Transactional Flash

Vijayan Prabhakaran, Thomas L. Rodeheffer, Lidong Zhou
Microsoft Research, Silicon Valley
{*vijayanp, tomr, lidongz*}*@microsoft.com*

Abstract

Transactional flash (TxFlash) is a novel solid-state drive (SSD) that uses flash memory and exports a transactional interface (`WriteAtomic`*) to the higher-level software. The copy-on-write nature of the flash translation layer and the fast random access makes flash memory the right medium to support such an interface. We further develop a novel commit protocol called* cyclic commit *for TxFlash; the protocol has been specified formally and model checked.*

Our evaluation, both on a simulator and an emulator on top of a real SSD, shows that TxFlash does not increase the flash firmware complexity significantly and provides transactional features with very small overheads (less than 1%), thereby making file systems easier to build. It further shows that the new cyclic commit protocol significantly outperforms traditional commit for small transactions (95% improvement in transaction throughput) and completely eliminates the space overhead due to commit records.

1 Introduction

Recent advances in NAND-based flash memory have made solid state drives (SSDs) an attractive alternative to hard disks. It is natural to have such SSDs export the same block-level read and write APIs as hard disks do, especially for compatibility with current systems. SSDs are thus simply "hard disks" with different performance characteristics.

By providing the same API as disks, there is a lost opportunity of new abstractions that better match the nature of the new medium as well as the need from applications such as file systems and database systems. In this paper, we propose a new device called Transactional Flash (TxFlash) that exports such an abstraction. TxFlash is an SSD exposing a linear array of pages to support not only read and write operations, but also a simple *transactional* construct, where each transaction consists of a series of write operations. TxFlash ensures atomicity, *i.e.*, either all or none of the write operations in a transaction are executed, and provides isolation among concurrent transactions. When committed, the data written by a transaction is made durable on the SSD.

The atomicity property offered by the transactional construct has proven useful in building file systems and database systems that maintain consistency across crashes and reboots. For example, a file creation involves multiple write operations to update the metadata of the parent directory and the new file. Often, a higher-level system employs copy-on-write (CoW) or a variant of it, such as write-ahead-logging (WAL) [13], to ensure consistency. The essence of these mechanisms is to avoid *in-place* modification of data.

Although known for decades, these mechanisms remain a significant source of bugs that can lead to inconsistent data in the presence of failures [30, 16, 29], not to mention the redundant work needed for each system to implement such a mechanism. For example, in Linux, the common journaling module (jbd) is used only by Ext3 [28], although there are several other journaling file systems, such as IBM JFS [3] and XFS [26].

Having a storage layer provide a transactional API reduces the complexity of the higher level systems significantly and improves the overall reliability. Indeed, previous work has attempted to provide such constructs on hard disks using CoW and logging [5, 6, 22]. Unfortunately, one common side effect of CoW techniques is the *fragmentation* of the linear address space, *i.e.*, CoW tends to scatter related pieces of information over the disk when updating them. Reading those fragmented pieces of related information requires seeks, leading to poor performance. To mitigate this performance problem, systems that implement CoW also employ some form of checkpointing and cleaning [20], where related pages are reorganized into their home locations. However, cleaning costs can themselves be high [24].

The significant extra complexity at the disk controller layer and the poor read performance due to fragmentation are some of the main obstacles for providing a transactional API on hard disks. SSDs mitigate both problems, making it an ideal medium for supporting transactions. Modern SSD controllers already implement variants of CoW for performance reasons. Furthermore, fragmentation does not lead to performance degradation on SSDs because random read accesses are fast.

We develop a new *cyclic commit* protocol for TxFlash to allow efficient implementation of transactions. Cyclic commit uses per-page metadata to eliminate the need for a separate commit record as in a standard commit protocol. Traditionally, the existence of a commit record is used to judge whether a transaction is committed or not. In our cyclic commit protocol, the judgment is based on the metadata stored along with the data. The result is a better commit protocol in terms of performance and space overheads.

We evaluate TxFlash in three complementary ways.

- We formally specify the cyclic commit protocol in TLA+ and check it using the TLC model checker [10].
- We design and implement TxFlash as an extension to an SSD simulator from previous work [1]. The resulting simulator is used to compare the variants of cyclic commit with traditional commit, as well as to assess the overheads of transactional support by comparing TxFlash with the basic SSD.
- We develop TxExt3, a version of the Linux Ext3 file system modified to exploit TxFlash's transactional API. To obtain realistic end-to-end performance numbers, we run TxExt3 on a real SSD using an intermediate pseudo-device driver to emulate the TxFlash firmware.

Compared to traditional commit, cyclic commit improves the performance significantly, especially for small transactions, while eliminating the space overhead completely. For transactions less than 100 KB in size, cyclic commit improves transaction throughput by 95% over traditional commit. For transactions larger than 4 MB, cyclic commit performs as well as traditional commit. Our simulation results show that TxFlash can provide the transactional capabilities with negligible overhead (less than 1%) when compared to an SSD. Finally, for I/O intensive synchronous applications, TxExt3 reduces the run time by as much as 65%, in both data and metadata journaling modes. However, the benefits for compute-intensive applications are small (about 2%).

The rest of the paper is organized as follows. Next, we make a case for TxFlash and explain its API and architecture (§2). The core cyclic commit protocol and its variations are the subject of the following section (§3). We follow this with the details of the implementation (§4) and evaluation (§5). Then, we cover the related work (§6) and finally conclude (§7).

2 The Case for TxFlash

The rationale for TxFlash is deeply tied to the fundamentals of the flash based SSDs, which are covered in this section. We then describe the API and the architecture of TxFlash, followed by a discussion of the rationale for TxFlash. For the rest of the paper, we interchangeably use "page" or "logical page" to refer to pages as used by a higher-level system. A page on the stable storage is referred as a "physical page" or "flash page".

2.1 SSDs: A Primer

Similar to disks, NAND-flash based SSDs provide a persistent medium to store data. Unlike disks, however, SSDs have no mechanically moving parts, yielding drastically different performance characteristics from disks.

An SSD consists of multiple flash packages that are connected to a controller, which uses some volatile memory for buffering I/O requests and maintaining internal data structures. A flash package is internally made up of several planes, each containing thousands of blocks; each block in turn consists of many 4 KB pages. In addition to the data portion, a flash page also contains a metadata portion: for every 4 KB page, there is a corresponding 128 bytes for metadata such as checksums and error correction codes. Reads and writes can be issued at page granularity and SSDs can be designed to ensure atomicity for a single page write, covering both the data and the metadata.

Another characteristic of NAND-flash is that, after a physical page has been written, it must be *erased* before any subsequent writes, and erasures must be performed at the block granularity. Since a block erase is costly (1.5 ms), SSDs implement a *flash translation layer* (FTL) that maintains an in-memory *remap table*, which maps logical pages to physical ones. When a logical page is written, the FTL writes the data to a new physical page and updates the mapping. Essentially, the FTL implements CoW to provide the illusion of in-place writes and hide the cost of block erasures.

The in-memory remap table must be reconstructed during boot time. An SSD can use the metadata portion of a physical page to store the identity and version number of the logical page that is stored.

The FTL further maintains a list of free blocks. Because of CoW, obsolete versions of logical pages may be present and should eventually be released to generate free space. SSDs implement a *garbage collection* routine that selects a block, copies valid pages out of the block, erases it, and adds it to the free-blocks queue. The remapping and garbage collection techniques are also used to balance the wearing down of different blocks on the flash, often referred to as *wear-leveling*.

