Möbius: A High Performance Transactional SSD with Rich Primitives

Wei Shi*, Dongsheng Wang†, Zhanye Wang* and Dapeng Ju†
*Department of Computer Science and Technology, Tsinghua University, Beijing, China 100084
†Research Institute of Information Technology and the Department of Computer Science and Technology, Tsinghua University, Beijing, China 100084
Email: {shiwei09, wangzhanye10}@mails.tsinghua.edu.cn, {wds, judapeng}@tsinghua.edu.cn

***Abstract*—Providing transactional primitives of NAND flash based *solid state disks* (SSDs) have demonstrated a great potential for high performance transaction processing and relieving software complexity. Similar with software solutions like *write-ahead logging* (WAL) and *shadow paging*, transactional SSD has two parts of overhead which include: 1) write overhead under normal condition, and 2) recovery overhead after power failures. Prior transactional SSD designs utilize *out-of-band* (OOB) area in flash pages to store transaction information to reduce the first part of overhead. However, they are required to scan a large part of or even whole SSD after power failures to abort unfinished transactions. Another limitation of prior approaches is the unicity of transactional primitive they provided.**

In this paper, we propose a new transactional SSD design named Möbius. Möbius provides different types of transactional primitives to support static and dynamic transactions separately. *Möbius flash translation layer* (mFTL), which combines normal FTL with transaction processing by storing mapping and transaction information together in a physical flash page as *atom inode*. By amortizing the cost of transaction processing with FTL persistence, MFTL achieve high performance in normal condition and does not increase write amplification ratio. After power failures, Möbius can leverage *atom inode* to eliminate unnecessary scanning and recover quickly. We implemented a prototype of Möbius and compare it with other state-of-art transactional SSD designs. Experimental results show that Möbius can at most 67% outperform in transaction throughput (TPS) and 29 times outperform in recovery time while still have similar or even better *write amplification* ratio comparing with prior hardware approaches.

I. Introduction

Transaction is an abstract firstly introduced in database field which means a serial of database operations must succeed or fail as a complete unit. It has a set of properties: *atomicity, consistency, isolation and durability* (ACID) [1], which serve to guarantee the transactional semantics. To provide data integrity and avoid data contention, transaction is then extended to file systems [2], [3] and memory systems [4], [5], [6]. *Write-ahead logging* (WAL) and *shadow paging* are two primary techniques adopted in database and file systems to guarantee the transactional semantics. The basic idea of these two techniques is to avoid in-place updates and recover data to the original version after transactions are aborted or interrupted by power failures. These techniques can protect database and file systems from data corruption. However, these techniques also introduce complexity of transactional code and a proportionate number of error-prone points in software layer [7], [8]. For example, ext3 file system (kernel version 2.6.34.15) has about 28.7% of its code to handle transactional updates on *journaling block device* (JBD) layer and MySQL 5.1.34 has about 10.1% code to handle transaction processing in its InnoDB storage engine.

Recently, with the price per bit of NAND flash decreases rapidly, NAND flash based *solid state disks* (SSDs [1]) become popular in high performance storage systems [9], [10], [11], [12]. They feature low power consumption and high responsiveness. However, NAND flash presents several disadvantages. For instance, data are physically organized in a specific manner, in blocks of pages of bits. The flash blocks must be erased before they are able to program (i.e., write) their pages again, which results in *out-of-place* updates. These limitations are somehow mitigated by a software abstraction layer, called a *flash translation layer* (FTL), which maps *logical page numbers* (LPNs) to *physical page numbers* (PPNs) and make *out-of-place* updates, and internal operations like *garbage collection* (GC) *wear leveling* transparent to upper layer.

Since SSDs have already supported *out-of-place* updates, it is intrinsic to leverage this feature to enable transactional primitives in SSDs. There are several prior approaches to realize transactional processing logic in the firmware of NAND-like storage devices. TFFS is a transactional file system for embedded flash micro-controllers [13], but flash is managed by upper layer file system and not considered as an independent device in this paper. As a type of conceptual device, transactional SSD, is firstly introduced by Prabhakaran et al. in their work [14], which allows multiple write operations to be issued as a single atomic unit with rollback support. Ouyang et al. proposed another transactional SSD design and implemented it on FusionIO firmware [15] which could support both read and write operations in transactions [16]. LightTx is another transactional SSD design presented by Lu et al. in their inspiring paper [17]. It utilized a zone-based scheme to scan and recover transaction states. MARS is another inspiring transactional *non-volatile memory* (NVM) SSD which allows applications to safely access and modify log contents to implement complex transactions simpler and more efficient [18]. Since TFFS and MARS are not NAND devices, we will only

[1] In this paper, we refer to the *NAND flash based SSD* simply as the *SSD*. Since there exists other SSD like *NVM SSD*, we will specify in such situations.

978-1-4799-5671-5/14/$31.00 ©2014 IEEE

concentrate on the other three types of transactional SSDs in the following sections. These transactional SSDs efficiently support transactional primitives in their firmware, but there are still some unsolved problems left.

First and the most important problem is the long recovery time after power failures. Since *commit flags* and other transaction information are stored in *out-of-band* (OOB) area of flash pages, existing transactional SSDs are all recovered by whole or a large part of SSD scanning to discard unfinished transactions. It will introduce a long period of offline time to repair database or file systems live in SSDs. And it is intolerable for today's data-centric applications.

Second, different types of transactional primitives are introduced for different purposes. Basically, existing transactional primitives can be classified into two main categories, which are **static** and **dynamic** transactional primitives (Details can be found in Section II-C). Each transactional SSD is designed with a single transactional primitive to support a certain type of transactions. And the unicity of transactional primitive they provided limit the usage of these transactional SSDs.

Essentially, the data of a page is persisted only after its mapping information becomes durable in SSD. The third problem of existing transactional SSDs is that they persist their metadata such as mapping information in FTL once the transactions are committed. So after each transaction is committed, there will be some extra FTL persistence writes. It will cause serious performance problems in some scenarios.

In this paper, we propose a novel transactional SSD design named Möbius and implement it on Jasmine OpenSSD platform [19] as a prototype. Möbius can be recovered from power failures quickly by scanning *atom log area* (ALA) of SSD using a *directed edge graph* (DAG) verification method. Besides, Möbius provides host interfaces which support rich transactional primitives for different types of transactions. For file system transactions, Möbius provides *static primitives*; and for long database transaction will include read and write operations, Möbius provides *dynamic primitives*. By combining FTL persistence with transaction processing, Möbius can avoid extra FTL persistence writes in a larger time scale and enhance the performance.

By revising Linux ATA drivers and adding hundreds of lines code in upper layer software, we can support database transaction for MySQL and file system transactions which provide version consistency for the ext2 file system easily by using transactional primitives provided by Möbius. It can get a better transaction processing performance compared with other transactional SSDs or software solutions. And after a crash, we can roll back database or file systems to a clean state quickly.

