proving Formula 6, which involves non-determinism ($\vee$), we can use Z3 to prove the following two sufficient conditions separately—if we know that the first flash state corresponds to $t'.stable = t.stable$, and the second corresponds to $t'.stable = t.volatile$:

$$\forall s, t, s'.\, AR(s,t) \wedge RI(s) \wedge s' = f_1(s) \\ \implies \exists t'.\, t'.stable = t.stable \wedge CR(s', t') \quad (7)$$

$$\forall s, t, s'.\, AR(s,t) \wedge RI(s) \wedge s' = f_2(s) \\ \implies \exists t'.\, t'.stable = t.volatile \wedge CR(s', t') \quad (8)$$

Although this technique requires manual inspection of each crash state generated during an operation, it significantly improves the scalability of the verification of SCFTL.

6 Discussion

Using an SMT solver has the advantage that once the formulae are constructed, their proofs are done fully automatically. If some of the constructed formulae cannot be solved in a reasonable time, we apply the techniques mentioned above to simplify them systematically. In total, the representation invariant (*RI*) and the abstraction relation (*AR*) contain 98 conditions; their weaker versions (*CI* and *CR*, respectively) contain 17 conditions. We also use *loop invariants* and an inductive proof rule [23] to handle large loops. With these, all but three conditions can be proven correct.

The three unverified conditions are: (i) after a successful recovery, the L2P table is an one-to-one mapping except for

invalid entries; (ii) after a successful recovery, there is sufficient space to accommodate the follow-up writes and gc_{rl}s; (iii) after a successful flush, there is sufficient space to accommodate the follow-up writes and gc_{rl}s. For the three unverified conditions, we use the *validation* technique [37] to ensure their correctness. The validator itself is formally verified (explained in §6.1) and will notify the user when our SCFTL implementation violates the property. Such notification never occurs in all of our experiments.

The Z3 verification time is 4640 seconds for a 8 GB flash disk and 6302 seconds for a 256 GB flash disk. We choose to verify a particular flash disk size at a time (rather than for all sizes) to reduce the number of quantifiers and thus improve the verification time. We also tried to change other parameters (e.g., several write bounds ranging from 2048 to 20480), the verification time ranges from 1 hour to 2 hours; interestingly, a larger value does not imply a longer time.

Table 1. Lines of code for SCFTL.

Component	**Lines of code**
SCFTL implementation	950 (C)
Snapshot consistency theorems	652 (Agda [33])
SCFTL specification	22 (Rosette [43])
Invariants & Relations	2010 (Rosette)
Ghost variables	538 (Rosette)
Proof partitions	96 (Rosette)
Flash memory model	232 (Rosette)
Core framework	1048 (Rosette)
Total	3946 (Rosette)

Table 1 shows the lines of code for SCFTL. We count the specification, loop invariants, representation invariants, abstraction relations, ghost variables, and proof partitions as our proof, resulting in a proof-to-implementation ratio of 2.8:1. The total development effort is about 6 person-months; a significant part is devoted to finding (an efficient SMT encoding of) the required invariants and scaling the verification with the techniques we introduced in §5. For the trusted computing base, we assume that (i) the flash memory is free of error, (ii) the verification tools Z3, Agda, and Serval are correct, (iii) the translation from LLVM to machine code is correct, and (iv) the LightNVM [6] Linux kernel module, which we used to host our SCFTL, is correct.

6.1 Validating unverified conditions

For each of the unverified conditions, we implement a validator to monitor if the condition is indeed satisfied during runtime. More specifically, we add to the $\mathbb{P}$-states a set of *validation variables*, including a flag indicating whether the validation fails or succeeds. The validators are not allowed to modify the $\mathbb{P}$-states other than the validation variables. We prove that the validator establishes the following postcondition: if the flag indicates a successful validation, then the condition holds. Although the validation approach is not as useful for storage systems as for compilers, we regard the approach as a last resort to circumvent the limitation of automatic verification.

Validation can also be used to incorporate unverified components into SCFTL. For example, an unverified block allocator may keep track of the block usage and allocate the block with the least amount of usage for wear leveling. A validator can then validate whether the allocated block is actually in the free block queue, and if so, returns the block. Otherwise, SCFTL falls back to the default verified behavior, e.g., allocating the first block in the free block queue.