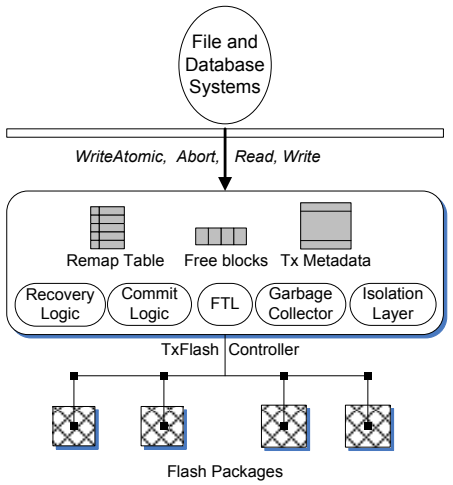

Figure 1: **Architecture of a TxFlash Device.** *The controller runs the logic for ensuring atomicity, isolation, and recovery of transactions. Data and metadata are stored on the flash packages.*

2.2 TxFlash API and Architecture

Figure 1 shows a schematic of TxFlash. Similar to an SSD, TxFlash is constructed with commodity flash packages. TxFlash differs from an SSD only in the API it provides and in the firmware changes to support the new API.

Transactional Model. TxFlash exports a new interface, `WriteAtomic`$(p_1 \ldots p_n)$, which allows an application to specify a transaction with a set of page writes, p_1 to p_n. TxFlash ensures atomicity, *i.e.*, either all the pages are written or none are modified. TxFlash further provides isolation among multiple `WriteAtomic` calls. Before it is committed, a `WriteAtomic` operation can be aborted by calling an `Abort`. By ensuring atomicity, isolation, and durability, TxFlash guarantees consistency for transactions with `WriteAtomic` calls.

In addition to the remap table and free-blocks queue maintained in an SSD, TxFlash further keeps track of in-progress transactions. When a transaction is in progress, the isolation layer ensures that no conflicting writes (*i.e.*, those updating the same pages as the pending transactions) are issued. Once a transaction is committed, the remap table is updated for all the pages in the transaction.

At the core of TxFlash is a commit protocol that ensures atomicity of transactions despite system failures. A commit protocol includes both a commit logic that is executed when a transaction is committed, and a recovery logic that is executed at boot time. The latter reconstructs the correct mapping information based on the information persisted on the flash. The actions of other modules, such as the garbage collector, depend on the actual commit protocol used.

2.3 Rationale for TxFlash

There are two main points in favor of TxFlash: utility and efficiency. TxFlash is useful because its API can benefit a large number of storage applications. TxFlash is efficient because the underlying SSD architecture matches the API well.

Interface Design. We choose to support a limited notion of transactions in TxFlash because we believe this choice reflects a reasonable trade-off between complexity and usefulness. The `WriteAtomic` interface is desirable to any storage system that must ensure consistency of multi-page writes despite system failures; file systems and database systems are known examples.

We choose not to implement full-fledged transactions, where each transaction consists of not only write operations, but also read operations. This is because they introduce significant additional complexity and are overkill for applications such as file systems.

Compatibility is often a concern for a new API. This is not an issue in this case because we preserve the simple disk APIs so that existing systems can be run directly on TxFlash. However, by using the additional transactional constructs certain parts of a system can be made simpler and more efficient. We show later in the paper (§4) that while Ext3 can be run directly on TxFlash, parts of its journaling code can be simplified to use the new `WriteAtomic` construct.

Transactions on SSDs. Compared to hard disks, SSDs are particularly ideal for supporting transactions for the following reasons.

• Copy-on-write nature of SSDs. Extending a log-structured system to support transactions is not new [23]. The FTL already follows the CoW principle because of the write-erase-write nature of the flash pages and wear-leveling. Extending FTL to support transactions introduces relatively little extra complexity or overhead.

• Fast random reads. Unlike hard disks, fragmentation is not an issue in SSDs, again because of their inherent solid-state nature: an SSD can rapidly access random flash-memory locations in constant time. Although SSDs perform cleaning for freeing more re-usable space, there is no need for data re-organization for locality.

• High concurrency. SSDs provide a high degree of concurrency with multiple flash packages and several planes per package, and multiple planes can operate concurrently. Enabled with such high parallelism, SSDs can support cleaning and wear-leveling without affecting the foreground I/O.

• New interface specifications. Traditional storage interfaces such as SATA do not allow the devices to export new abstractions. Since SSDs are relatively new, alternative specifications can be proposed, which may provide the freedom to offer new device abstractions.

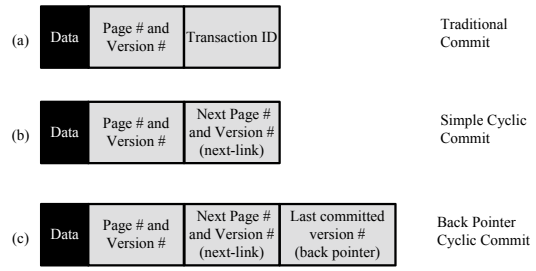

Figure 2: **Page Format.** *Information stored in the metadata portion of a flash page by various commit protocols.*

3 Commit Protocols

To ensure the atomicity of transactions, TxFlash uses a commit protocol. The protocol specifies the steps needed to commit a transaction, as well as a recovery procedure. The recovery procedure is executed after a system reboot to determine which transactions are committed based on the persistent state on the storage. Commit protocols tend not to update the data in-place and therefore invariably require a separate garbage collection process, whose purpose is to release the space used by obsolete or aborted transactions.

3.1 Traditional Commit (TC)

Modern journaling file systems such as Ext3 [28], IBM JFS [3], XFS [26], and NTFS [25] use a form of redo logging [13] for atomicity; we refer to this mechanism as the *traditional commit* protocol. In traditional commit, new data for each page is written to the storage device as an *intention record*, which contains a data portion and a metadata portion. The metadata portion stores the identity of the page and the transaction ID as shown in Figure 2(a). Once all the writes of the intention records have completed successfully, a *commit record* is written to the storage device. Once the commit record is made persistent, the transaction is committed. When recovering during a reboot, traditional commit decides whether a transaction is committed or not based on the existence of the corresponding commit record.

Typically, the intention records and the commit records are written into a log. The updates in committed intention records are written in-place to the home locations by a *checkpoint* process. Once checkpointing is completed, all the intention records and commit records are garbage collected by truncating the log.

Traditional Commit on SSDs. With an SSD as the underlying storage device, thanks to the indirection provided by the remap table, no separate checkpointing is necessary: the logical pages can be remapped to the new locations when a transaction commits. Also, all writes within the same transaction can be issued concurrently, thereby exploiting the inherent parallelism on an SSD.

However, the need for the separate commit record in traditional commit may become particularly undesirable. The commit record write must be issued only after all the intention record writes are completed; such write ordering introduces the latency of an extra write per transaction. Because of the absence of a separate checkpointing process, a special garbage collection process is needed for commit records: a commit record can be released only after all the intention records of the corresponding transaction are made obsolete by later transactions. Both the performance and space overheads introduced by commit records are particularly significant for small transactions. Group commit [7] was designed to reduce some of these overheads but it works well only when there are multiple operations that can be delayed and grouped together.

3.2 Simple Cyclic Commit (SCC)

Instead of using commit records to determine whether a transaction is committed or not, a *cyclic commit* protocol stores a link to the next record in the metadata of an intention record (*i.e.*, the logical page of an SSD) and creates a cycle among the intention records of the same transaction. This eliminates the need for a separate commit record for each transaction, thereby removing the space and performance overheads.