Our main **contributions** can be summarized as follows:

- *Design and implementation of a quickly-recoverable transactional SSD.* We call our transactional SSD Möbius and implement it in OpenSSD platform as a prototype. Möbius puts transactional processing logic into FTL and stores FTL persistence and transaction information together in a certain area called *atom log area* (ALA) of SSD. After power failures, SSD can be quickly recovered by scanning transaction information stored in ALA using a *directed edge graph* (DAG) verification method. In addition, we revise the *page allocation*, *garbage collection* and *wear leveling* procedure of SSD to guarantee these mechanisms won't affect Möbius.
- *Rich transactional primitives.* To the best of our knowledge, this is the first work to provide both static and dynamic primitives for transactional SSD. Möbius can support both primitives with corresponding design and implementation.
- *Real system test and evaluation on our prototype.* We first compare Möbius with normal SSD to demonstrate performance overhead to support transactional primitives. Then, for end-to-end comparison, we compare Möbius with other transactional SSD designs. Experimental results show that Möbius can at most 67% outperform in transaction throughput (TPS) and 29 times outperform in recovery time comparing with other transactional SSDs.

The rest of this paper is organized as follows. Section II gives the background of transactional SSD architecture. In Section III, we propose the atom file abstract and give our design of Möbius. Then we describe implementation details of Möbius in OpenSSD platform in Section IV. We evaluate Möbius on our prototype by real system tests in Section V. Conclusions are provided in Section VI.

II. Background

A. SSD Primary

NAND flash-based SSD is composed by NAND flash chips and a chip is composed by several flash planes, while planes are composed by dozens of flash blocks. A block has 64 or more flash pages live in it and each page has a data area sized from 2KB to 16KB and OOB area sized 64Byte to 512Byte [20], [21]. Page is the minimum read and write unit in NAND flash and each page needs to be erased before write, while erase is an operation with high latency in block granularity.

To avoid expensive erase operation before every in-place update, modern SSDs write data in form of log and use a mapping table to map logical page number (lpn) to internal physical page number (ppn). This is known as "out-of-place" updating specialty of SSDs. And the logical to physical address mapping scheme within the device is known as the Flash Translation Layer (FTL). FTL plays the core role in SSD design and has direct impact on the performance and durability of the SSD device and significant effort [22], [9], [23], [24], [25], [26] has gone into optimizing the FTL for performance, space efficiency, durability, or a combination of these properties.

By mapping granularity, FTL techniques can be divided into block-level FTL, hybrid FTL and page-level FTL. Because of high cost block merge operation, block-level FTL and hybrid FTL are replaced by page-level FTL with big size DRAM chips are used in SSD.

DFTL [23] is a typical page-level FTL designed for limited RAM size which can selectively cache page-level address mapping entries in RAM. DFTL divides SSD flash into three parts: Data Blocks Area (DBA), Translation Blocks Area

(TBA) and Global Translation Directory (GTD). DBA is used to store user data, while TBA is used to store mapping table. A translation page in TBA stores sequential lpn to ppn mapping entries. GTD is used to index translation pages in TBA since translation pages are changed frequently. GTD size is small (4KB for 1GB flash), so it can be cached in RAM with hot translation pages. DFTL has better performance comparing with hybrid FTL in limited RAM environment because locality.

Except FTL, out-of-place updating specialty induces extra complexity in the hardware design of SSD. For example, Garbage Collection (GC) operation is periodically executed in SSD to erase and reclaim outdated physical pages and migrate useful pages in blocks. To realize the sophisticated internal mechanism in SSD, there are some other components except NAND flash chips. Like a small computer, SSDs usually have a ARM-based controller and on board RAM chips to cache mapping entries and buffer data. Some high-end SSDs have super capacitors as Sudden Power-Off Recovery (SPOR) facility.

B. Sudden Power-Off Recovery Mechanism in SSD

In this paper, we regard mapping table and other data stored in flash except user data as the metadata of SSD. We divide metadata of SSD into two categories: Global Metadata (GM) and Local Metadata (LM). GM means metadata describes whole-disk information and stored in separate pages like mapping table, free block list and pointers. While LM means metadata attached with user data in page OOB like lpn, ECC and version number.

GM is extremely important to SSD since it describes the SSD. But if we persist GM like mapping entry after each new page write, it will induce serious Write Amplification [27] problemi in SSD. So with the size of DRAM in SSD become bigger, more GM are cached in RAM. But when crash or power failure or happened, GM in RAM like mapping entries and pointers will be lost without persistence.

Because SSD internal techniques are confidential in commercial SSDs, there are only few academic work about SPOR mechanism in FTL after power failure [24], [25]. In addition, some patents are published on this topic [28], [29]. PORCE [24] is used in block level FTL SSD and mainly focus on consistency problem when SSD faces power failure in GC operation. It utilizes some separate blocks to store log information when GC begins and ends. By these logs, PORCE can recovery from power failure quickly. CR-FAST [25] is another work on SSD crash recovery based on a hybrid FTL named FAST [30]. It writes periodically newly generated address mapping information in a log structured way. So after power failures, mapping table is rebuilt by scanning pages which mapping information is still uncheckpointed. Rogers et al. presented a similar periodical checkpointing method in their patent [28]. Lee et al. gave a method by tracing updates with the same lpn using pointers stored in OOB to efficiently recover SSD from power failures [29].

C. Transactional SSDs Interface and Design

State-of-the-art transactional SSD interface appears a little messy. Different transactional SSD designs provide different interfaces. Actually, they can be divided into two categories by supporting different types of transaction. We define **static transaction** that all data manipulated in the transacton is determined before the transaction begins, e.g., all data are already in cache, and **dynamic transaction** that all data in this trasaction are not determined when it begins. Accordingly, **static interface** only supports the static transaction processing and **dynamic interface** can support the dynamic transaction processing. It is worth noting that dynamic interface can also support static transaction processing easily. Comparing with static interface, dynamic interface is more flexible. Static interface usually can only support write operation in transaction, and dyanmic interface could support read operation in transaction. But for dynamic interface, supporting read operation in transaction could be difficult in prior designs.

TxFlash [14] proposes two cyclic commit protocols named SCC and BPCC to realize its static AtomicWrite interface only for write operations. It links all pages in each transaction in one cyclic list by keeping pointers in the OOB area of pages. It judges whether the cyclic list is closed to determine the state of each transaction after power failures. Since SCC and BPCC are simple methods by adding a little pointers in OOB, there's no extra metadata in SSD GM. But the pointer-based protocols in TxFlash have two limitations. First, because there's no global metadata can be used, TxFlash is required to scan the whole SSD to find the unfinished transactions and abort them after power failures. Second, garbage collection should be carefully performed to avoid the ambiguity between the aborted transactions and the partially erased committed transactions, because neither of them has a cyclic list.