6.2 Support for concurrency

Our verification methodology does not support concurrent SCFTL operations, and our specification has a sequential nature. However, both of them do not limit an implementation from exploiting the high degree of hardware parallelism commonly seen in modern flash disks (e.g., multiple channels and flash chips). More specifically, executing a single step (e.g., a write operation) in the specification corresponds to performing a top-level C function in the SCFTL implementation. The C function usually uses asynchronous flash commands to avoid waiting for slow flash operations to finish. This design allows multiple flash operations to be executed concurrently until a SYNC. Reordering due to the concurrency of flash operations can only be observed when a crash occurs. We describe our technique to capture reordering in §5.1.

7 Evaluation

To evaluate SCFTL, we conducted experiments designed to answer the following questions:

- Is SCFTL actually correct? (§7.1)
- How does SCFTL perform compared to other FTLs implementing different disk models? (§7.2)
- Is the guarantee of snapshot consistency provided by SCFTL useful to its upper layers? (§7.3)

All experiments were done on a host machine with a 12-core 3.2 GHz Intel i7-8700 CPU and 16 GB of DRAM. To emulate the flash memory, we run experiments on Linux 4.15 hosted by FEMU [29], a QEMU-based emulator that can emulate an Open-Channel SSD (OCSSD). We used liblightnvm [2] with the libaio [1] backend to access the underlying OCSSD in an asynchronous way. The original version of FEMU supports latency emulation for OCSSD commands, but as libaio can only issue NVMe base commands, we followed FEMU's approach to emulate latency for NVMe base commands. We validated that the results produced by the two sets of commands are consistent. The OCSSD has 4 channels, 4 dies per channel, and a total of 8 GB flash memory. We reserved 16 MB of flash memory for the full checkpoint region, 256 MB for the delta region, and set the write bound to 2048.

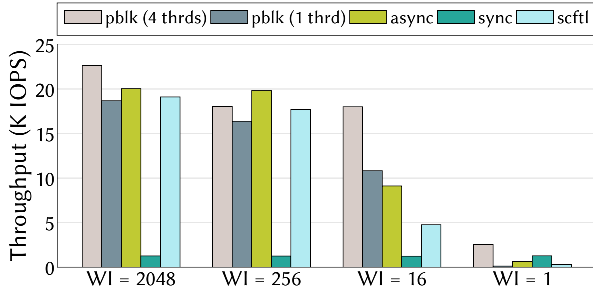

Figure 5. Throughput of FTLs under random writes with different write intervals (WI).

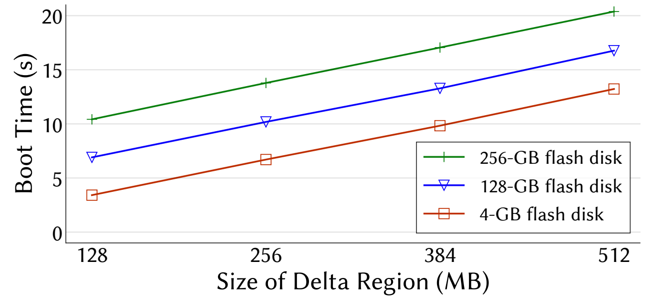

Figure 6. Boot time of SCFTL.

7.1 Stress testing and crash state simulation

To validate the correctness of SCFTL, we designed a testing framework that allows fast stress testing and crash state simulation. The framework hosts SCFTL by emulating the flash memory with DRAM, and simulates a crash by overwriting all pages affected by asynchronous WRITE or ERASE since the last SYNC with garbage data. The framework uses a *workload generator* to issue a sequence of writes and flushes to SCFTL, and simulates a crash based on a given probability. We set a higher probability for configurations that are more likely to result in corner cases (e.g., crashes during recovery).

The framework maintains a pair of arrays, volatile and stable, as golden results, and changes their states according to Figure 4. A *result checker* is then periodically activated to read every sector of SCFTL and check whether the results produced by SCFTL is consistent with the volatile array. To speed up testing, the checker only compares the first few bytes of the read data. We ran the test with 4 configurations for about 8 hours. In total, the workload generator wrote more than 1.4 TB of data, issued about 12 millions of flushes and simulated about 10 thousands of crashes. SCFTL successfully recovered from every crash state and passed all checks.

