Reliability Analysis of SSDs Under Power Fault

MAI ZHENG, New Mexico State University, The Ohio State University, HP Labs
JOSEPH TUCEK, Amazon Inc, HP Labs
FENG QIN, The Ohio State University
MARK LILLIBRIDGE, BILL W. ZHAO, and ELIZABETH S. YANG, HP Labs

Modern storage technology (solid-state disks (SSDs), NoSQL databases, commoditized RAID hardware, etc.) brings new reliability challenges to the already-complicated storage stack. Among other things, the behavior of these new components during power faults—which happen relatively frequently in data centers—is an important yet mostly ignored issue in this dependability-critical area. Understanding how new storage components behave under power fault is the first step towards designing new robust storage systems.

In this article, we propose a new methodology to expose reliability issues in block devices under power faults. Our framework includes specially designed hardware to inject power faults directly to devices, workloads to stress storage components, and techniques to detect various types of failures. Applying our testing framework, we test 17 commodity SSDs from six different vendors using more than three thousand fault injection cycles in total. Our experimental results reveal that 14 of the 17 tested SSD devices exhibit surprising failure behaviors under power faults, including bit corruption, shorn writes, unserializable writes, metadata corruption, and total device failure.

Categories and Subject Descriptors: B.8.1 [**Reliability, Testing, and Fault-Tolerance**]

General Terms: Design, Algorithms, Reliability

Additional Key Words and Phrases: Storage systems, flash memory, SSD, power failure, fault injection

ACM Reference Format:
Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. 2016. Reliability analysis of SSDs under power fault. ACM Trans. Comput. Syst. 34, 4, Article 10 (October 2016), 28 pages.
DOI: http://dx.doi.org/10.1145/2992782

1. INTRODUCTION

Compared with traditional hard disk, flash-based (solid-state disks (SSDs) offer much greater performance and lower power draw. Hence SSDs are already displacing hard disk in many datacenters [Metz 2012]. However, while we have over 50 years of collected wisdom working with hard disk, SSDs are relatively new [Bez et al. 2003] and not nearly as well understood. Specifically, the behavior of flash memory in adverse conditions has only been studied at a component level [Tseng et al. 2011]; given the

This work was partially supported by NSF Grants No. CCF-0953759 (CAREER Award), No. CCF-1218358, No. CCF-1319705, and No. CNS-1566554; by the CAS/SAFEA International Partnership Program for Creative Research Teams; and by a gift from HP.
Authors' addresses: M. Zheng, New Mexico State University, 1290 Frenger Mall, Las Cruces, NM 88003; email: zheng@nmsu.edu; J. Tucek, Amazon Web Services, 1900 University Cir, East Palo Alto, CA 94303; email: tucekj@amazon.com; F. Qin, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; email: qin.34@osu.edu; M. Lillibridge, B. W. Zhao, and E. S. Yang, HP Labs, 1501 Page Mill Road, Palo Alto, CA 94304; emails: {mark.lillibridge, bill.zhao, elizabeth.yang}@hpe.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0734-2071/2016/10-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2992782

opaque and confidential nature of typical flash translation layer (FTL) firmware, the behavior of full devices in unusual conditions is still a mystery to the public.

This article considers the behavior of flash-based SSDs (which we refer to as SSDs from this point forward) under fault. Specifically, we consider how commercially available SSDs behave when power is cut unexpectedly during operation. As SSDs are replacing spinning disk as the non-volatile component of computer systems, the extent to which they are actually non-volatile is of interest. Although loss of power seems like an easy fault to prevent, recent experience [Miller 2012; Leach 2012; McMillan 2012; Claburn 2012] shows that a simple loss of power is still a distressingly frequent occurrence even for sophisticated datacenter operators like Amazon. If even well-prepared and experienced datacenter operators cannot ensure continuous power, then it becomes critical that we understand how our non-volatile components behave when they lose power.

By creating an automatic failure testing framework, we subjected 17 SSDs from six different vendors to more than 3,000 fault injection cycles in total. Surprisingly, we find that *14 of the 17* devices, including the supposedly "enterprise-class" devices, exhibit failure behavior contrary to our expectations. Every failed device lost some amount of data that we would have expected to survive the power fault. Even worse, 2 of the 15 devices became massively corrupted, with one no longer registering on the SAS bus at all after 136 fault cycles and another suffering one third of its blocks becoming inaccessible after merely 8 fault cycles. More generally, our contributions include:

—**Hardware to inject power faults into block devices.** Unlike previous work [Yang et al.] that simulates device-level faults in software, we actually cut the power to real devices. Furthermore, we purposely used a side-channel (the legacy serial port bus) to communicate with our power cutting hardware, so none of the operating system (OS), device driver, bus controller, or the block device itself have an opportunity to perform a clean shutdown.

—**Software to stress the devices under test and check their consistency post-fault.** We propose a specially crafted workload that is stressful to a device while allowing efficient consistency checking after fault recovery. Our record format includes features to allow easy detection of a wide variety of failure types with a minimum of overhead. Our consistency checker detects and classifies both standard "local" failures (e.g., bit corruption) as well as "global" failures such as lack of serializability. Further, the workload is designed considering the advanced optimizations modern SSD firmwares use in order to provide a maximally stressful workload.