Atomic-Write is another transactional SSD interface proposed by Ouyang el al [16]. Their design is based on log-based FTL and it stores an extra one-bit commit flag with the value 0 or 1 to determine whether or not the last transaction was committed before crashes. Atomic-Write is lightweight and can support both static and dynamic transactions and it can even support read operations in a transaction. The cost of transaction can be ignored because it only adds an one-bit flag in FTL log. But this design also restricts that there is at most one running transaction in the SSD and this can be the bottleneck of whole system.

LightTx is a trasactional SSD design which supports dynamic interface for write operations [17]. It divides flash blocks into four zones: checkpointed zone, unavailable zone, available zone and free zone. When crash happens, LightTx only scan the unavailable and available zones to abort unfinished transactions. This will reduce the recovery time from power failures in a certain degree. But scanning and verifying overhead could be high if the zone becomes bigger. Besides mapping entries are persisted to flash once the transaction is committed. In page level FTL, it will incur dozens of page write for a large transaction with hundreds of pages and make the Write Amplification problem severe. Moreover, LightTx claims that it provides flexible isolation levels for transaction processing. Nonetheless, as we will discuss in Section III-G, this kind of flexibility could induce performance even correctness problems.

D. Summary

In this paper, we propose a novel transactional SSD architecture named Möbius. Möbius is motivated by providing

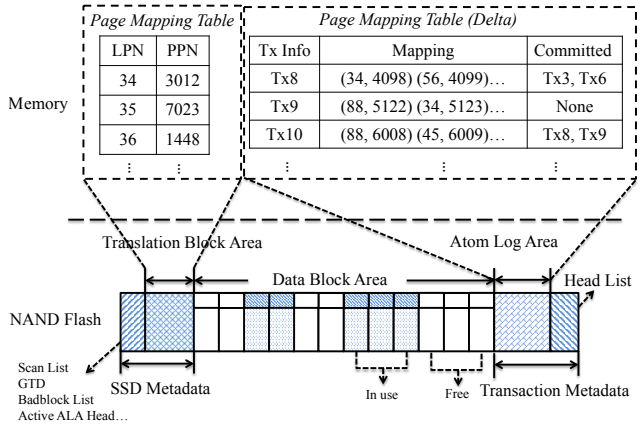


Fig. 2: Architecture of Möbius.

FTL SPOR mechanism combining with transaction processing in SSD. Möbius supports both static and dynamic transaction interface with corresponding implementation and it can support read operation in dynamic transaction. Besides, by combining FTL persistence with transaction processing, Möbius can avoid extra FTL persistence write in a larger time scale. After power failure, Möbius can avoid large scale scanning and recover SSD immediately to a right state.

To differentiate between existing transactional SSD designs clearly, Figure 1 explains how transactions grow in different transactional SSDs. In Atomic-Write, transactions are grew in one-dimension and at any time there will be at most one running transactions. Trsactions grow in LightTx is in two-dimension, this will make state tracking difficult. In MU, transactions are growing in two-dimension, and we use Atom Log Area (ALA) to track transaction information and recover from power failure quickly.

III. Design of Möbius

In this section, we first outline the architecture of Möbius and then depict its major components in detail.

A. Overview

Möbius is designed as a subsystem in SSD that can store transaction in the form of a special type of file, atom file, which has exact one-page-size metadata. Transaction information and mapping information for pages in the transaction and other global metadata of SSD are written to a page as atom inode. Figure 2 gives an overview of Möbius and unveils its major differences from other transactional SSDs:

- First, Möbius supports both static and dynamic transactions and provides two different interfaces for the system. For different interfaces, Möbius uses different mechanism to realize it. In our design, it is possible that several static transactions running at the same time without conflict. But for dynamic transactions, we cannot be whether or not there is proactively. And to avoid potential confliction, we take a conservative method. There is at most one dynamic transaction to run in Möbius simultaneously.
- Second, Möbius combines SSD SPOR mechanism in its design. Instead of just persisting newly generated mapping information, i.e. remap table, to flash periodically, Möbius regards remap table as delta of mapping table and stores it in the one-page-size atom metadata. And after each transaction is committed, there is no need for FTL to persist remap table explicitly. New mapping table is built by merging the original mapping table and deltas in the startup procedure.
- Third, Möbius can provide concurrent processing for static transactions. And Möbius supports read operation in dynamic transactions. In prior approaches, write is the only operation considered in transactional SSDs because most of them focus on write-back cache. For write-through cache, computation may also rely on data read from persistent layer. Thus, read is an important operation in dynamic transactions.

Details of the interface, internal mechanisms and commit protocol of Möbius are given in the following subsections.

B. The Interface Matters

Operations	Description
WRITE(*p*)	normal write
READ(*p*)	normal read
SWRITE(*uuid,* $p_1, \ldots, p_n$)	static transactional write
SREAD(*p*)	serializable read
DWRITE(*p, flag*)	dynamic transactional write
ABORT(*uuid*)	abort the transaction

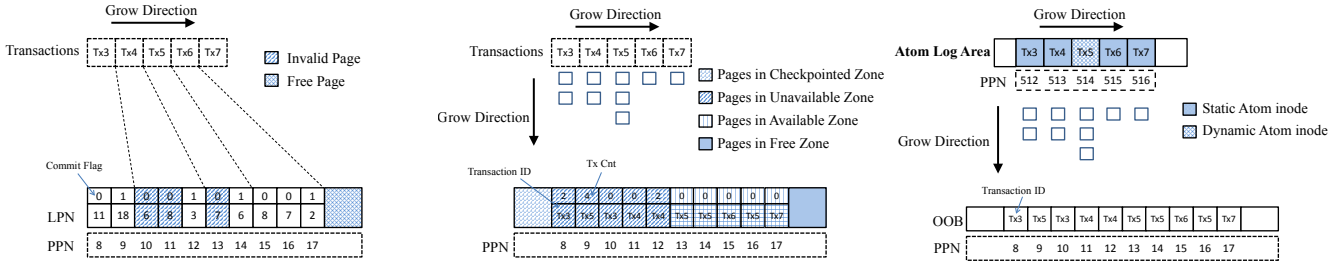
TABLE I: Host Interface.

In order to support the transaction primitives, we revise the device interface of SSDs by two principles: a) compatible with current interface; b) minimize interactions between system and device.

As Table I shows the interface used in Möbius. *WRITE* and *READ* are inherited from the current interface which, means writing a page and reading a page.

SWRITE is a new command added to support static transactions. The *uuid* parameter is generated by the system and can be guaranteed as a universal unique identifier. The $p_1, \ldots, p_m$ parameter denote the pages in memory to be written and here *m* is greater than 1. Although there's no read operation in static transactions, there may be simultaneous read operations outside the static transaction. So different isolation level reads are provided. Normal *READ* can guarantee it won't return the data in the middle of a transaction, i.e. Read Committed isolation level. While *SREAD* can provide the strictest Serializable isolation level, which means it can guarantee all write operations arrive before it are persistent.