Figure 2(b) shows the intention record used by the cyclic commit, where the next page and version numbers are additionally stored in the metadata portion as the *next-link*. For each transaction, the next-link information is added to the intention records before they are concurrently written. The transaction is *committed* once all the intention records have been written. Starting with any intention record, a cycle that contains all the intentions in the transaction can be found by following the next-links. Alternatively, the transaction can be *aborted* by stopping its progress before it commits. Any intention record belonging to an aborted transaction is *uncommitted*.

In the event of a system failure, TxFlash must be restarted to recover the last committed version for each page. The recovery procedure starts by scanning the physical pages and then runs a recovery algorithm to classify the intention records as committed or uncommitted and identify the last committed version for each page based on the metadata stored in the physical pages.

We use $\mathcal{S}$ to refer to the set of intention records (in terms of their logical page numbers and versions) obtained by scanning the stable storage and $\mathcal{R}$ for the set of intention records that are referenced by the next-link field of any intention record in $\mathcal{S}$. All records in $\mathcal{S} \ominus \mathcal{R}$ are present on the storage, but not referenced by any other record ($\ominus$ represents set difference); similarly, all records

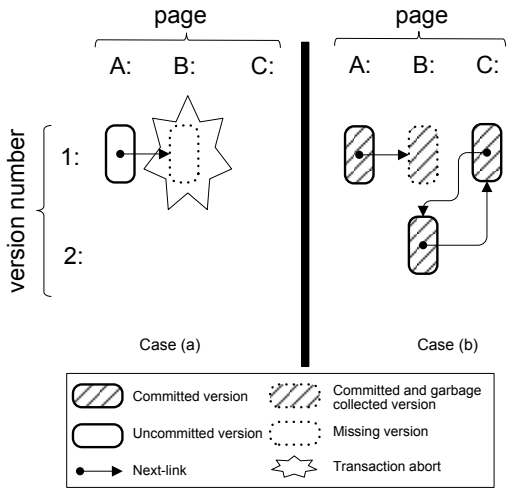

Figure 3: **Ambiguous Scenario of Aborted and Committed Transactions.** *Two cases with a broken link to B_1: an aborted transaction on the left, where B_1 was not written; a committed transaction on the right, where B_1 was superseded and cleaned.*

in $\mathcal{R} \ominus \mathcal{S}$ are referenced by some record in the stable storage but not present themselves. For an intention record r, we use $r.next$ to refer to the next-link in r.

The following Cycle Property from cyclic commit is fundamental to the recovery algorithms. It states that the classification of an intention record can be inferred from its next-link; this is because they belong to the same transaction. It further states that, if a set of intention records forms a cycle, then all of them are committed.

Cycle Property. For any intention record $r \in \mathcal{S}$, r is committed if and only if $r.next$ is committed.

If there exists a set T of intention records $\{r_i \in \mathcal{S} \mid 0 \leq i \leq n-1\}$, such that for each $0 \leq i \leq n-1$, condition $r_i.next = r_{(i+1) mod\ n}$ holds, then any $r \in T$ is committed.

It is worth noting that a break in a cycle (i.e., $r \in \mathcal{S}$ and $r.next \notin \mathcal{S}$) is not necessarily an indication that the involved intention record is uncommitted. Figure 3 illustrates this case. In this example, pages are referred to by the letters A through C and the version numbers are 1 and 2. Next links of intention records are shown by arrows. In this Figure, various versions are labeled as to whether they are committed (crosshatch fill) or uncommitted (white fill). Missing versions are indicated by a dotted border. We use "P_i" for version i of page P. Consider the scenario where A_1 has its next-link set to B_1, but B_1 does not exist on the SSD. There are two cases that could lead to the same ambiguous scenario: in the first case, as shown in Figure 3(a), the transaction with A_1 and B_1 was aborted and B_1 was never written; in the second case, as shown in Figure 3(b), the transaction with A_1 and B_1 commits, followed by another successful transaction that creates B_2 and C_1, making B_1 obsolete and causing B_1 to be garbage collected. In the first case, A_1 belongs to an aborted transaction and should be discarded, while in the second case A_1 belongs to a committed transaction and should be preserved.

Observe that an intention record P_i can be garbage collected only when there is a higher version P_j ($j > i$) that is committed. SCC is based on the following SCC Invariant, which is ensured by correct initialization and handling of uncommitted intention records.

SCC Invariant: If $P_j \in \mathcal{S}$, any intention record $P_i \in \mathcal{S} \cup \mathcal{R}$ with $i < j$ is committed.

SCC Initialization. When TxFlash starts for the first time, it initializes the metadata of each page by setting the version number to 0 and the next-link to itself.

Handling Uncommitted Intention Records. If an intention record $P_i \in \mathcal{S} \cup \mathcal{R}$ belongs to an aborted transaction, to preserve the SCC Invariant, before a newer version of P is written, P_i and Q_j must be erased, where $Q_j.next = P_i$. This avoids misclassification of P_i (due to the newer version of P) and Q_j (by following the next link). That is, any uncommitted intention on the stable storage must be erased before any new writes are issued to the same or a referenced page.

SCC Garbage Collection. With SCC Invariant, any committed intention record can be garbage collected as long as a newer version of the same logical page is committed. Any uncommitted intention record can be garbage collected at any time. Garbage collection involves selecting a candidate block, copying the valid pages out of it, and erasing the block to add to the free-blocks list. TxFlash copies each valid version to another location, preserving the same metadata. If the system crashes after copying a version and before erasing the block, multiple identical versions may be present for the same page. This is a minor complication. TxFlash can pick one copy as the principal copy and treat the others as redundancies to be erased when convenient.

SCC Recovery. During recovery, SCC classifies the intention records and identifies the highest committed version for each logical page, as follows:

Since isolation is guaranteed, *i.e.*, there are no overlapping write operations for the same page, for each logical page, the recovery algorithm only has to choose between the intentions having the *highest* and the *second highest* version numbers. This is true for the following reason. The intentions having the second highest version numbers must have been committed, since the application must have completed their transactions before going on to start a subsequent transaction on the same page. The only question to answer is whether the highest version numbered intention is also committed for a page.

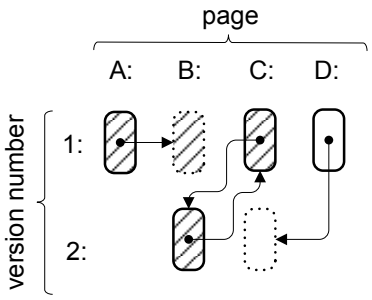

Figure 4: **An Example TxFlash System State with SCC.**

Let h_P represent the highest version number for a page P and P_h be the intention record with the highest version number. The goal of the recovery algorithm is to determine, for every page P, whether P_h is committed or uncommitted using the following analysis. Let $Q_l = P_h.next$. These values are available from the metadata portion of the P_h intention record. Let h_Q be the highest version number of any intention that exists on the storage device for page Q. There are three cases.

I. $h_Q > l$: P_h is a committed intention because Q_l is committed because of the presence of h_Q (SCC Invariant), and so is P_h (Cycle Property). For example, consider A_1 in Figure 4, whose next-link is B_1. Since the highest version for B is B_2, B_1 is committed and therefore A_1 is committed too.

II. $h_Q < l$: P_h is an uncommitted intention. The reasoning is as follows: the transaction involving P_h could not have completed, because if it had, there would be an intention of page Q with a version number at least as high as l. Consider D_1 in Figure 4, whose next-link C_2 is greater than the highest version numbered intention C_1 for page C. Therefore, D_1 is uncommitted.