—**Experimental results for 17 different SSDs and two hard drives.** Using our implementation of the proposed testing framework, we have evaluated the failure modes of 17 commodity SSDs as well as two traditional spinning-platter hard drives for comparison. Our experimental results show that SSDs have counterintuitive behavior under power fault: Of the tested devices, only three SSDs and one enterprise-grade spinning disk adhered strictly to the expected semantics of behavior under power fault. *Every other drive failed to provide correct behavior under fault*. The unexpected behaviors we observed include bit corruption, shorn writes, unserializable writes, metadata corruption, and total device failure.

Note that this article is an improvement over our previous work [Zheng et al. 2013]. Specifically, in this article, we study a new type of potential failure based on the characteristics of flash memory (i.e., read disturbs); we discover that a recent kernel patch could change the behavior of serialization errors on SSDs significantly; we design an advanced circuit for fault injection; we analyze the characteristics of unserializable writes and shorn writes; we measure the current waveform of SSD operations, which may explain certain failure behaviors; we verify the built-in power loss protection

mechanisms and compare the performance of selected devices, which may explain the design tradeoffs; and two more advanced SSDs are evaluated.

SSDs offer the promise of vastly higher performance operation; our results show that many of them do not provide reliable durability under even the simplest of faults: loss of power. Although the improved performance is tempting, for durability-critical workloads many currently available flash devices are inadequate. Careful evaluation of the reliability properties of a block device is necessary before it can be truly relied upon to be durable.

2. BACKGROUND

In this section, we will give a brief overview of issues that directly pertain to the durability of devices under power fault.

2.1. NAND Flash Low-Level Details

The component that allows SSDs to achieve their high level of performance is NAND flash [Tal 2002]. NAND flash operates through the injection of electrons onto a "floating gate." If only two levels of electrons (e.g., having some vs. none at all) are used, then the flash is single-level cell (SLC); if instead many levels (e.g., none, some, many, lots) are used, then the device is a multi-level cell (MLC) encoding 2 bits per physical device or possibly even an eight-level/3-bit "triple-level cell" (TLC).

In terms of higher-level characteristics, MLC flash is more complex, slower, and less reliable compared to SLC. A common trick to improve the performance of MLC flash is to consider the 2 bits in each physical cell to be from separate logical pages [Takeuchi et al. 1998]. This trick is nearly universally used by all MLC flash vendors [Personal Communication 2012]. However, since writing[1] to a flash cell is typically a complex, iterative process [Liu et al. 2012], writing to the second logical page in a multi-level cell could disturb the value of the first logical page. Hence, one would expect that MLC flash would be susceptible to corruption of previously written pages during a power fault.

NAND flash is typically organized into *erase blocks* and *pages*, which are large-sized chunks that are physically linked. An erase block is a physically contiguous set of cells (usually on the order of 1/4 to 2MB) that can only be zero-ed all together. A page is a physically contiguous set of cells (typically 4KB) that can only be written to as a unit. Typical flash SSD designs require that small updates (e.g., 512 bytes) that are below the size of a full page (4KB) be performed as a read/modify/write of a full page.

The floating gate inside a NAND flash cell is susceptible to a variety of faults [Grupp et al. 2012; Liu et al. 2012; Grupp et al. 2009; Sanvido et al. 2008] that may cause data corruption. The most commonly understood of these faults is write endurance: Every time a cell is erased, some number of electrons may "stick" to the oxide layer separating the floating gate from the substrate, and the accumulation of these electrons limits the number of program/erase cycles to a few thousand or tens of thousands. However, less well known faults include program disturb (where writes of nearby pages can modify the voltage on neighboring pages), read disturb (where reading of a neighboring page can cause electrons to drain from the floating gate), and simple aging (where electrons slowly leak from the floating gate over time). All of these faults can result in the loss of user data. Note that program disturb and read disturb are also called inter-cell interference [Taranalli et al. 2015; Qin et al. 2014].

[1]We use the terms "programming" and "writing" to mean the same thing: injecting electrons onto a floating gate. This is distinct from erasing, which is draining electrons from all of the floating gates of a large number of cells.

Table I. Brief Description of the Types of Failures We Attempted to Detect

Failure	Description
Bit Corruption	Records exhibit random bit errors
Flying Writes	Well-formed records end up in the wrong place
Shorn Writes	Operations are partially done at a level below the expected sector size
Metadata Corruption	Metadata in FTL is corrupted
Dead Device	Device does not work at all, or mostly does not work
Unserializability	Final state of storage does not result from a serializable operation order
Read disturb	Reading of a neighboring page causes electrons to drain from the floating gate

2.2. SSD High-Level Concerns

Because NAND flash can only be written an entire page at a time, and only erased in even larger blocks, namely erase blocks, SSDs using NAND flash have sophisticated firmware, called an FTL, to make the device appear as if it can do update-in-place. The mechanisms for this are typically reminiscent of a log-structured filesystem [Rosenblum and Ousterhout 1992]. The specifics of a particular FTL are generally considered confidential to the manufacturer; hence, general understanding of FTL behavior is limited. However, the primary responsibility of an FTL is to maintain a mapping between logical and physical addresses, potentially utilizing sophisticated schemes to minimize the size of the remapping tables and the amount of garbage collection needed [Lee et al. 2008]. Some FTLs are quite sophisticated and will even compress data in order to reduce the amount of wear imposed on the underlying flash [Ku 2011].

Because writing out updated remapping tables for every write would be overly expensive, the remapping tables are typically stored in a volatile write-back cache protected by a large supercapacitor. A well-designed FTL will be conservative in its performance until the supercapacitor is charged, which typically takes under a minute from power on [2012]. Due to cost considerations, manufacturers typically attempt to minimize the size of the write-back cache as well as the supercapacitor backing it.