7.2 Comparing SCFTL with other FTLs

Besides SCFTL, we implemented two additional FTLs with different crash guarantees. The two FTLs implement the asynchronous (denoted by async) and synchronous (denoted by sync) disk models respectively. async is implemented in a way similar to SCFTL, except async (i) does not do checkpointing, (ii) does not comply with 2PGC (i.e., victim blocks are erased immediately after all valid data is relocated), and (iii) has no write count constraint. sync is the same as async, except sync always uses synchronous operations to access the underlying flash memory. We assume the one-page merge buffer of sync is backed by a battery (i.e., data copied to the buffer is guaranteed to be persistent); thus sync can safely ignore any flush request. We also used the state-of-the-art FTL pblk [6], which has similar features to async (e.g., both of them implement the asynchronous disk model and use a sector-level L2P), to understand the quality of our FTL implementation.

We wrote a small program to randomly issue 4 KB writes to a disk, and periodically flush the disk after a fixed number of writes. We call this period the *write interval*. For pblk, we also used a concurrent version (employing 4 threads) of the same program as a way to identify the limitation of SCFTL's sequential nature. Figure 5 shows the average throughput. We first draw two conclusions: (i) Our baseline FTL, async (3rd bars), has a performance characteristic similar to pblk (2nd bars). (ii) Concurrent workloads (1st bars), in general, have higher total throughput than sequential workloads (2nd bars); but the improvement is less obvious when the write interval is higher (e.g., WI = 2048 and WI = 256) because the underlying flash memory has fewer idle resources to serve the concurrent requests. Next, we compare SCFTL with async and sync.

When the write interval is set to 2048, SCFTL throughput is within 5% of async on average; with the write interval set to 256, SCFTL throughput is still within 11%. In both settings, SCFTL outperforms sync by more than 14*x*. With the write interval set to 16, SCFTL throughput drops to 53% of async, but still outperforms sync by 3.8*x*. When the write interval is reduced to 1, SCFTL throughput is only one half of async and one quarter of sync, because SCFTL writes one additional delta page on receiving a flush. Note that the last setting is not a reasonable usage of SCFTL, but it shows the overhead of SCFTL under the worst-case scenario.

We also analyze the write and flush latency and make the following observations: (i) SCFTL has a slightly higher average flush latency (12 ms) than async (10 ms) because SCFTL writes an additional delta page during a flush. (ii) SCFTL has a higher maximum flush latency (398 ms) due to full checkpointing; we can reduce the latency with a hybrid setting (similarly to the WAFL file system [22] which puts a log of requests on non-volatile memory and other data on slower disks), in which the full checkpoints go to memory technologies with a lower latency (e.g., SLC flash and 3D XPoint memory [21]), and other data stays in ones with a higher latency but a lower cost (e.g., MLC and TLC flash).

Finally, Figure 6 shows the boot time of SCFTL. We have not yet implemented optimizations for recovery to reduce the boot time, so the boot time is nearly the same regardless of whether there is a graceful shutdown or where a crash occurs. In general, the boot time is directly proportional to the size of the delta region and the size of the logical address space.

7.3 Modifying xv6 with SCFTL

To understand the usefulness of snapshot consistency guaranteed by SCFTL, we used xv6 [16], a simple *log-based* file system, as our example. In order to prevent any file system inconsistency (e.g., a directory entry pointing to a free inode)

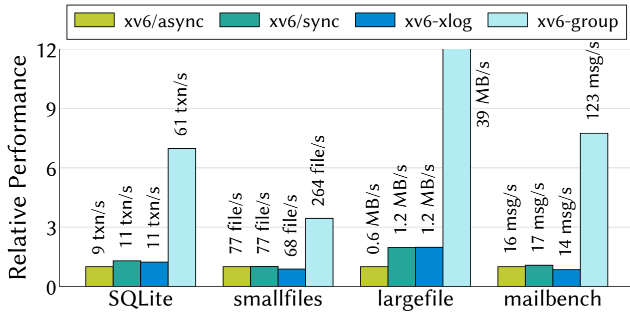

Figure 7. Performance of xv6 on different FTLs. SQLite generates 1K insert transactions followed by 1K update transactions; smallfiles repeatedly creates a file, writes 100 bytes of data to it, and calls `fsync`; largefile writes 4 MB of data to a file and calls `fsync` for every 1 MB; mailbench models a mail server running on the sv6 operating system [14]. To ensure durability, mailbench invokes an `fsync` for each message. We ran each workload 5 times and reported the average. The standard deviation is less than 8%.