III. $h_Q = l$: P_h links to another highest version numbered intention Q_h, and the answer is the same as for the intention Q_h, which may be determined recursively. If this results in a cycle, then all of the involved intentions are committed intentions (Cycle Property). For example, in Figure 4, following the next-link from B_2, a cycle is detected and B_2 is classified as committed.

For each page, the last committed intention is identified using the above analysis and the remap table is updated accordingly. Since each logical page is visited only once and only the top two versions are considered for each logical page, the running time of the SCC recovery takes $O(n)$, where n is the number of logical pages.

3.3 Back Pointer Cyclic Commit (BPCC)

The SCC has the advantage that it can be implemented relatively easily with minimal changes to the recovery and garbage collection activities normally performed by an SSD. The simplicity of SCC hinges on the SCC Invariant, which necessitates the erasure of uncommitted intentions before newer versions can be created. The needed erasure could add to the latency of the writes. We introduce Back Pointer Cyclic Commit (BPCC), a variation of the SCC, that does not require such erasures.

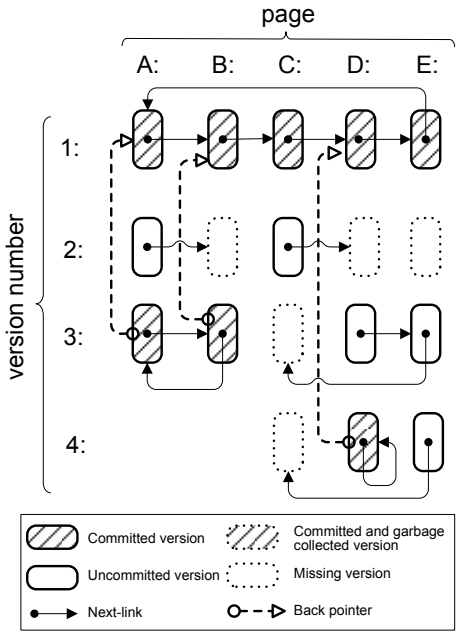

Figure 5: **An Example TxFlash System State with BPCC.**

BPCC indicates the presence of uncommitted intention records by adding more information to the page metadata. Specifically, the intention record r for a page also stores the last committed version number for that page through a *back pointer*, $r.back$. That is, before writing an intention P_k of a page P, in addition to the identity of the page and the next-link, a pointer to the last committed version of P, say P_i (where $i < k$), is also stored. Typically, the last committed version number will be the version number immediately previous to the version number of the intention (*i.e.*, $i = k-1$). This last committed version number provides enough information to determine whether uncommitted intentions are left behind by aborted transactions. Specifically, if there exists a $P_j \in \mathcal{S} \cup \mathcal{R}$, where $i < j < k$, then P_j must be uncommitted. Figure 2(c) shows the necessary additions to the metadata to include the back pointer.

For any intention record P_j, an intention record P_k with $P_k.back = P_i$ satisfying $i < j < k$ is a *straddler* of P_j as it straddles the uncommitted intention P_j. It is important to notice that a committed intention can never be straddled. For correct operation, BPCC upholds the following BPCC Invariant:

BPCC Invariant: For a highest version numbered intention record $P_h \in \mathcal{S}$, let $Q_l = P_h.next$. If there

exists a $Q_k \in \mathcal{S}$ with $k > l$ and there exists no straddler for Q_l, then P_h is committed.

BPCC Initialization. When TxFlash starts for the first time, it initializes the metadata for each page by setting the version number to 0, the next-link to itself, and the back pointer to itself.

Handling Uncommitted Intention Records. If an intention record P_i belongs to an aborted transaction or is classified as uncommitted during recovery, a newer version of P can be written with a back pointer to the committed version lower than P_i, effectively straddling P_i.

Figure 5 shows an example system state as updated by BPCC, where the back pointers are shown as dashed arrows. Consider the transaction updating A_3 and B_3. When the transaction is in progress, the last committed version of A is A_1 and therefore A_3 stores a pointer to A_1. This back pointer provides the proof that the intermediate version A_2 is uncommitted and, moreover, that any highest version numbered intention with a next-link pointer that refers to A_2 must also be uncommitted. Notice that, unlike SCC, BPCC can proceed with writing A_3 and B_3 without erasing the uncommitted intentions.

BPCC Garbage Collection. To preserve BPCC Invariant, before an intention record r can be garbage collected, the protocol must make sure that the status of the other intention records straddled by r is no longer useful during recovery. We capture such information by introducing a *straddle responsibility set* (SRS) for each intention record, as follows:

Given an intention record $P_k \in \mathcal{S}$ with $P_k.back = P_i$, the straddle responsibility set of P_k is

$$\{Q_l \in \mathcal{S} \mid Q_l.next = P_j \; and \; i < j < k\}.$$

For each Q_l in P_k's straddle responsibility set, the fact that it is uncommitted hinges on the availability of P_k's back pointer. Therefore, P_k can be garbage collected *only* after all such Q_l are erased. More precisely, for BPCC, an intention record P_i can be garbage collected if and only if $SRS(P_i)$ is empty and P_j for some $j > i$ is committed.

Various optimizations exist to remove entries from the straddle responsibility sets. This makes it possible to have certain intention records garbage collected earlier. We list some of them here.

- For any $Q_l \in SRS(P_k)$, if a higher version of Q is committed, then Q_l can be removed from $SRS(P_k)$. This is because Q_l can never in the future be the highest numbered version, once a later version is committed. Observe in Figure 5 that version B_3 straddles the next-link of A_2 but A_3 is a more recent committed version. So, A_2 can be removed from $SRS(B_3)$.
- If $Q_l \in SRS(P_j) \cap SRS(P_k)$, and $j < k$, Q_l can be removed from $SRS(P_j)$. This is because a higher version P_k straddles Q_l as well. In Figure 5, D_3 straddles the next-link of C_2, but D_4 is a later version of D and also straddles the next-link of C_2, so C_2 can be removed from $SRS(D_3)$.
- If P_j belongs to an aborted transaction and no higher version of P is committed, then $SRS(P_j)$ can be set to empty. This is because even if P_j is garbage collected, any uncommitted version of P above the highest committed version can be classified correctly, either by following its next-link or by another straddler of P with a higher version. In Figure 5, version E_4 straddles the next-link of D_3 but E_4 is later than the last committed version on page E so $SRS(E_4)$ can be set to empty.

BPCC Recovery. For BPCC, the recovery algorithm works by determining whether the highest-numbered version for a given page is committed or not. If it is committed, then it is the last committed version; otherwise the version itself indicates the last committed version through the back pointer, and all the intermediate versions between the highest-numbered version and its back-pointer version must be uncommitted.

For every page P, let P_h be the intention record with the highest version number, $Q_l = P_h.next$, and Q_h be the highest version numbered intention present for Q. The commit status of P_h is determined by the following analysis. There are three cases.

I. $h_Q > l$: In this case, check if Q_l is an uncommitted version by looking for a straddler. That is, look for some Q_i such that $i > l$ and $Q_i.back < l$. If such a straddler is present, then P_h is uncommitted, else P_h is committed (BPCC Invariant). In Figure 5, consider the next-link of C_2, which is D_2. Since the highest version numbered intention D_4 is greater than D_2 and since D_4 straddles D_2, C_2 can be decided as uncommitted.