Loss of power during program operations can make the flash cells more susceptible to other faults, since the level of electrons on the floating gate may not be within usual specification [Tseng et al. 2011]. Erase operations are also susceptible to power loss, since they take much longer to complete than program operations [Birrell et al. 2007]. The result of corruption at the cell level is typically an error correction code (ECC) error at a higher level. An incompletely programmed write may be masked by the FTL by returning the old value rather than the corrupted new value if the old value has not been erased and overwritten yet [Personal Communication 2012].

3. TESTING FRAMEWORK

Rather than simply writing to a drive while cutting power, it is important to carefully lay out a methodology that is likely to cause failures; it is equally important to be prepared to detect and classify failures.

Table I briefly describes the types of failures that we expected to uncover. For each of these, we have an underlying expectation of how a power loss could induce this failure behavior. Because half-programmed flash cells are susceptible to bit errors, of course, we expected to see bit corruption. Unlike in spinning disks, where flying writes[2] are caused by errors in positioning the drive head (servo errors), we expected to see flying writes in SSDs due to corruption and missing updates in the FTL's remapping tables. Because single operations may be internally remapped to multiple flash chips to improve throughput, we expected to see shorn writes. Because an FTL is a complex

[2]"Flying writes" is also called "misdirected writes" [Krioukov et al. 2008a].

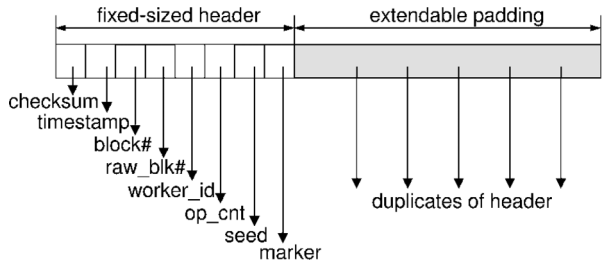

Fig. 1. Record format.

Table II. Description of Header Fields and Why They Are Included

Field	Reason
Checksum (*checksum*)	Efficiently detect bit corruption and shorn writes
Timestamp (*timestamp*)	Allows checking for unserializable writes
Block Number (*block#*)	Detects flying writes
Raw Block Number (*raw_blk#*)	Detects errors in device size and workload code
Worker Id (*worker_id*)	Allows checking for unserializable writes and to regenerate a workload
Operation Count (*op_cnt*)	Allows checking for unserializable writes and to regenerate a workload
Seed (*seed*)	Allows regeneration of the workload
Marker (*marker*)	Allows easy detection of header boundary

piece of software, and corruption of its internal state could be problematic, we expected to see metadata corruption and dead devices. Moreover, due to the high degree of parallelism inside an SSD, and because we feel that current FTLs are likely not as well tested as the firmware of traditional spinning disks, we expected to see unserializable writes. In addition, because the flash cells programmed at the powrer fault may be more sensitive to the noise caused by reading neighboring cells, we expected that the read disturb phenomenon may be more severe.

Of course, expecting to see all of these things is one thing; to actually see them one needs to be capable of detecting them if they do happen. Overall, these failures can be divided into two sorts: local consistency failures and global consistency failures. Most of the failures (e.g., bit corruption and flying writes) can be detected using local-only data: Either a record is correct or it is not. However, unserializability is a more complex property: Whether the result of a workload is serializable depends not only on individual records but also on how they can fit into a total order of all the operations, including operations that may no longer be visible on the device.

3.1. Detecting Local Failures

Local failures require examining only a single record to understand. Hence, these can be checked in a single read-pass over the entire device.

In order to detect local failures, we need to write records that can be checked for consistency. Figure 1 shows the basic layout of our records: a header with fields that allow consistency checking and repetitions of that header to pad the record out to the desired length. Table II shows the structure of the header format.

Some of the fields in the header are easy to explain. Including a *checksum* field is obvious: It allows detection of bit corruption. Similarly, to check the ordering of operations (the heart of serializability checking), a *timestamp* is necessary to know when they occurred. Finally, a *marker* allows easy visual detection of header boundaries when a dump is examined manually.

However, the other fields bear some discussion. The *block#* field is necessary to efficiently check flying writes. A flying write is when an otherwise correct update is applied to the wrong location. This is particularly insidious because it cannot be detected by a checksum over the record. However, by explicitly including the location, we intended to write to in the record itself, flying writes become easy to detect.

The *raw_blk#* field is more straightforward: We generate our workload by generating 64-bit random numbers, and our devices are not 2^{64} records long. By including the raw random number, we can check that the workload knew the number of records on the device and performed the modulo operation correctly. An unfortunate bug in an early version of our workload code motivates this field.

The *worker_id*, *op_cnt*, and *seed* fields work together for two purposes. First, they allow us to more efficiently check serializability, as described in Section 3.2. Second, all three of them together allow us to completely regenerate the full record except timestamp and to determine if our workload would have written said record (e.g., as opposed to garbage from an older run). To do this, we do not use a standard psuedorandom number generator like a linear-shift feedback register or Mersenne Twister. Instead, we use a hash function in counter mode. That is, our random number generator (RNG) generates the sequence of numbers $r_0, r_1, \ldots, r_n$, where $r_{op_cnt} = hash(worker_id, seed, op_cnt)$. We do this so we can generate any particular r_i in constant time, rather than in $O(i)$ time, as would be the case with a more traditional RNG.