DWRITE is the interface to support dynamic transactions. The *flag* parameter can tell the device which type of page to get written. There are 3 types of pages: *HEAD*, *BODY* and *REAR*. *HEAD* means *p* is the first page of a transaction, and *REAR* means *p* is the last page of a transaction. For the rest cases, value of *flag* is *BODY*. *ABORT* is a new command used to abort a transaction with a given id, when the id equals to 0 it means



(a) One-dimensional transaction growing in Atomic- (b) Two-dimensional transaction growing in LightTx (c) Two-dimensional transaction growing in Möbius Write

Fig. 1: Transaction Processing Procedure in Different Transactional SSDs.

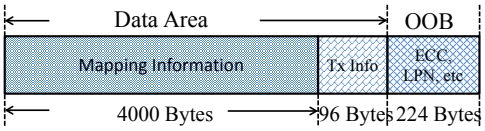


Fig. 3: 4KB *atom* inode.

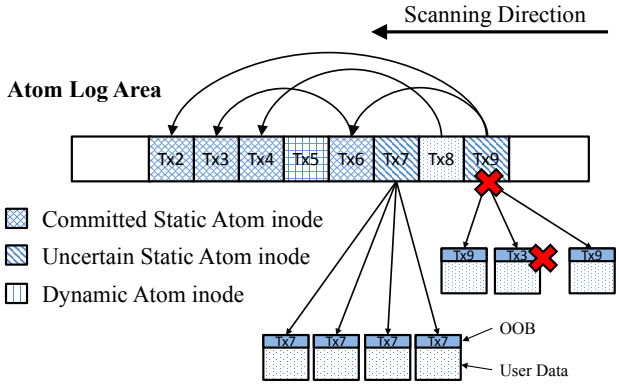


Fig. 4: DAG Verification Method.

to abort dynamic transaction since there is at most one running dynamic transaction in Möbius. In dynamic transaction, it can use normal *READ* to support inner-transaction read. Since there is at most one dynamic transaction running in Möbius at a moment, we don't consider isolation for dynamic transactions.

C. Static Transaction Processing

Atom File: We firstly introduce the abstract in Möbius, which is used throughout this paper: *atom file*, which is also abbreviated to *atom*. We define *atom* is *a type of special file with exact one-page-size inode*. In Section II-A, we mentioned that SSD can guarantee atomic write operation. In our design, we combine mapping information persistence with transaction processing together by storing transaction information with mapping information in the one-page-size *atom* inode.

For static transaction, SSD can get all the addresses of logical pages changed in the transaction before it begins. After SSD assigns all physical pages for the transaction, Möbius writes mapping information and transaction information as *atom* inode to certain area named Atom Log Area (ALA) in SSD continuously before starting the transaction. We will introduce how we implement ALA in Section IV-C. The size of *atom* inode always equals to the physical page size of the SSD. Figure 3 gives an example of 4 KB size *atom* inode. Since we mentioned above, a 4 KB physical page usually has a 128 Byte OOB. An address mapping item with a 32 bit logical page address to a 32 bit physical page address will occupy 8 Byte. In our design, there will be at most 500 mapping items in the data area of a page. And the left 96+128 Byte space will be occupied by transaction information and other local metadata for SSD like ECC.

After writing *atom* inode to the certain area in SSD, Möbius will write the data to the assigned physical addresses. Unlike normal SSD write, a 4 byte transaction id will be written in the OOB with every page write. For every data write in static transaction, mapping information will not be changed in RAM immediately after the data are written to SSD. They will be changed together in RAM after the transaction is committed. So read request will not read any data that belongs to an uncommitted transaction. But there's no extra procedure to persist the dirty mapping information to SSD because they are actually persisted to SSD before the transaction started within the *atom* inode.

When all the pages in a transaction are written to SSD, the status of this transaction, i.e. commit flag, will be changed in RAM. Commit flag of a transaction is usually stored within the transaction information. But in our design, the commit flag will not be written to the same transaction while it will be written to the coming-in transactions. So there's no extra write for the commit flag. And one *atom* inode can store as many as 8 commit flag for other transactions. Since *atom* inode that denotes the transaction is stored one by one in ALA, if we consider the commit flag as a directed edge from the source transaction to the committed transactions, there will be a directed edge graph (DAG) in ALA, Möbius use this DAG to quickly recover from power failures.

DAG Commit Protocol: If a transaction is committed, the commit flag of this transaction will be written within the *atom* inode of a following transaction.

When power failure happens, *atom* inodes will to check transaction status by backward scanning. If an *atom* inode is pointed by another *atom* inode, it means that the transaction

denoted by the first *atom* inode is committed. For those *atom* inodes don't pointed by some other *atom* inodes, since mapping information can be easily get, a read-and-verify procedure will be carried out to check the transaction status. For every physical pages in a transaction, Möbius will check whether they are belonging to this transaction by read pointers stored in OOB. As Figure 4 shows that, Möbius firstly scans the ALA to get the DAG, then further checking will be carried out on those *atom* inodes without any pointer pointing to them. To minimize the scanning cost, Möbius set a threshold value S_{max}. S_{max} means commit flag of a transaction can be stored in a *atom* inode after at most S_{max} transactions.

D. Dynamic Transaction Processing with Normal READ and WRITE

Since dynamic transactions are unpredictable, when we start a dynamic transaction we don't know the coming request of this transaction. Furthermore, as we will introduce in Section III-G, dynamic transactions will have contention problem in concurrent processing. So Möbius is designed to process dynamic transactions with strict serializable order. We regard dynamic transaction as normal write in our design. But when Möbius is processing dynamic transaction, SSD cannot accept other write request like SWRITE or normal WRITE requests. Whenever DWRITE or normal WRITE begins, the first page will be written to Head List in a determined area in SSD.

Unlike static transactions, Möbius cannot write *atom* inode before dynamic transactions committed. Möbius use a link-based commit protocol like SCC/BPCC [14] to solve dynamic trasaction processing. In normal SSD, there will be a free block list to be writing. In Möbius, except the free block list, there will be another Head List to update the first page of a dynamic transaction or a new begin normal WRITE. When Möbius accept a dynamic transaction request, it will write the first page to Head List with a pointer stored in its OOB to point the next physical page for writing. Then the following pages also store a pointer in their OOB to point the next page. When the transaction is committed, the pointer in last page will point to its head. When the dynamic transaction is committed, *atom* inode with mapping information and transaction information is written to ALA and marked as dynamic transaction. For normal WRITE, after the mapping area of an atom inode is full, the atom inode is written to flash and Möbius will accept a new around WRITE requests. Normal WRITE cannot be processed concurrently with any type of transaction interface in our design.

Read operation is supported in dynamic transaction processing. Unlike static transactions, when each page in dynamic transaction is written to SSD, the mapping information in RAM is changed. So read operation can be easily supported whether inner-transaction or inter-transaction read.

When a crash happens, since there's at most one dynamic transaction running in Möbius, we only check the last transaction in the Head List. Although read the linked list cannot be accelerated by internal parallelism of SSD, it is still acceptable because there's only one linked list to be verified.