II. $h_Q < l$: P_h is uncommitted and the reasoning is similar to the case (**II**) of SCC recovery. Consider the next-link of E_4, which is C_4 in Figure 5. Since C_4 is greater than the highest version for C (*i.e.*, C_2), E_4 is uncommitted.

III. $h_Q = l$: The next-link must be a highest-numbered version and its commit status can be determined by recursive analysis. The recursive application must check for a complete cycle, which indicates that all of the involved versions are committed (Cycle Condition). The cycle between A_3 and B_3 in Figure 5 is an example of this case.

3.4 Discussion

Dynamic Transactions. Transactions as implemented in most journaling file systems are static, *i.e.*, the pages to write are known before the transaction begins. But in general, transactions are dynamic, *i.e.*, all the writes is-

	TC	SCC	BPCC
Metadata/page	16 bytes	24 bytes	28 bytes
Space overhead	1 commit/tx	None	None
Perf. overhead	1 write/tx	None	None
Garbage collection	Simple	Simple	Complex
Recovery	Simple	Simple	Complex
Aborted transactions	Leave	Erase	Leave

Table 1: **Protocol Comparison.** *Traditional commit compared with the cyclic commit protocol variants.*

sued inside a transaction are not known before the transaction begins and they are determined only during the course of the execution. It is important to note that cyclic commit can support this more general semantics. For cyclic commit to work correctly, it is *not* necessary to know all the pages that are written within a transaction. For the next-link pointers to be filled correctly in the cycle, it is sufficient to know the next page write in the transaction. This is achieved simply by holding the current write in a buffer until the next write is issued by the transaction. When the last page is written, it is linked back to the first write of the transaction.

TxFlash for Databases. In addition to file systems, databases can benefit from the transactional support in a storage system. However, there are certain challenges that must be overcome. First, databases require the generic transactional interface with reads and writes instead of a simple `WriteAtomic` model. Second, in our current implementation we use a simple lock-based isolation technique, which may not be sufficient for a database system. We are exploring the use of a more fine-grained concurrency control mechanism that can take advantage of the multiple page versions in an SSD [18]. In addition, we may need a deadlock detection and abort mechanism in TxFlash for databases.

Metadata Overhead. In Table 1, we present a qualitative comparison of traditional commit with the two cyclic commit variants. We assume 4 bytes for transaction ID, 4 bytes for version number, and 8 bytes for logical page number.

One possible concern in cyclic commit is the space and write overheads imposed by additional pointers stored in the metadata portion of an intention record. As shown in Table 1, we need a maximum of 28 bytes to store the cyclic commit specific metadata. This still leaves enough space for larger ECCs to be stored in the future. Regarding the write overhead, the hardware specification of the Samsung flash package [21] states that when a flash memory page is written, it is recommended to write the entire 4 KB of data *and* 128 bytes of metadata to maintain correctly the on-chip error detection and correction codes. Therefore, the overhead of writing additional metadata is common for all the protocols even though the traditional commit uses less space than cyclic commit.

3.5 Summary

SCC requires that the uncommitted intentions are erased before the system updates the corresponding pages, whereas BPCC demands more metadata to be kept in each intention record and a more complicated analysis in the recovery algorithm, but it does not require the erasure of any uncommitted intentions. BPCC also requires that the intention records containing obsolete versions be reclaimed according to a certain precedence order, whereas SCC has no such requirement. Neither protocol requires any checkpointing (or reorganization) overhead for garbage collection. Depending on the overhead of erasing a storage page, the expected rate of failures and updates, and the space available to store metadata, either SCC or BPCC may be preferred.

4 Implementation

In this section, we present the implementation details of the TxFlash simulator, a pseudo-device driver, and the modified TxExt3 file system.

4.1 TxFlash Simulator

We modify the trace-driven SSD simulator from previous work [1] to develop the TxFlash simulator. The SSD simulator itself is extended from the DiskSim simulator [4] by adding an SSD model into the framework. The SSD simulator can evaluate various flash package configurations, interleaving, wear-leveling, and garbage collection techniques. The simulator maintains an in-memory remap table and a free-blocks list to process requests; the requests are stored in a per-flash-package queue. In order to reduce the recovery time, the simulator also writes a per-block summary page. During recovery, instead of reading every page, only the summary pages are read.

WriteAtomic Interface. We built our TxFlash simulator to support the `WriteAtomic` interface, a restricted form of transaction where pages written are known before the transaction commit begins. File systems issue a `WriteAtomic` command in which they pass all the page numbers of the transaction writes and receive a transaction ID; then they issue the data for each transactional page write. The simulator also supports an `Abort` command in addition to the regular `read` and `write` commands.

Other Modifications. The TxFlash simulator keeps track of the list of pages modified by an in-progress transaction to ensure isolation. This list is cleared when the transaction completes or aborts. Even though each individual page write is atomic, a block erase operation is not. That is, TxFlash can crash during a block erase, which may leave the block contents in an unpredictable

state. The page integrity can be verified if checksums are stored in its metadata, and the simulator models a 16 byte checksum per physical page. Because of its hardware limitations, TxFlash supports only bounded transactions (this is not a limitation of the cyclic commit), and our simulator is configured to support transactions of maximum size 4 MB (*i.e.*, containing up to one thousand 4 KB pages).

TxFlash supports a regular write operation because journaling file systems such as Ext3 issue single page writes to file system super blocks outside a transaction even in data journaling mode. TxFlash treats the regular `write` operation to a single page as a trivial transaction, *i.e.*, a transaction containing that single write, and this can be implemented with no additional costs.

Traditional Commit in TxFlash Simulator. To evaluate the trade-offs, we also implemented the traditional commit protocol in the TxFlash simulator. It works by writing one commit record for every transaction that modifies more than one page. Since there is no log reorganization, the commit record is preserved until all the page versions become obsolete, and the commit record serves as a proof as to whether a page version is committed or not. In order to do a fair evaluation, we tuned our traditional commit implementation to be efficient for single-page transactional write, which is handled by just writing the page version with a bit set in its metadata to indicate that it is a single-page transaction without any additional commit record. That is, for a single page write, traditional commit is as efficient as cyclic commit.

We simulate a 4 KB commit record write because the smallest write-granularity on modern NAND-based flash packages is 4 KB (similar to the 512 bytes granularity on disks). However, commit records are typically small, storing the transaction ID and a commit identifier. Therefore, simulating a 4 KB write adds unnecessary overhead. If SSD controllers support other types of byte-addressable memory such as battery-backed RAM or NOR-based flash, then commit records can be stored more efficiently on them. In such cases, the overhead in traditional commit is likely to be very small. Since there is no log reorganization, recovery in traditional commit scans through the stable storage and finds the most recent version with a commit record for each page.

4.2 TxFlash Pseudo-Device Driver

We implement a pseudo-device driver supporting the new transactional calls, `WriteAtomic` and `Abort`, through the `ioctl` interface. The driver associates a unique transaction ID with each transaction and forwards the read and write requests to the underlying storage device. A tracing framework is also built within the driver to generate traces for the TxFlash simulator. We record attributes such as page number, I/O type (read/write), time stamp, I/O size, and transaction ID in the trace.

New transactional commands, when issued, may cause a small overhead. For example, when a file system issues a `WriteAtomic` command, the page numbers from the command are copied to the TxFlash buffer and the new transaction ID generated by the TxFlash is copied back to the file system. The driver emulates such overheads to a certain extent by copying data in the system memory.

4.3 TxExt3

Our initial goal was to build the transaction support inside Ext2 with the help of TxFlash. However, this requires the abstraction of an in-core transaction that can buffer writes for write-back caching. The journaling module (jbd) of Ext3 already provides this abstraction (in addition to the redo logging facility) and therefore, instead of re-implementing the transaction abstraction in Ext2, we reuse the jbd module and the Ext3 file system.