By storing the information necessary to recreate the complete record in its header, we can not only verify that the record is correct but also identify partial records written in a shorn write so long as at least one copy of the header is written. Although this also allows detection of bit corruption, using the checksum plus the block number as the primary means of correctness checking improves the performance of the checker by 33% (from 15 minutes down to 10 minutes). Storing the checksum alone does not allow accurate detection of shorn writes, and hence both capabilities, quick error detection from the checksum and detailed error categorization from the recreatable record, are necessary.

3.1.1. Dealing with Complex FTLs. Because we are examining the block level behavior of the SSDs, we want the size of a record to be the same as the common block size used in the kernel, which is typically 4KB. Hence, we need to fill out a full record beyond the relatively small size of the header. A naive padding scheme may fill in the padding space of different records with identical values, which make records largely similar. This approach does not allow us to detect shorn writes, since the records asides from the header would be identical: If the shorn write only writes the later portion of the record, then it would be undetectable. Another approach is to fill the padding of the record with random numbers. This indeed makes each record unique, but while we can detect a shorn write, we cannot identify the details: Without the header, we do not know where the partial record came from.

A much better design is to pad with copies of the header. This not only makes each record unique but also provides redundancy in the face of corruption and allows us to tell which write was partially overwritten in the case of a shorn write.

However, a particular optimization of some FTLs means we cannot quite be this straightforward. By padding the record with duplicated headers, the whole record exhibits repetitive patterns. As mentioned in Section 2, some advanced SSDs may perform compression on data with certain patterns. Consequently, the number of bytes written to the flash memory is reduced, and the number of operations involved may also be reduced. This conflicts with our desire to stress test SSDs. In order to avoid such compression, we further perform randomization on the regular record format.

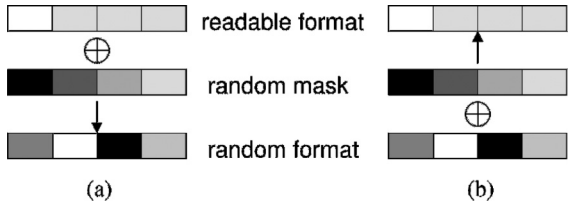

Fig. 2. Randomization of record format. (a) Generate the random format by XORing the readable format with the random mask. (b) Recover from the random format by XORing it with the random mask again.

As shown in Figure 2, we XOR the record with a random mask before sending it to the device. This creates a less compressible format. Similarly, we can recover from the random representation by XORing with the same mask again. By using the same random mask for each record, we can convert between the understandable format and the less compressible format without knowing which record was written. In this way, we avoid the interference of compression, while maintaining a readable record format.

3.2. Detecting Global Failures

Local failures, such as simple bit corruption, are straightforward to understand and test for. Unserializability, on the other hand, is more complex. Unserializability is not a property of a single record, and thus cannot be tested with fairly local information; it may depend on the state of all other records.

According to the POSIX specification, a write request is synchronous if the file is opened with the O_SYNC flag. For a synchronous write request, the calling process thread will be blocked until the data have been physically written to the underlying media. An unserialized write is a violation of the synchronous constraint of write requests issued by one thread or between multiple threads; that is, a write that does not effect the media in a manner consistent with the known order of issue and completion. For the simple case of a single thread, synchronous requests should be committed to disk in the same order as they are issued. For example, in Figure 3(a), thread A writes three records (i.e., $A1$, $A2$, and $A3$) synchronously. Each record is supposed to be commited to the physical device before the next operation is issued. Both $A1$ and $A2$ are issued to the same address (block#1), while $A3$ is issued to another block (block#3).

Figure 3(b) shows two states of the recovered device. The fact that $A3$ is visible on the recovered device (both states) indicates all of the previous writes should have completed. However, in the unserializable state shown, $A2$ is not evident. Instead, $A1$ is found in block#1, which means that either $A2$ was lost or $A2$ was overwritten by $A1$. Neither is correct, since $A1$ was before $A2$. Similarly, synchronously issued requests from multiple threads to the same addresses should be completed in the (partial) order enforced by synchronization.

Some SSDs improve their performance by exploiting internal parallelism at different levels. Hence we expect a buggy FTL may commit writes out of order, which means that a write "completed" and returned to the user early on may be written to the flash memory after a write that completed later. Further, during a power fault we expect that some FTLs may fail to persist outstanding writes to the flash or may lose mapping table updates; both faults would result in the recovered drive having contents that are impossible given any correct serial execution of the workload. We call such misordered or missing operations *unserialized writes*. Note that there is anecdotal evidence from exports in storage industry [Rajimwale et al. 2011] suggesting that some manufacturers may explicitly ignore requests to force writes to disks in order to boost

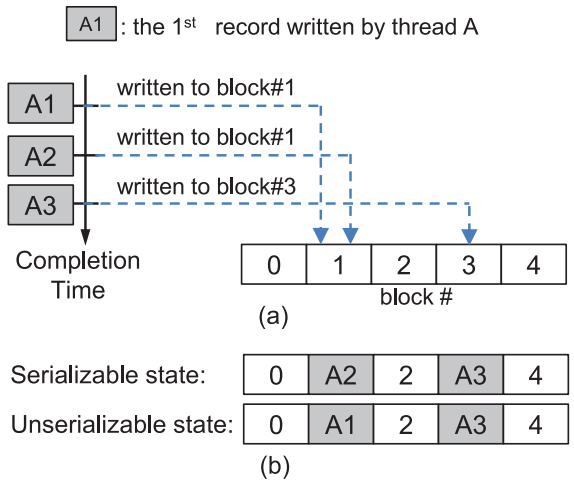

Fig. 3. Serializability of writes within one thread. (a) Thread A synchronously writes three records with both the 1st and 2nd records written to the same address (block#1). (b) In a serializable state, the on-device records reflect the order of synchronous writes; that is, later records (e.g., $A2$) overwrite earlier records (e.g., $A1$) written to the same address. In the shown unserializable state, $A2$ is either lost or overwritten by $A1$.