E. Other Interface Support

SREAD: SREAD is an interface to provide serializable isolation for static transactions. If a SREAD request is coming, it will return data after all the static transactions are committed. Möbius realizes it by adding a flag in RAM to wait for static transactions before it committed. By providing SREAD, Möbius can prodivd different isolation level read operations for static transactions. But for dynamic transactions, since any time there will be at most one running dynamic transaction in Möbius, it can provide strictest serializable isolation level for READ and SREAD is not supported in dynamic transactions.

ABORT: Abort is an interface used to stop a transaction. For static transactions, when the upper layer system wants to abort a running transaction, Möbius just don't store the commit flag in the following transactions, and it will be garbage collected if the threshold S_{max} is violated. For dynamic transactions, since *atom* inode is written after the data is written, Möbius will write *atom* inode to mark this transaction as aborted.

F. mFTL

To provide a logical-to-physical mapping in Möbius, mFTL is introduced as a page-level FTL based on DFTL [23]. As we outlined in Section II-A, DFTL can selectively cache page-level address mappings in RAM. In functionality, mFTL is the same with DFTL, but the persistence logic of translation pages in mFTL is different from DFTL.

Since DFTL is a fine-grained page level FTL, it avoids merge operations in hybrid FTL and gains performance benefit. Nonetheless, "fine-grained" means more mapping information. And more mapping information means more metadata persistence. In DFTL, dirty translation pages is written to flash only when the mapping item in this page is evicted from RAM and GTD pages in RAM is written to SSD periodically. Whole SSD scanning is needed to avoid data lost, otherwise DFTL can only promise data persistence under GTD protection. Since the gap between data write and translation page and GTD write, DFTL cannot simply be used in Möbius.

In mFTL, since mapping information is written to flash in *atom* inodes when transactions are committed, there's no need to write to translation pages again. But this incurs a problem: How Möbius read the mapping information stored in *atom* inodes? In mFTL, it splits Cached Mapping Table (CMT) into Evicting Area and Resident Area. Evicting Area is like CMT used in DFTL, but Resident Area is used for mapping information stored in *atom* inodes and cannot be evicted out of RAM. Under extreme condition, every *atom* inode stores maximum number of mapping items, the size of *atom* inode equals to Resident Area. Evicting Area is used from low to high address while Resident Area is used from high to low address. And Resident Area will be set a threshold $c\%$. In Jasmine OpenSSD platform, since the DRAM is 64 MB, we set c=20. This threshold $c\%$ is close relevant to the threshold $Size_{Active}$ which denotes active area size of ALA because $Size_{Active}$ should promise size of Resident Area does not violate the $c\%$ threshold under extreme condition.

When SSD is booting, the first step is aborting unfinished transactions, after that, Möbius will read mapping information stored in the active area of ALA to Resident Area. When the size of the active area of ALA is greater than $Size_{Active}$, mapping information stored in the active area of ALA will be merged into translation pages and the head pointer of ALA will be updated.

coreutils/lib/full-write.c

```
41   size_t
42   full_write (int desc, const char *ptr,
size_t len)
43   {
44     ssize_t total_written = 0;
45
46     while (len > 0)
47       {
48         ssize_t written = write (desc,
ptr, len);
:            :::
:            :::
:            :::
61         total_written += written;
62         ptr += written;
63         len -= written;
64       }
65     return total_written;
66   }
```

coreutils/src/copy.c

```
317   buf = (char *) alloca (buf_size +
sizeof (int));
318
319   for (;;)
320     {
321       ssize_t n_read = read
(source_desc, buf,
            buf_size);
:            :::
:            :::
:            :::
372       if (ip == 0)
373         {
374           size_t n = n_read;
375           if (full_write (dest_desc,
buf, n) != n)
376             {
377               error (0, errno, _("writing
%s"),
                      quote (dst_path));
378               return_val = -1;
379               goto
close_src_and_dst_desc;
380             }
381           last_write_made_hole = 0;
382         }
383     }
```

Fig. 5: Code segment of cp in GNU coreutils.

G. Discussion on Concurrent Transaction Processing

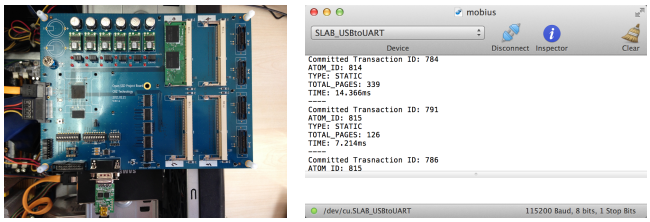
Contention between Transactions: *A)* For dynamic transactions, it makes concurrent transaction processing more difficult because of their unpredictable property. Figure 5 gives a segment of code in GNU *cp* program as a simple example to address the thrashing problem in concurrent processing. As the code shows us, *cp* is a simple program to read data from one file into a fixed size buffer and then write the data in repeat buffer to another file. By repeating this operation, *cp* copys one file to somewhere. Let's think about running *cp* in a system with write-through page cache. As the system prefetch function works, *write* calls will send frequently. In a *write* system call, data will be written and then metadata will be changed. If we think one system call as a transaction (actually transactional file systems do that), then reading and modifying logical pages contain metadata of the file will be a preemption point. If such contention happens, only one transaction will be reserved while others be aborted. Such a procedure will invalidate a lot of pages and become a performance problem. *B)* For static transactions, contention brings correctness problem instead of a performance problem because right order to update contention pages can enhance concurrency in a certain degree. Let's think about two transactional writes to the same file again. This time, these writes are issued by journal commit process in a system which uses wirte-back page cache. Without consideration of data page contention, the metadata page will be a preemption point again. Tx(A) if the first static transaction including logical page A, B and C. C is the metadata page which changes the size of the file from 4KB to 8KB. Tx(B) is the second static transaction including logical page D, E and C. C is the metadata page which changes the size of the file from 8KB to 12KB. Tx(A) is in priority order against Tx(B) in write order. After Tx(A) and Tx(B) committed, size of the file will be 12KB. If unfortunately, Tx(B) is committed earlier than Tx(A), then there will be correctness problem and the file size will be wrong.

Inter- and Inner- Transaction Read Problem: In a transactional SSD designed to process dynamic transaction concurrently, if mapping item is updated in RAM after each page write, then inter-transaction read operation may get data in unfinished transactions; if if mapping items are updated in RAM after transaction is committed, then inner-transaction read may get the outdated data. As we mentioned above, LightTx is a transactional SSD design to process dynamic transaction concurrently. But the author didn't point whether or not LightTx will support read operation.

Because of these two problems in concurrent dynamic transaction processing, Möbius chooses a conservative way to process dynamic transactions, i.e. sequentially processes dynamic transaction and there's no concurrency between dynamic transactions.

IV. Implementation

A. Platform



(a) OpenSSD Platform (b) Serial Interface Debugging

Fig. 6: Implementation Platform for Möbius Architecture.