We modify the jbd module to use the `WriteAtomic` interface to create the TxExt3 file system. Instead of writing to the journal, the commit function uses the transactional commands from the pseudo-device driver. In TxExt3, there is no separate checkpointing process as in Ext3. Therefore, once all the transactional writes are over, TxExt3 releases the in-core buffers and proceeds with the normal operations.

5 Evaluation

We evaluate our design and system through several stages: first, we show that cyclic commit outperforms traditional commit, both in terms of performance and space overheads (§5.1); second, we compare TxFlash against an SSD to estimate the overheads of transactional support (§5.2); and finally, we run benchmarks on top a real SSD to study the end-to-end file system performance improvement (§5.3). Throughout this section, by transaction we refer to the transactional writes as issued by the TxExt3 file system during its journal commit.

Workloads and Simulator Settings. We collect traces from TxExt3 under both data and metadata journaling modes by mounting it on top of the pseudo-device driver and using the driver's tracing framework. We run a variety of benchmarks: IOzone [14] is a complex microbenchmark suite and it is run in auto mode with a maximum file size of 512 MB; Linux-Build is a CPU intensive workload and it copies, unpacks, and builds the entire Linux 2.6.18 source tree; Maildir simulates the widely used maildir format for storing e-mails [2] and we run it with a distribution of 10,000 emails, whose sizes vary from 4 KB to 1 MB; TPC-B [27] simulates

a database stress test by issuing 10,000 credit-debit-like operations on the TxExt3 file system. The TxExt3 file system issues one transaction at a time, so there is little concurrency in the workload. In our experiments, the transaction size varies among the benchmarks and is determined by the sync intervals. Since IOzone and Linux-build do not issue any sync calls, their transaction sizes are quite large (over 2 MB); Maildir and TPC-B issue synchronous writes and therefore result in smaller transactions (less than 100 KB).

Ext3 provides only a limited transaction abort functionality. After a transaction abort, Ext3 switches to a fail-stop mode where it allows only read operations. To evaluate the performance of the cyclic commit under transaction aborts, we use a synthetic workload generator, which can be configured to generate a wide variety of transactions. The configuration parameters include the total number of transactions, a distribution of transaction sizes and inter-arrival times, maximum transaction concurrency, percentage of transaction aborts, and a seed for the random generator.

We configure our simulator to represent a 32 GB TxFlash device with 8 fully-connected 4 GB flash packages and use the flash package parameters from the Samsung data sheet [21]. The simulator reserves 15% of its pages for handling garbage collection. I/O interleaving is enabled, which lets the simulator schedule up to 4 I/O operations within a single flash package concurrently.

Model Checking. We verify the cyclic commit algorithm by specifying the SCC and BPCC protocols in TLA+ and checking them with the TLC model checker [10]. Our specifications model in-progress updates, metadata records, page versions, aborts, garbage collection, and the recovery process, but not issues such as page allocation or I/O timing. Our specifications check correctly up to 3 pages and 3 transactions per page; state explosion prevents us from checking larger configurations. This work is published elsewhere [19].

5.1 Cyclic Commit vs. Traditional Commit

In order to find the real performance benefits of the commit protocols, we must compare them only under transactional writes, as all of them work similarly under non-transactional writes. Therefore, in the following section, we only use traces from the data journaling mode or from the synthetic transaction generator.

Impact of Transaction Size. First, we compare cyclic commit with traditional commit to see whether avoiding one write per transaction improves the performance. The relative benefits of saving one commit write must be higher for smaller transactions. Figure 6 compares the transaction throughput of BPCC and TC under different transaction sizes. The TxFlash simulator is driven with the trace collected from a sequential writer, which varies its `sync` interval to generate different transaction sizes. Note that the performance numbers are the same for both SCC and BPCC, as they differ only when there are aborted transactions. For small transactions, all the page writes can be simultaneously issued among multiple flash packages and therefore, while BPCC takes time t to complete, TC takes $2t$ because of the additional commit write and write ordering, and this leads to a 100% improvement in transaction throughput. From the Figure, we observe that the performance improvement is about 95% when the transaction is of the order of 100 KB, and drops with larger transactions. Even larger transactions benefit from BPCC, for example, throughput improves by about 15% for transactions of size 1000 KB. For single page transactions, both the protocols perform similarly (not shown).

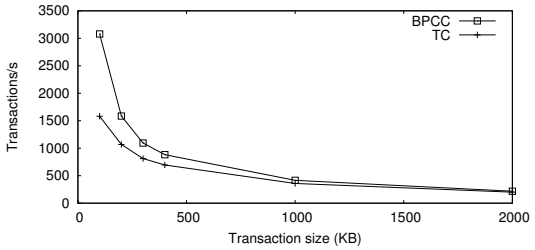

Figure 6: **Impact of Transaction Size.** *Transaction throughput vs. transaction size. TC uses a 4 KB commit record.*

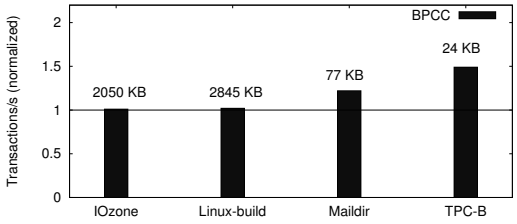

Figure 7: **Performance Improvement in Cyclic Commit.** *Transaction throughput in BPCC, normalized with respect to the throughput in TC. The throughput of IOzone, Linux-build, Maildir, and TPC-B in TC are 31.56, 37.96, 584.89, and 1075.27 transactions/s. The average transaction size is reported on top of each bar.*

Performance Improvement and Space Overhead. Next, we compare the commit protocols under macro benchmarks. Figure 7 plots the transaction throughput in TxFlash with BPCC and it is normalized with respect to the throughput in TC under various macro benchmarks. Since IOzone and Linux-build run asynchronously with large transactions, BPCC does not offer any benefit (less than 2% improvement in transaction throughput). On the other hand, Maildir and TPC-B stress the storage system with a large number of small transactions that cause high commit overhead in TC; under these cases, BPCC offers about 22% and 49% performance improvement for Maildir and TPC-B respectively. SCC performs similarly

	IOzone	Linux-build	Maildir	TPC-B
Space overhead	0.23%	0.15%	7.29%	57.8%

Table 2: **Space Overhead in Traditional Commit.** *Space overhead (ratio of the number of commit to the number of valid pages) for different macro benchmarks under TC.*

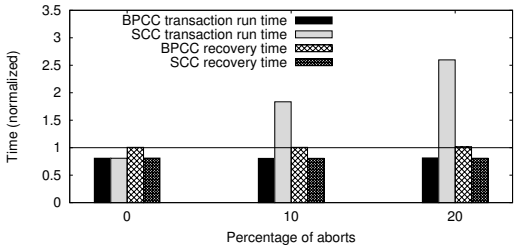

Figure 8: **Performance Under Aborts.** *Transaction time and recovery time of cyclic commit protocols under different percentages of transaction aborts. Y-axis is normalized with respect to the corresponding time in TC, which is around 1.43 ms and 2.4 s for the transaction time and recovery time, respectively.*

		TxFlash		
	SSD	+TC	+SCC	+BPCC
LOC	7621	9094	9219	9495

Table 3: **Implementation Complexity.** *Lines of code in SSD and TxFlash variants.*

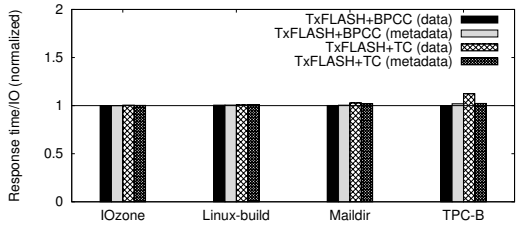

Figure 9: **TxFlash Overhead.** *I/O response time of BPCC and TC, normalized with respect to that of an SSD. The I/O response times of IOzone, Linux-build, Maildir, and TPC-B in an SSD are (0.72, 0.71), (0.70, 0.68), (0.59, 0.62), and (0.42, 0.39) ms in data and metadata journaling modes.*

under all workloads.