performance, which could also lead to unserializble write during a power fault. Also, from a users' perspective, unserializable writes may appear as "lost writes" [Triantos 2006]. For example, given the unserializable device state shown in Figure 3(b), $A2$ is invisible and thus is a lost write to the user, even if it was overwritten by $A3$.

To detect unserializability, we need information about the completion time of each write. Unfortunately, a user-level program cannot know exactly when its operations are performed. Given command queueing, even an external process monitoring the serial ATA (SATA) bus cannot pin down the exact time of a write but merely when it was issued and when the acknowledgement of its completion was sent. Instead of using a single atomic moment, we make use of the time when the records were created, which is immediately before they are written to disk (also referred to as thier "generating time"), to derive the completion partial order of synchronous writes conservatively. For example, in Figure 4, thread A generates and commits two records ($A1$ and $A2$) in order. The completion time of $A1$ (i.e., $A1'$) must be bounded by the generating times of $A1$ and $A2$ since they are synchronous and ordered within one thread. In other words, there is a happens-before relation among these operations within one thread. Similarly, $B1$ happens before $B1'$, which happens before $B2$. Cross-thread, the system time gives us a partial happens-before relation. Figure 4(a) shows how this partial order may allow two writes to happen "at the same time," while Figure 4(b) shows completion times $A1'$ and $B1'$ unambiguiously ordered.

For efficiency reasons, we do not store timestamps off of test device and thus must get them from the recovered device. Because records may be legitimatly overwritten, this means we must be even more conservative in our estimates of when an operation happens. If no record is found for an operation, which prevents us from just directly retrieving its timestamp, then we fall back to using the (inferred) timestamp of the previous operation belonging to the same thread or the start of the test if no such operation occurred.

Given a set of operations and ordering/duration information about them, we can determine if the result read at the end is consistent with a serializable execution of the operations (e.g., using the algorithms of Golab et al. [2011]). Given the information

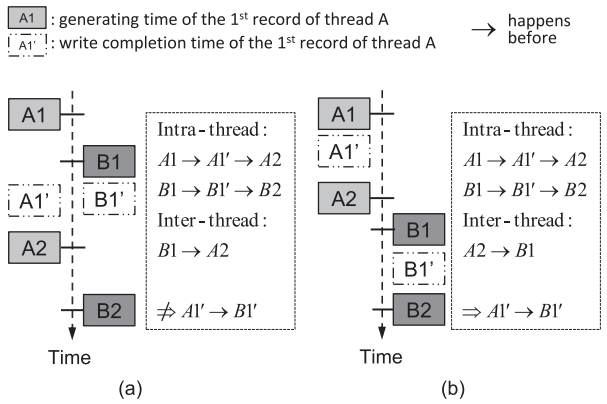

Fig. 4. Deriving the completion-time partial order based on the generating time of records. (a) Within one thread, the write completion time of a record (e.g., $A1'$) is bounded by the generating time of the record (e.g., $A1$) and the next record (e.g., $A2$). Thus, $A1$ happens before its completion ($A1'$), which happens before $A2$. Similarly, $B1$ happens before $B1'$, which happens before $B2$. Between threads, $B1$ happens before $A2$ based on their timestamps. However, this relation does not imply $A1'$ happens before $B1'$. (b) Here $A2$ happens before $B1$. Therefore, due to transitivity, the completion of $A1$ ($A1'$) happens before $B1'$.

we have available, it is not possible in general to determine which writes, if any, were unserialized: Consider observing order A3 A1 A2; is A3 reordered early or are both A1 and A2 reordered late? We can, however, determine the minimal number of unserialized writes that must have occurred (1 here). More precisely, our algorithm conservatively detects *serialization errors*, each of which must have been caused by at least one different unserialized write. Some serialization errors may actually have been caused by multiple unserialized writes, and some unserialized writes may not have caused detectable serialization errors. We thus provide a lower bound on the number of unserialized writes that must have occurred.

Specifically, our algorithm works as follows: The record format allows us to reconstruct the full operation stream (who wrote what where). For example, by reading the Worker Id (e.g., $worker_id$ = A) and Seed (e.g., $seed = 999$) fields of a record, we regenerate the Raw Block Number ($raw_blk\#$) of the Nth operation (op_cnt = N) of the same thread by using a hash function: $hash(worker_id = A, seed = 999, op_cnt = N)$. We then further convert the $raw_blk\#$ to $block\#$ via modulo as described in Section 3.1. In this way, we identify the expected location (i.e., $block\#$) of the Nth operation of thread A. One thread at a time, we consider that thread's operations until the last one of its operations visible on disk (we presume later writes simply were not issued/occurred after the power fault) and examine the corresponding locations to which they wrote. For each location, we expect to see the result of (1) the operation we are examining; (2) an unambiguiously later operation, as per the happens-before relations we can establish from the timestamps in the observed records; (3) an unambiguiously earlier operation; or (4) an operation that could have happened "at the same time." The orders of operations are derived using the generating time of records as described in Figure 4. We count instances of the third case as serialization errors.