As Figure 6 shows that, we implement Möbius architecture in Jasmine OpenSSD platform [19], which is a reference implementation of SSD based on the Indilinx controller. The firmware of Jasmine OpenSSD can be revised and it provides a serial prot for debugging. To support transactional interface in OpenSSD, we add several SATA command to extend the existing interface of OpenSSD. To support transactional interface in upper layer system, we also revise libATA to support these command for Linux. In our current implementation, it only works for SATA interface SSDs. Table II gives the brief hardware specification of Jasmine OpenSSD platform. There are two types of NAND flash chips used in our implementa-

Hardware Parameter	Value
CPU	ARM7TDMI-S up to 87.5MHz
SRAM	96KB
DRAM	64MB
Interface	SATA 2.0
Capacity	64GB (open-ended)
NAND Flash 1 Page Size	4KB + 224Byte
NAND Flash 2 Page Size	8KB + 448Byte
ECC	24b/1KB

TABLE II: Brief hardware specification of Jasmine OpenSSD.

tion[2].

B. Sync-SWRITE vs. ASync-SWRITE

We implement two modes of SWRITE in our prototype. Sync-SWRITE returns when all pages in transaction are written to flash, but ASync-SWRITE returns once the atom inode is written to flash.

ASync-SWRITE can also provide transaction guarantee, but as we will mention in Section IV-E, ASync-SWRITE may have "false positive" problem because system thought the transaction is committed but it could be aborted as unfinished transaction when recovery from power failure.

C. Internal Structures and Mechanisms Implementation

Atom Log Area (ALA) is used to store inodes of *atom* files. ALA is an area with sequential physical addresses. ALA is used to quickly scanning at the boottime because continuous physical addresses can enhance internal parallelism of SSD and shorten the scanning time. The most important characteristic of ALA is that the left boundary of active ALA (Active ALA Head) is assured and stored as Global Metadata in SSD, but the right boundary of it is determined by scanning. When the scanning meets a blank page, it is the right boundary.

As we mentioned in Section III-F, the size of active ALA cannot exceed *c%* of RAM size. In our implementation, we set size of active ALA 12MB, and it can store at most 12MB/4KB=3072 *atom* inodes. We set the total size of ALA 500MB in our implementation. When the size of active ALA exceeds 12MB, mapping entries in Resident Area will be written to flash. This operation won't block FTL read and write. After all mapping entries are persistent in flash, left boundary of active ALA is changed and written to flash.

In static transaction processing, physical addresses are allocated before transaction starts. We implement a simple addresss allocation scheme in our prototype which allocates physical pages sequentially.

D. Garbage Collection

Except ALA, garbage collection scheme is the same as DFTL. Since data page is invalid only when the mapping entry is merged into translation page, the garbage collection process appears to be delayed. Since ALA is a new internal structure in SSD, mFTL gives the garbage collection procedure in ALA. When the ALA is merged with translation pages, these blocks are outdated and there's no useful information needed to be collected. So after left boundary of active ALA is changed, pages live in the left of the boundary will be erased directly when SSD is idle.

[2]NAND flash 1 uses 34nm Micron MT29F32G08CBABA flash chips and NAND flash 2 uses 35nm Samsung K9LCG08U1M flash chips. We mainly test on 4KB page size flash, and only use 8KB page size flash to compare Write Amplification ratio.

E. Limitations

Static transaction size limitation is one of the limitations in Möbius architecture. For large transactions whose mapping information bigger than one page size, Möbius cannot directly support it. In our implementation, we divide the transaction into several small transactions but this actually breaks the transactional semanteme. For a flash chip with 4KB page size, it can support at most 500 pages in a trasacntion, which is 2000KB in size; while for a 8KB page size flash chip, it can support at most 1000 pages in a transaction, which is 8000KB in size. For super large transaction, it can be supported by regarding its atom inodes update as a transaction and use another atom file update to abstract the transaction. This is just an iteration process.

Besides, Write Amplification problem seems to be another limitation when Möbius is used to process small transactions since for every transaction, Möbius will wirte an extra page as atom inode. In our implementation, for small static transactions with less than 10 pages size, Möbius will merge them as a big transaction to alleviate the extra *atom* inode write cost.

"False positive" problem of ASync-SWRITE interface is another limitation in Möbius. Under ASync-SWRITE, Möbius will return after atom inode is written to flash. At this time, the left of the transaction is still in RAM. Nonetheless, ASync-SWRITE can still provide the transactional guarantee.

Prior transactional SSD designs use various techniques to avoid commit log for transactions [14], [16], [17]. But in Möbius, atom inode is not simply a commit log. It contains both transaction information and FTL mapping entries to avoid extra mapping persistence. We will show experiment results in Section V, Möbius doesn't make Write Amplification problem more severe. On the contrary, Möbius has a similar performance with others in most cases. In some cases, Möbius even has better performance because it avoids mapping persistence in a large time scale.

V. Evaluation

We evaluate our prototype of Möbius by experimental tests under different baselines. All tests are performed on a real machine for which the specification is shown in Table III. For comparing requirements, we also implement raw DFTL [23], Atomic-Write [16] and LightTx [17] on OpenSSD. We implement Möbius with 20MB active ALA size.

In the following, we firstly compare Möbius with raw DFTL SSD using micro-benchmarks to demonstrate potential performance overhead of transaction processing. Then we run macro-benchmarks on top of real transactional SSDs to compare Möbius against Atomic-Write and LightTx in terms of performance, write amplification ratio and recovery time.

Processor	Xeon X3210 @ 2.13GHz
DRAM	8GB DDR3 1333MHz 2x4GB DIMMs
Boot Device	256GB Samsung SSD
Storage Device	Möbius SSD
Operating System	Ubuntu 10.04 Linux Kernel 2.6.32

TABLE III: Experimental Machine Configuration.

Since hardware transaction processing outperforms software solutions has been proved in prior work [14], [16], be restricted by content length, we don't give the results of Möbius comparing with software transaction processing solutions in our paper.

A. Möbius vs. Raw DFTL SSD

Since Möbius is based on DFTL and it doesn't change the read procedure in DFTL, so we mainly focus on write operation in this comparison. Actually, there are three types of write operation in Möbius: SWRITE, DWRITE and WRITE. As we described in Section III-D, DWRITE and WRITE essentially have the same internal mechanism. So in the following comparison tests, we only compare SWRITE and DWRITE in Möbius with DFTL. Besides, as we mentioned in Section IV-B, SWRITE has two modes: Sync-SWRITE (S-SWRITE) and ASync-SWRITE(A-SWRITE).

We compare DFTL with Möbius by performing a compound write which consists of 32, 64, 128 and 256 pages to storage (averaged over 50 iterations). Since transaction size is not measured in this test, we only running tests on NAND Flash 1 with the 4KB page size. To avoid the noise of environment, we disable the Linux buffer cache for raw device in our tests to evaluate the overhead of Möbius. Since we don't revise page cache for file systems, it will not affect system performance obviously. Besides, single thread I/O process is running sequentially, it cannot measure the performance of raw device accurately. So we also utilize the Linux native asynchronous I/O library, libaio, to submit all pages via one I/O request, wait for the operation to complete.