Table 2 presents the space overhead due to the additional commit record in TC. The space overhead is measured as the ratio of the number of commit pages to the valid pages in the system. The space overhead can increase significantly if there are small transactions updating different pages in the system (*e.g.*, Maildir and TPC-B). For large transactions, this overhead is quite small as evident from the IOzone and Linux-build entries.

Performance Under Aborts. Our next experiment compares the commit protocols under transaction aborts. We use the synthetic workload generator to create 20,000 transactions, each with an average size of 250 KB and measure the normal and recovery performance under different degrees of transactions aborts. Figure 8 presents the results, where for the normal performance we plot the average transaction time, and for the recovery we plot the time to read the stable storage and find the consistent versions. During recovery, only the per-block summary pages are scanned from the stable storage. The results are normalized with respect to TC.

From Figure 8, we draw the following conclusions. First, because SCC must erase an aborted page before letting another transaction write to the same logical page, its performance suffers as transaction aborts increase. BPCC does not incur any such overhead. Second, SCC has better recovery time than BPCC and TC because during recovery it considers only the top 2 versions for every page rather than paying the cost of analyzing all the page versions. In the presence of aborts and failures, the recovery time of BPCC also includes the time to find the appropriate straddle responsibility sets. This results in a small overhead when compared to TC (less than 1%). The recovery time can be improved through several techniques. For example, TxFlash can periodically checkpoint the remap table in the flash memory.

Protocol Complexity. Although beneficial, cyclic commit, specifically BPCC, is more complex than TC. In Table 3, we list the lines of code (LOC) for the regular SSD and TxFlash with different commit protocols. Treating the LOC as an estimator of complexity, TxFlash adds about 25% additional code complexity to the SSD firmware. Among the three commit protocols, TC is relatively easier to implement than the other two; BPCC is the most complex and most of its complexity is in the recovery and garbage collection modules.

5.2 TxFlash vs. SSD

Our next step is to measure the overhead of TxFlash when compared to a regular SSD under the same workloads. We use the traces collected from TxExt3 and run them on the TxFlash and SSD simulators. When running on the SSD simulator, we remove the `WriteAtomic` calls from the trace. Note that an SSD does not provide any transactional guarantees to the TxExt3 traces and we just measure the read-write performances.

Performance Overhead. Figure 9 presents the average I/O response time of TxFlash and SSD under various workloads in both data and metadata journaling modes. We configure TxFlash to run BPCC, but it performs similarly under SCC. From the Figure, we can notice that TxFlash with BPCC imposes a very small overhead (less than 1%) when compared to a regular SSD, essentially offering the transactional capability for free. This small overhead is due to the additional `WriteAtomic` commands for TxFlash. However, in TxFlash with TC, the additional commit writes can cause a noticeable performance overhead, especially if there are a large number of small transactions; for example, in Maildir and TPC-B under data journaling, TxFlash with TC incurs an additional overhead of 3% and 12%, respectively.

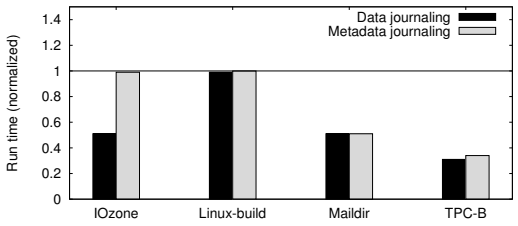

Figure 10: **End-to-End Performance.** *Benchmark run times on TxExt3-with-TxFlash under the data and metadata journaling modes, normalized with respect to their corresponding run times in Ext3-with-SSD. IOzone, Linux-build, Maildir, and TPC-B take (94.16, 43.38), (267.51, 264.40), (115.77, 146.79), and (82.73, 121.73) seconds in data and metadata journaling modes on Ext3.*

Memory Requirements. An SSD stores its remap table, free-blocks list, block-specific, and package-specific metadata in volatile memory (this is in addition to the memory that may be used for read or write buffering). For our configuration, an SSD requires about 4 MB per 4 GB flash package for its volatile structures. In addition, TxFlash needs memory to keep track of the in-progress and aborted transactions, to store the extra metadata per page, and to maintain the straddlers (only in BPCC). This requirement can vary depending on the maximum size of a transaction, the number of concurrent transactions, and the number of aborts. For a 4 GB flash package, to support a maximum of 100 concurrent transactions, with each having a maximum size of 4 MB, and an average of 1 abort per 100 transactions, we need an additional 1 MB of memory per 4 GB flash package. That is, for this configuration, TxFlash need 25% more memory than a regular SSD to support transactions.

5.3 End-to-End Improvement

While we use the simulator to understand the device-specific characteristics, we want to find out the end-to-end file system performance when running on a TxFlash. We run the pseudo-device driver on top of a 32 GB real SSD and export the pseudo-device to TxExt3 and Ext3. All our results are collected with the SSD cache disabled. Our previous evaluation from §5.2 shows that TxFlash (running either BPCC or SCC) adds little overhead when compared to SSD, and even this small overhead is emulated by the pseudo-device driver. For Ext3, the pseudo-device driver just forwards the I/O requests to the SSD.

We run the benchmarks on TxExt3 and Ext3 mounted on the pseudo-device under both data and metadata journaling, and the results are presented in Figure 10, which plots the run time of each benchmark normalized with respect to the corresponding run time on Ext3. TxExt3 with TxFlash outperforms Ext3 for two reasons: first, on each transaction, a commit write is saved and this can result in large savings for small transactions, even in metadata journaling (for example, in Maildir and TPC-B); second, Ext3 performs checkpointing, where it rewrites the information from the log into its fixed-location, and for data journaling this overhead can be significant (for example, in IOzone). Both the absence of commit write and checkpointing combine to reduce the run time by as much as around 65% (for TPC-B). However, Linux-build is compute intensive and the improvements are less than 1% because the transactions are large and most of the checkpointing happens in the background.

File system complexity can be reduced by using the transactional primitives from the storage system. For example, the journaling module of TxExt3 contains about 3300 LOC when compared to 7900 LOC in Ext3. Most of the reduction were due to the absence of recovery and revoke features and journal-specific abstraction.

5.4 Discussion and Summary

Another possible evaluation would be to compare a file system implementing cyclic commit and running on a regular SSD with TxExt3 running on TxFlash. This would let us find out if there are performance benefits in keeping the transactional features in the file system.

However, we face several limitations in building the cyclic commit inside a file system. First, current SSDs do not export the metadata portion of physical pages. As a result, cyclic commit may not be implemented as efficiently as described in this paper and therefore, the comparison would not be meaningful. Second, SSDs do not expose their garbage collection policies and actions. But, in BPCC, it is important to collect the obsolete pages in certain order and unfortunately, this control is not available to the file systems. Finally, if cyclic commit is implemented in a file system, it must use a variant of CoW and as a result, multiple indirection maps will be present in the system (one in the file system and the other in the SSD) that may lead to performance and space overheads.