3.3. Detecting Read Disturb

Besides the local failures and the global failure, another potential failure mode is read disturb, where reading of a neighboring page can cause electrons to drain from the floating gate. For fully programmed pages (i.e., no power faults), it typically takes several milllion read operations to manifest read disturb [Grupp et al. 2009]. As for

ALGORITHM 1: Read Disturb Detection

```
last_op = find_last_op(disk_image);
iter_num = N;
repeat
    iter_num - -;
    for op_i from last_op downto last_op - M do
        blk_i = get_blk_num(op_i);
        for blk_j from blk_i - CHECK_RANGE to blk_i + CHECK_RANGE do
            rec_j = read_rec(disk_image, blk_j);
            blk_j += 1;
        end
        op_i -= CHECK_STEP;
    end
until iter_num == 0;
for op_i from last_op downto 1 do
    blk_i = get_blk_num(op_i);
    rec_i = read_rec(disk_image, blk_i);
    check_rec(rec_i);
end
```

the "half-programed" pages during power faults, however, read disturb could appear at the flash-level after merely 1,000 reads [Tseng et al. 2011]. This section discusses our design to detect device-level read disturb.

As shown in Algorithm 1, we perfom N iterations of reads before checking the potential disturb. In each iteration, we consider the last M write operations observable on the disk image after a power fault because these operations could be more susceptible to read disturb. In each (i.e., when CHECK_STEP = 1) of the M operations, we find the corresponding disk block and read its neighboring blocks. The range of neighborhood is controlled by a parameter (i.e., CHECK_RANGE). After the M iterations of reads, we check the intergrity of all records on the disk.

3.4. Applying Workloads

The main goal of applying workloads to each SSDs for this work is to trigger as many internal operations as possible. To this end, we design three types of workloads: concurrent random writes, concurrent sequential writes, and single-threaded sequential writes.

Let us first consider concurrent random writes. The garbage collector in the FTL usually maintains an allocation pool of free blocks, from which the new write requests can be served immediately. From this point of view, the access pattern (i.e., random or sequential) of the workloads does not matter much in terms of performance. However, in the case of sequential writes, the FTL may map a set of continuous logical addresses to a set of continuous physical pages. This continuous mapping has two benefits: First, it can reduce the amount of space required by the mapping table, and, second, it makes garbage collection easier since the continuous allocation requires fewer merging operations of individual pages. By comparison, random writes may scatter to random pages, which could lead to more merging operations when recycling blocks. We thus consider random writes the most stressful workload for a SSD.

To make the random write workload even more stressful, we use concurrent threads to fully exercise the internal parallelism of a SSD. More specifically, we use a number of worker threads to write records to the SSD concurrently. Each worker thread is associated with a unique seed. The seed is fed into a random number generator to generate the address of each record, which determines the location that record should

be written to. The worker thread writes the record to the generated address and then continues on to generate the next record. In this way, multiple workers keep generating and writing records to random addresses of the SSD.

The second type of workload is concurrent sequential writes. As mentioned above, the sequential pattern may trigger a FTL sequential mapping optimization. In the concurrent sequential workload, each worker thread writes records sequentially, starting from a random initial address. In other words, there are several interleaved streams of sequential writes, each to a different section of the SSD. Advanced SSDs have multiple levels of parallelism internally and may detect this kind of interleaved pattern to make full use of internal resources.

The third type of workload is single-threaded sequential writes. Some low-end SSDs may not be able to detect the interleaved access pattern of the concurrent sequential writes workload. As a result, the concurrent sequential writes may appear as random writes from these devices' point of view. So we designed the single-threaded sequential workload to trigger the internal sequential optimization, if any, of these devices. Specifically, in this workload, there is only one worker thread generating and writing records to continuous addresses of the SSD.

3.5. Power Fault Injection

While the worker threads are applying a workload to stress a SSD, the fault injection component, which we call the Switcher, cuts off the power supply to the SSD asynchronously to introduce a sudden power loss.

The Switcher includes two parts: a custom hardware circuit and the corresponding software driver. The custom circuit controls the power supply to the SSD under test, which is independent from the power supply to the host machine. This separation of power supplies enables us to test the target device intensively without jeopardizing other circuits in the host systems. Figure 7 shows the photographs of two versions of the control circuit.

There were several improvements made on the circuit shown in Figures 5 and 6, although the purpose of the hardware essentially remained the same. The initial circuit performed experiments through serial port; it was ideal given the equipment present. However, this posed to be an impediment for future experiments due to specifications on serial ports so they work only on ports with voltages of 6 or higher. As a result, the hardware shorted with a workstation that had a serial port voltage of 4.5.

To solve the issue, the hardware was redesigned to use general purpose input output (GPIO) and universal serial bus (USB) plugin instead. This proved to be an efficient change as it covered for newer machines with more ideal specifications of 5V or 3.3V.

One major adjustment to the hardware was the conversion from breadboard to PCB layout, which made for more efficient experimentation where eight SSDs could be tested simultaneously. The pricing of the PCB circuit in total was around $214.16, whereas the initial circuit cost approximately $60.04. The significant price increase was compensated for by the fact that $214.16 was needed to test up to eight SSDs, whereas the price of the old circuit would cost $60.04 to test a single SSD. Had we continued with the initial design to test multiple devices, several workstations would have been needed, as most have only one serial port.