For SWRITE interface, similar with experiments in Atomic-Write [15], we encapsulate all pages in a single transaction, issue the request to Möbius, then wait for its completion. For DWRITE interface, we send the WRITE request one by one as a dynamic transaction. Both SWRITE and DWRITE will not buffer data. For Random workload, pages are randomly scattered within a 5 GB range and aligned to 4KB boundaries. For sequential workload, logical pages be written are in sequential order.

Write Latency: Figure 7 shows the average latency to complete these writes with one single process or libaio under random or sequential workloads. Under random workloads, raw DFTL SSD outperforms normal single I/O process using libaio because single process cannot utilize SSD sufficiently. DFTL is able to slightly outperform DWRITE in Möbius both using single process and using libaio because there is no transaction processing logic in DFTL. But both Sync-SWRITE and ASync-SWRITE is better than AIO DFTL because their are extra system calls when we use libaio in raw DFTL SSD. ASync-SWRITE is best among these interfaces because it returns when the atom inode is persisted in flash. Although transactions use ASync-SWRITE may have "false positive" problem, but it still can provide transactional guarantees. Under sequential workloads, both DFTL and Möbius have better performance because locality advantage in DFTL. And DFTL SSD use libaio is better than its random workload result because of I/O consolidatation functionality in libaio.

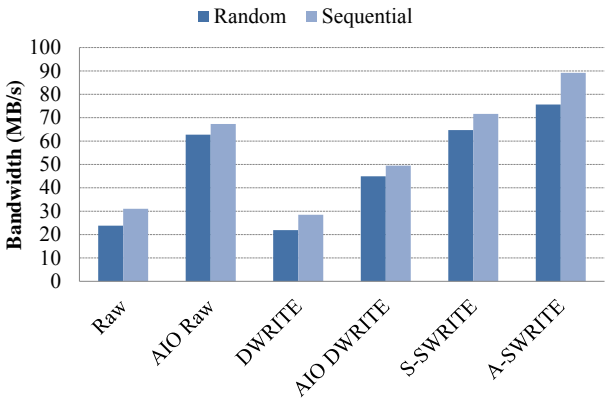


Fig. 8: Bandwidth Comparison Over Raw DFTL and Möbius on OpenSSD.

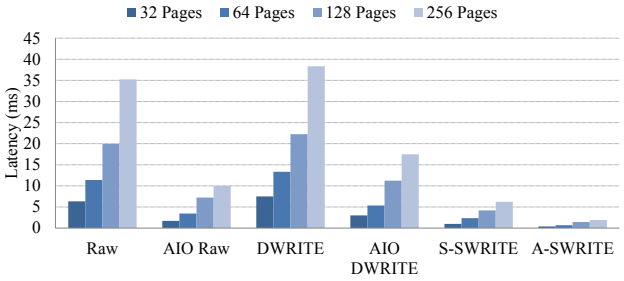
Write Bandwidth: Figure 8 shows the average bandwidth with one single process or libaio under random or sequential workloads in DFTL SSD and Möbius. Since number of pages in a transaction has little impact bandwidth, we only give the bandwidth of 256 pages compound write. Because OpenSSD is an experimental platform, so the extreme bandwidth is limited in about 95MB/s. In Async mode SWRITE, it can approach the extreme write bandwidth of our platform.

B. Möbius vs. Other Transactional SSD Designs

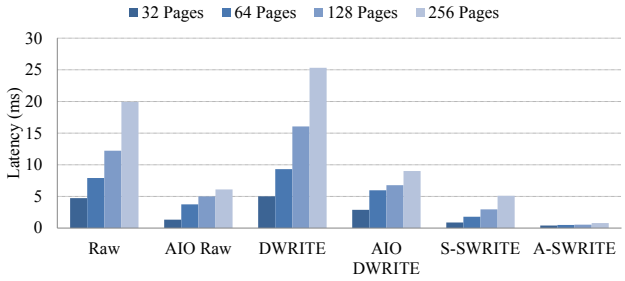
To compare Möbius with other transactional SSD designs by macro-benchmark, we evaluate an industry standard transaction processing workloads DBT-2 [31] which is implementation of TPC-C [32] and fileserver and webserver benchmark in Filebench [33]. For these three types of micro-benchmark, DBT-2 is used on revised MySQL database and filebench is running on a revised jbd layer for ext3. We first run these benchmarks on different transactional SSD to get the bandwidth and then get the TPS (Transaction per Second) metric as the performance index for each transactional SSD.

Figure 9 shows that the performance under macro-benchmark in terms of bandwidth and Transaction per Second (TPS). Transaction throughput is normalized by the result of Atomic-Write. We can find that, Möbius can at most 67% outperform Atomic-Write using A-SWRITE in transaction throughput result. S-SWRITE is a more strict form of transactional interface and it can provide similar throughput comparing with Atomic-Write. In filebench tests, Atomic-Write outperforms S-SWRITE because transactions are small in these scenarios.

Figure 10 shows the Write Amplification ratio comparison between Möbius, raw DFTL Atomic-Write and LightTx. Result shows that Möbius do not enlarge Write Amplification

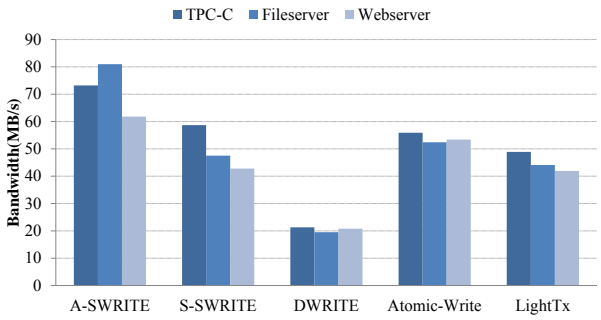


(a) Random write latency

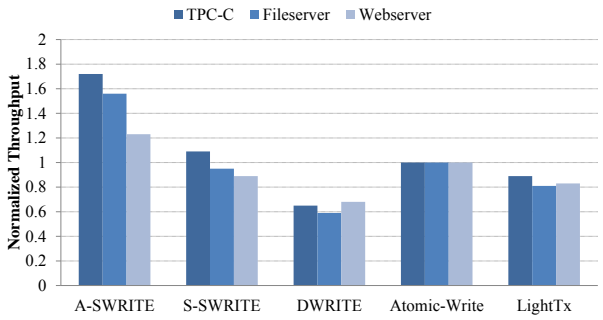


(b) Sequential write latency

Fig. 7: Latency Comparison Over Raw DFTL and Möbius on OpenSSD.



(a) Bandwidth



(b) Transaction per Second

Fig. 9: Performance Comparison Between Different Transactional Interfaces.