In summary, we derive the following conclusions. First, in comparison with traditional commit, cyclic commit has the potential to improve the transaction throughput (by as much as 100%) and reduce the space overhead for small transactions, while matching the traditional performance for large transactions. Second, TxFlash with cyclic commit can provide transactional features with negligible overhead. Finally, a file system running on TxFlash can eliminate the write ordering problem and cut down the number of writes to half, resulting in large improvements for I/O intensive workloads.

6 Related Work

Mime [5] provides transaction support on disk drives using shadow copies. Mime offers the new function-

alities through visibility groups, which are used to ensure isolation between concurrent transactions. In addition to the standard `read` and `write` calls, Mime provides a richer set of APIs such as barriers. In contrast, TxFlash provides a simple `WriteAtomic` call, motivated by minimal complexity and file system support. Mime and TxFlash run on the storage controllers and share some of the implementation techniques, *e.g.*, CoW, block remapping, and recording data version in metadata. However, the underlying protocols are quite different. Mime uses the standard checkpointing and operation logging, whereas TxFlash uses cyclic commit.

Atomic recovery unit (ARU) [9] is an abstraction provided by the Logical Disk [6], which exports a general abstraction of a logical block interface to separate disk management from file system management. ARU operates at a higher level than TxFlash, which runs on the SSD controller. ARU allows both `read` and `write` operations in transactions (TxFlash, in contrast, supports only writes) and offers a more general isolation semantics for reads; however, the concurrency control for write operations must be implemented by the clients, whereas TxFlash provides isolation among multiple `WriteAtomic` operations. Blocks allocated under an uncommitted ARU must be identified and released during recovery, which is similar to the garbage collection requirements in TxFlash.

Stasis [22] is a library that provides a wide-range of highly flexible transactional storage primitives using WAL for applications that require transactional support but cannot use a database. Stasis is more flexible than TxFlash: it supports user-level operations, enables redo or undo logging, provides different concurrency control mechanisms, and supports atomic updates to large objects. We consider TxFlash and cyclic commit as complementary to Stasis. For example, Stasis can implement cyclic commit as one of the commit protocols.

One of the main differences between TxFlash and other disk-based transactional systems like Mime and ARU is that disk-based systems must reorganize the data for improved read performance, whereas TxFlash does not. In fact, it is harder to reorganize data in certain systems because the logical relationship between two disk blocks is not known at the disk controller.

Rio file cache is a recoverable area of main memory and Vista is a user-level library, and together, they provide light-weight transactional features that can be used to eliminate costlier synchronous disk I/O operations [11]. Rio Vista delivers excellent performance by avoiding the redo log and system calls and by using only one memory copy. TxFlash and Rio Vista operate at different layers of storage hierarchy; since Rio operates at a higher level (main memory), it works only for working sets that fit in main memory. Moreover, Rio does not provide isolation, while TxFlash offers this guarantee.

Park *et al.* [15] propose an atomic write interface for flash devices, taking advantage of the non-overwrite nature of flash pages. They store the transaction IDs on all the pages and write a commit record with the transaction ID to ensure atomicity. They modify the FAT file system to use the new interface and run it on top of an emulator. Since file system buffering can complicate the construction of transactions, their modified FAT file system runs synchronously. In contrast, we use the cyclic commit and modify Ext3, which already buffers transactions.

Transactional Flash File System (TFFS) [8] is built for NOR-based flash devices on embedded microcontrollers and provides transactional guarantees with a richer set of APIs. Unlike TxFlash, TFFS supports both reads and writes within a transaction, but write operations are all synchronous. Similar to other systems, TFFS uses an indirection called logical pointers and a variant of CoW called versioned tree structures to implement transactions. TFFS also allows non-transactional operations but does not guarantee any serializability.

Other flash-based file systems, such as JFFS2 [17] and YAFFS2 [12], have been designed for embedded devices. They use variations of log-structured design [20] and run directly on top of flash, sidestepping the FTL to avoid the cost of double-layering. Unlike TxFlash, which reads only the summary pages, JFFS2 scans the entire flash memory to rebuild its data structures; YAFFS2 carefully avoids this using checkpoints. None of these file systems provide transactional support.

Since file systems have higher-level semantic knowledge, *e.g.*, whether a page is free or not, they can do better garbage collection than a storage controller. Such information can be quite useful in TxFlash, not only in garbage collection, but also to quickly recover by not examining free pages.

7 Conclusion

In this paper, we revisit the concept of transactional support in the storage device in light of a new storage medium, the flash memory. The unique properties of the medium demand new designs in order to provide better performance and space benefits. The main contribution of this work is the novel cyclic commit protocol, which ensures atomicity by using the additional metadata on physical pages, thereby removing the overheads associated with a separate commit record. We design and implement two variants of the cyclic commit, SCC and BPCC, and both perform better than the traditional commit, especially for small transactions.

We learned a few things along the way. First, model checking our protocols helped us not only verify their correctness, but also understand *why* the protocols are

correct. Moreover, the model checker pointed out flaws in the alternative designs we investigated, which we would have missed otherwise. Second, actual implementation can bring out issues that are otherwise missed. For example, we came across complex interactions between garbage collection, block allocation, and in-progress transactions in our simulation (we did not model check some of them for simplicity reasons) and fixed those corner cases. Finally, we believe that hardware innovations can often bring new software designs; this is true in the case of cyclic commit, which was motivated by developing a commit protocol for flash memory.

8 Acknowledgments

We are grateful to our shepherd, Lorenzo Alvisi, for his valuable and detailed feedback on our paper. We also thank James Hamilton, the members of Microsoft Research Silicon Valley, and the anonymous reviewers for their excellent suggestions and comments.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 57–70, June 2008.

[2] D. J. Bernstein. Using maildir format. http://cr.yp.to/proto/maildir.html.

[3] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[4] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[5] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[6] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, December 1993.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 1–8, June 1984.

[8] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 89–104, April 2005.

[9] R. Grimm, W. C. Hsieh, W. de Jonge, and M. F. Kaashoek. Atomic Recovery Units: Failure Atomicity for Logical Disks. In *International Conference on Distributed Computing Systems (ICDCS '96)*, pages 26–37, May 1996.

[10] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] D. Lowell and P. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, October 1997.

[12] C. Manning. YAFFS: Yet Another Flash File System. `http://www.aleph1.co.uk/yaffs`, 2004.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[14] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[15] S. Park, J. H. Yu, and S. Y. Ohm. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics*, pages 155–160, June 2005.

[16] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 206–220, Oct 2005.

[17] Red Hat Corporation. JFFS2: The Journalling Flash File System. `http://sources.redhat.com/jffs2/jffs2.pdf`, 2001.

[18] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[19] T. L. Rodeheffer. Cyclic Commit Protocol Specifications. Technical Report MSR-TR-2008-125, Microsoft Research, September 2008. ftp://ftp.research.microsoft.com/pub/tr/TR-2008-125.pdf.

[20] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[21] Samsung Corporation. K9XXG08XXM Flash Memory Specification. `http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ds_k9f8g08x0m_rev10.pdf`, 2007.

[22] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, 2006.

[23] M. Seltzer. *File System Performance and Transaction Support*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1993.

[24] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, January 1993.

[25] D. A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[26] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, January 1996.

[27] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[28] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[29] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 131–146, November 2006.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 273–288, December 2004.