Another hardware modification to the circuit was the transition from using complementary metal-oxide-semiconductor (CMOS) logic to p-type metal-oxide-semiconductor (PMOS) logic. This decision made more inexpensive use of parts as the two power transistors creating the NOT gate were no longer needed. Instead, one of the power transistors could be replaced by a resistor.

In addition to economic reasons, this design alteration was preferred as it was less susceptible to shorting. The new design has a pull-down resistor to have a known voltage for when the GPIO is powered off and an additional resistor that functions

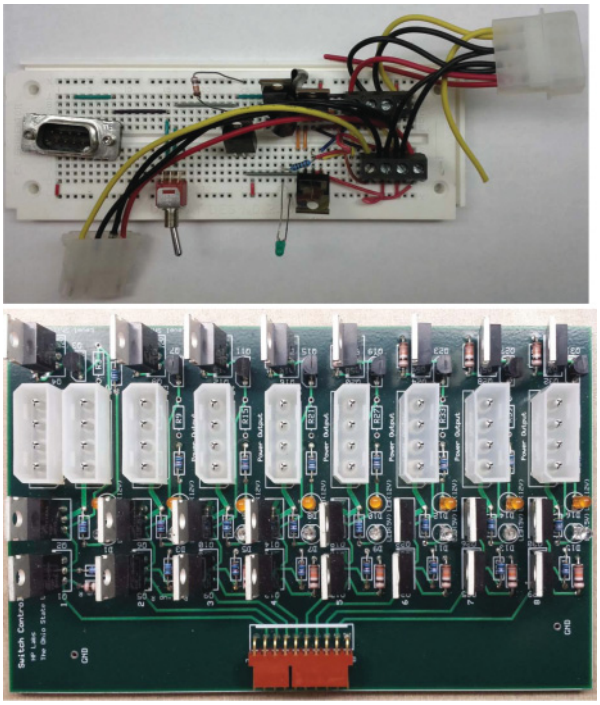

Fig. 5. The hardware used to cut power in our experiments. The top one is the initial breadboard circuit, while the bottom one is the enhanced PCB circuit.

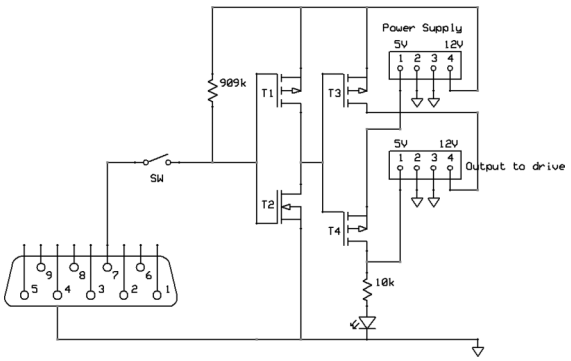

Fig. 6. Schematic of intial circuit.

both as a current limiting component that creates a ceiling to the voltage range and as an essential part to the PMOS logic.

However, this also introduced a few problems. A major issue was that the 12V line was not being powered enough. This prevented testing of SSDs that required the 12V line and spinning disk drives. The root of the inaccuracy was found to be within the logic level shift component in the circuit, which provided a residue current to the line going to ground, thus blocking one of the P-channel MOSFETs from powering on. The former NOT gate configuration had a pull-up resistor connected to one of the transistors in order to have a known voltage for when the toggle switch was off. The benefit to keeping the old system would be that it would be a temporary solution to

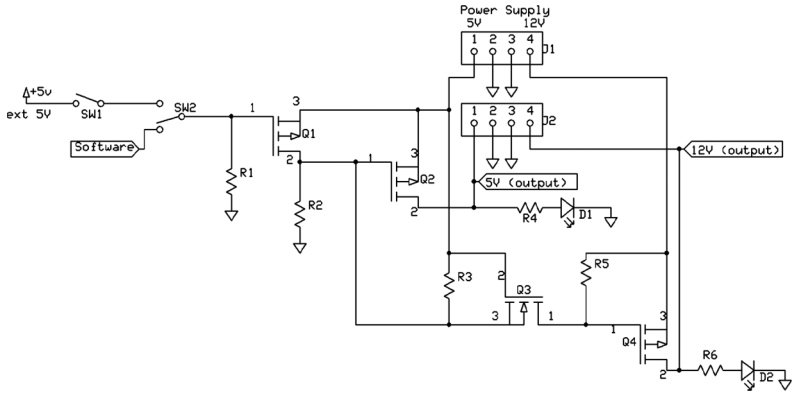

Fig. 7. Schematic of improved circuit.

the inability to test spinning disk drives and SSDs using the 12V line. However, if we re-implement this design, it would be more vulnerable to shorting as there are no current limiting resistors, and the cost efficiency would become null. Therefore, we plan to further enhance the circuit by instead using a solid state relay to replace the faulty logic level configuration in the latest hardware.

The software part of the Switcher is used to send commands to the hardware circuit to cut off the power during workload application and to turn the power back on later to allow recovery and further checking. In this way, we can cut the power and bring it back automatically and conveniently.

In order to cut off power at different points of the SSDs' internal operations, we randomize the fault injection time for each different testing iteration. More specifically, in each iteration of testing we apply workloads for a certain amount of time (e.g., 30s). Within this working period, we send the command to cut off the power at a random point of time (e.g., at the 8th second of the working period). Since SSD internal operations are usually measurable in milliseconds, the randomized injection time in seconds may not make much difference in terms of hitting one particular operation during power loss. However, the randomized injection points change the number of records that have been written before the power fault and thus affect whether there are sufficient writes to trigger certain operations (e.g., garbage collection). By randomizing the fault injection time, we test the SSD under a potentially different working scenario each iteration.