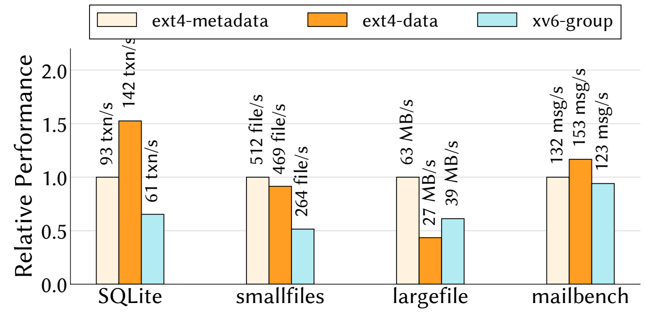

Figure 8. xv6 on SCFTL vs. ext4 on pblk. When running `mailbench` on `ext4-metadata` that does not guarantee the ordering between data and metadata, we invoked one additional `fdatasync` on the temporary file [12, Figure 1].

due to a crash occurring in the middle of a system call, xv6 uses a *write-ahead log* for atomically writing multiple sectors of data to its underlying disk. Such atomicity, however, can be easily achieved with SCFTL. We thus modified xv6 to bypass its log so that data does not need to be written twice, once to the log and once to its actual location. We further modified xv6 to support a common optimization known as *group commit*, which groups multiple system calls into one *transaction*, to reduce the number of flushes. With group commit, xv6 only issues a flush when a transaction is full or on receiving an `fsync`. The implementation is rather easy with SCFTL; we changed less than 30 lines of code of xv6. We compared the two modified versions of xv6 on SCFTL (denoted by xv6-xlog and xv6-group, respectively) with the original xv6 on the asynchronous and synchronous disks used in §7.2 (denoted by xv6/async and xv6/sync, respectively). We used existing file system benchmarks [14, 40] to evaluate the performance.

Figure 7 shows the results. The performance of xv6-xlog is only on par with that of xv6/async and xv6/sync. Although xv6-xlog has reduced the use of writes and flushes via bypassing the log, issuing a flush at the end of each system call would inevitably result in a small write interval, for which SCFTL does not perform very well as shown in Figure 5. xv6-group performs much better than the other three as the write interval becomes larger when multiple system calls are grouped together. The performance difference is particularly obvious for largefile, where `fsyncs` are less frequent.

Note that while xv6/async, xv6/sync, and xv6-xlog guarantee *immediate durability*, that is, the result of a system call is successfully stored in the disk after the call returns, xv6-group only guarantees system calls before the last `fsync` are persisted, and system calls after the last `fsync` will not be reordered. This property is also known as *sequential crash consistency* [7]. In practice, sequential crash consistency is a very strong property and is what most application developers actually require [35].

Finally we compare our group commit version of xv6 with the state-of-the-art storage stack: ext4 on pblk. We mounted ext4 with two configurations: The default metadata journaling mode `data=ordered` (denoted by ext4-metadata), and the data journaling mode `data=journal,journal_async_commit` (denoted by ext4-data). ext4-metadata does not journal data but it issues one more flush than ext4-data when committing an ext4 transaction.

Figure 8 shows the results. xv6-group performance is 7% to 49% lower than ext4-metadata. Compared with ext4-data, the performance difference is more divergent. For SQLite, xv6-group performance is only 43% of ext4-data; but for largefile, xv6-group is more than 1.4*x* of ext4-data. Such divergence can be explained by the behavior of the workloads: SQLite frequently issues `fsyncs` and causes the performance of SCFTL to degrade; on the other hand, largefile issues much less `fsyncs` and a huge amount of data is written in the journal of ext4-data. We believe the performance difference between our modified xv6 and ext4 is mainly owing to the simplicity of xv6. This can be improved by, e.g., optimizing xv6 with in-memory representations for file system operations [5].