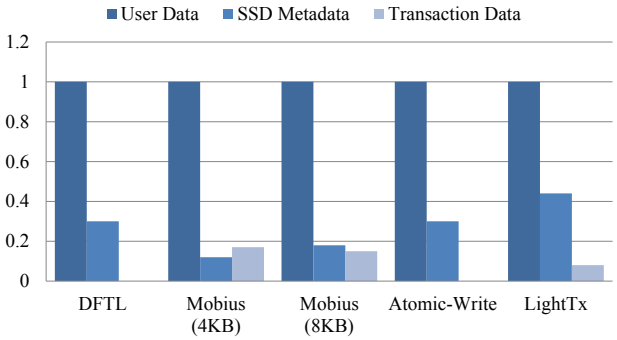


Fig. 10: Write Amplification in Raw DFTL and Other Transactional SSDs on OpenSSD.

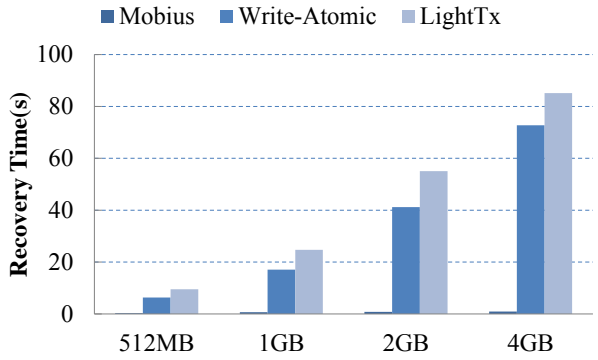


Fig. 11: Recovery Time of Different Transactional SSDs.

ratio comparing with other methods although it adds one page size atom inode for each transaction. It is because atom file also absorbs some mapping writes. Moreover, it is even better than LightTx. LightTx is a sliding zone based method, and the boundaries of zones will be written frequently once the transaction states are changed. As for Atomic-Write, it will write mapping information once a transaction is committed. This will introduce a serious number of extra mapping writes.

Figure 11 shows recovery time comparison between Möbius and other transactional SSDs under different scanning zone size. ALA is the scanning zone in Möbius while Available Zone and Unavailable Zone is the scanning zones in LightTx. In Atomic-Write implementation, its scanning zone size is fixed and small enough for quickly scanning. But in LightTx and Atomic-Write, it Sudden Power-Off Recovery (SPOR) procedure is operated by FTL. Specifically, DFTL needs to scan large part of the SSD. Result shows that Möbius 29 times outperform LightTx in recovery time, when the scanning zone size is 4GB. 4GB is a common size for scanning after power failure because common SSDs running I/O intensive workloads could easily get 4GB in nearly 50 seconds. Möbius combines SPOR and transaction logic together to get the shortest recovery time.

VI. Conclusion

Transaction processing is an important property required by modern storage systems. Since NAND flash based SSDs are extensively employed as storage devices, design of transactional SSD becomes challenging for software implementation due to "out-of-place" issue of SSDs. Thus, architecture level solutions become more attractive. With the help of our Möbius architecture, transaction processing is enabled for different interfaces. More important, the FTL consistency issue, which is neglected in previous approaches, is also addressed. In addition, the experimental results show that Möbius can outperform state-of-art approaches.

Acknowledgment

The authors acknowledge the support of the Nature Science Foundation of China under Grant No. 61373025, 61303002, the National 863 High-Tech Programs of China under Grant No.2012AA010905, 2012AA012609 and TNList cross-discipline foundation.

References

[1] Gray, Jim and Reuter, Andreas, "Transaction Processing: Concepts and Techniques." Morgan Kaufmann, 1992.

[2] M. I. Seltzer and M. Stonebraker, "Transaction support in read optimizied and write optimized file systems," in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 174–185.

[3] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "Extending acid semantics to the file system," *Trans. Storage*, vol. 3, no. 2, Jun. 2007.

[4] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300.

[5] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 102–.

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60.

[7] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88.

[8] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, no. 1, pp. 3:1–3:32, Jan. 2014.

[9] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, Apr. 2007.

[10] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 279–289.

[11] "Market share analysis: Ssd component and ssd-based appliances, worldwide, 2012." [Online]. Available: https://www.gartner.com/doc/2527217

[12] "Flash drives replace disks at amazon, facebook, dropbox." [Online]. Available: http://www.wired.com/wiredenterprise/2012/06/flash-data-centers/

[13] E. Gal and S. Toledo, "A transactional flash file system for microcontrollers," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 7–7.

[14] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 147–160.

[15] "Fusionio iomemory vsl driver." [Online]. Available: https://www.fusionio.com/load/-media-/1ufyu2/docsPress/press_release_optimus_prime.pdf

[16] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block i/o: Rethinking traditional storage primitives." in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 301–311.

[17] Lu, Youyou and Shu, Jiwu and Guo, Jia and Li, Shuai and Mutlu, Onur, "LightTx: A Lightweight Transactional Design in Flash-based SSDs to Support Flexible Transactions." in *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD '13)*, Asheville, NC, Oct 2013.

[18] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. 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*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 197–212.

[19] "Jasmine openssd platform." [Online]. Available: http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform

[20] Peter Desnoyers, "What Systems Researchers Need to Know about NAND Flash." in *the 5th USENIX Workshop on Hot Topics in File and Storage Technologies (HotStorage '13)*, San Jose, California, June 2013.

[21] Y. Kim, B. Taurus, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," Sep 2009.

[22] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. on Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.

[23] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 229–240.

[24] T.-S. Chung, M. Lee, Y. Ryu, and K. Lee, "Porce: An efficient power off recovery scheme for flash memory," *J. Syst. Archit.*, vol. 54, no. 10, pp. 935–943, Oct. 2008.

[25] S. Moon, S.-P. Lim, D.-J. Park, and S.-W. Lee, "Crash recovery in fast ftl," in *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, ser. SEUS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 13–22.

[26] A. Rogers, A. Kansal, and S. Patel, "Rapid crash recovery for flash storage," *US Patent. US7818610 B2*, pp. 1–23, Oct. 2010.

[27] "Write amplification." [Online]. Available: http://en.wikipedia.org/wiki/Write_amplification

[28] A. M. Rogers, A. Kansal, and S. C. Patel, "Rapid crash recovery for flash storage." Oct 2010, uS Patent 7,818,610.

[29] C.-W. Lee, S.-H. Chen, S.-t. Hung, P.-S. Chen, and P.-C. Lu, "Method for data recovery for flash devices." May 2013, uS Patent App. 12/784,593.

[30] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Songe, "Fast: A log-buffer based ftl scheme with fully associative sector translation," *The UKC, August*, 2005.

[31] "Osdl: Database test suite." [Online]. Available: http://osdldbt.sourceforge.net/

[32] "Tpc-c: an on-line transaction processing benchmark." [Online]. Available: http://www.tpc.org/tpcc

[33] "Filebench." [Online]. Available: http://sourceforge.net/projects/filebench/