8 Conclusion

We believe that our verified SCFTL brings new opportunities for the design of the storage stack. We demonstrate that starting at a lower-level of abstraction can make verifying crash safety easier while still resulting in an efficient system. Formal specification and verification give the user a clear picture and strong confidence of what he/she can assume while designing the upper layers of the storage stack. Our experimental results show that SCFTL can provide a strong crash guarantee without compromising its performance if upper layers can carefully reduce the flush frequency.

There are several avenues for future work. For instance, we would like to extend the work to cover upper layers, such as file systems or database systems, of the storage stack. With a careful design that fully utilizes the advantage of SCFTL, we believe it is likely that we can obtain a verified and yet efficient upper-layer system. The FTLs used in commercial products usually come with several optimizations, e.g., hot-cold data separation and wear leveling. We plan to extend SCFTL to use those optimizations.

Acknowledgments

We thank our shepherd, Frans Kaashoek, and the anonymous reviewers for their valuable feedback. This work was supported in part by Academia Sinica under grant no. AS-CDA-107-M05 and the Ministry of Science and Technology (MOST) of Taiwan under grant nos. 109-2628-E-001-001-MY3, 109-2222-E-001-002-MY3, 107-2923-E-001-001-MY3, 108-2221-E-001-001-MY3, and 108-2221-E-001-004-MY3.

References

[1] Aio. http://man7.org/linux/man-pages/man7/aio.7.html.

[2] liblightnvm. http://lightnvm.io/liblightnvm/.

[3] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 175–188, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. Bit-vector interpolation and quantifier elimination by lazy reduction. In *Formal Methods in Computer Aided Design*, FMCAD '18, pages 1–10, 2018.

[5] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 69–86, New York, NY, USA, 2017. Association for Computing Machinery.

[6] 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, page 359–373, USA, 2017. USENIX Association.

[7] 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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 83–98, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 209–224, USA, 2008. USENIX Association.

[9] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yun-Sheng Chang and Ren-Shuo Liu. OPTR: Order-preserving translation and recovery design for SSDs with a standard block device interface. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1009–1023, USA, 2019. USENIX Association.

[12] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4), January 2015.

[15] 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 Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 197–212, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Russ Cox, M. Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2020. https://pdos.csail.mit.edu/6.828/2020/xv6.html.

[17] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 15–28, New York, NY, USA, 1993. Association for Computing Machinery.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08/ETAPS '08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Inside a verified flash file system: Transactions and garbage collection. In *Revised Selected Papers of the 7th International Conference on Verified Software: Theories, Tools, and Experiments - Volume 9593*, VSTTE '15, page 73–93, Berlin, Heidelberg, 2015. Springer-Verlag.

[20] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, oct 2016.

[21] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[22] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC '94, page 19, USA, 1994. USENIX Association.

[23] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] Ben Hutchings. [patch 3.2 027/115] jbd2: fix fs corruption possibility in jbd2_journal_destroy() on umount path. April 2016. https://lkml.org/lkml/2016/4/26/1230.

[25] 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 SIGMOD International Conference on Management of Data*, SIGMOD '13, page 97–108, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. on Consum. Electron.*, 48(2):366–375, May 2002.

[27] Greg Kroah-Hartman. [patch 4.14 138/267] jbd2: Fix possible overflow in jbd2_log_space_left(). December 2019. https://lkml.org/lkml/2019/12/16/1638.

[28] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST '15, page 273–286, USA, 2015. USENIX Association.

[29] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The case of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, page 83–90, USA, 2018. USENIX Association.

[30] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[31] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 33–50, USA, 2018. USENIX Association.

[32] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.

[34] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. *ACM Trans. Storage*, 13(3), September 2017.

[36] 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 Conference on Operating Systems Design and Implementation*, OSDI '14, page 433–448, USA, 2014. USENIX Association.

[37] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, page 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[38] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 147–160, USA, 2008. USENIX Association.

[39] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), August 2013.

[40] 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.

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

[42] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 1–16, USA, 2016. USENIX Association.

[43] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 530–541, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. Efficiently solving quantified bitvector formulas. *Form. Methods Syst. Des.*, 42(1):3–23, February 2013.

[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, page 211–226, USA, 2018. USENIX Association.

[46] 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 Conference on Operating Systems Design and Implementation*, OSDI '14, page 449–464, USA, 2014. USENIX Association.