3.6. Putting It Together

This section briefly summarizes the work flow of our proposed framework. As shown in Figure 8, there are four major components: Worker, Switcher, Checker, and Scheduler.

The Scheduler coordinates the whole testing procedure. Before workloads are applied, it first selects one Worker to initialize the whole device by sequentially filling in the SSD with valid records. This initialization makes sure the mapping table of the whole address space is constructed. Also, it is easier to trigger background garbage collection operations if all visible blocks of the SSD (there may be extra non-visible blocks if the SSD has been over provisioned) are filled with valid data. Additionally, the full initialization allows us to detect bad SSD pages.

After the initialization, the Scheduler schedules the Workers to apply workloads to the SSD for a certain working period (e.g., 30s). Meanwhile, the Switcher waits for a random amount of time within the working period (e.g., 8s) and then cuts down the power supply to the SSD. Note that Workers keep working for several seconds after the injected power failure; this design choice guarantees that the power fault is injected

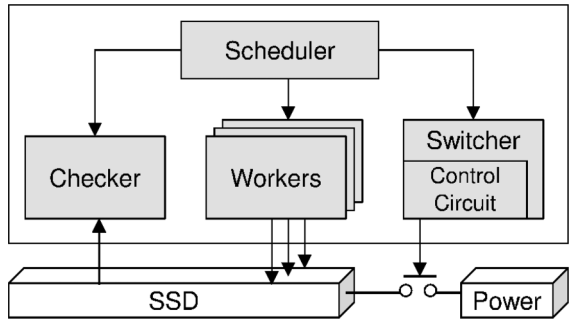

Fig. 8. Framework for testing SSDs using injected power faults.

when the SSD is being actively exercise by a workload. Of course, the input/output (I/O) requests after the power fault only return I/O errors and do not effect the SSD. After this delay, the Scheduler stops the Workers, and the power is brought back up again by the Switcher. Then, the Checker reads the records present on the restarted SSD, and checks the correctness of the device state based on these records. All potential errors found are written to logs at this stage.

The testing procedure is executed iteratively. After the checking, the next iteration starts with the Scheduler selecting one Worker to initialize the device again. In this way, a device can be tested repeatedly.

4. EXPERIMENTAL ENVIRONMENT

4.1. Block Devices

Of the wide variety of SSDs available today, we selected 17 representative SSDs (12 different models) from six different vendors.[3] The characteristics of these SSDs are summarized in Table III. More specifically, the prices of the 17 SSDs range from low end (e.g., $0.63/GB) to high end (e.g., $6.50/GB), and the flash memory chips cover both MLC and SLC. Based on our examination and manufacturer's statements, five SSDs are equipped with power-loss protection. For comparison purposes, we also evaluated two traditional hard drives, including one low-end drive (5.4K RPM) and one high-end drive (15K RPM).

4.2. Host System

Our experiments were conducted on a machine with a Intel Xeon 5160 3.00GHz central processing unit (CPU) and 2GB of main memory. The SSDs and the hard drives are individually connected to an LSI Logic SAS1064 PCI-X Fusion-MPT SAS Controller. The power supply to the SSDs is controlled by our custom circuit.

The operating system is Debian Linux 6.0 with kernel 2.6.32. While evaluating new devices, we notice that a kernel patch for v2.6.32 might change the behavior of some devices in terms of serializability. Thus we also re-evaluate some devices on kernel 2.6.33 that applied the patch (see Section 5.5.2).

To explore the block-level behavior of the devices and minimize the interference of the host system, the SSDs and the hard drives are used as raw devices: That is, no file system is created on the devices. We use synchronized I/O (O_SYNC), which means each write operation does not return until its data is flushed to the device. By inspection, this does cause cache flush commands to be issued to the devices. Further, we set the I/O scheduler to *noop* and specify direct I/O (O_DIRECT) to bypass the buffer cache.

[3]Vendor names are blinded; we do not intend to "name-and-shame."

Table III. Characteristics of the Devices Used in Our Experiments. "Type" for SSDs Means the Type of the Flash Memory Cell While for HDDs It Means the revolutions per minute (RPM) of the Disk. "P" Indicates Presence of Some Power-Loss Protection (e.g., a Super Capacitor)

Device ID	Vendor-Model	Price ($/GB)	Type	Year	P?
SSD#1	A-1	0.88	MLC	'11	N
SSD#2	B	1.56	MLC	'10	N
SSD#3	C-1	0.63	MLC	'11	N
SSD#4	D-1	1.56	MLC	'11	-
SSD#5	E-1	6.50	SLC	'11	N
SSD#6	A-2	1.17	MLC	'12	Y
SSD#7	E-2	1.12	MLC	'12	Y
SSD#8	A-3	1.33	MLC	'11	N
SSD#9	A-3	1.33	MLC	'11	N
SSD#10	A-2	1.17	MLC	'12	Y
SSD#11	C-2	1.25	MLC	'11	N
SSD#12	C-2	1.25	MLC	'11	N
SSD#13	D-1	1.56	MLC	'11	-
SSD#14	E-1	6.50	SLC	'11	N
SSD#15	E-3	0.75	MLC	'09	Y
SSD#16	F-1	1.25	MLC	'11	N
SSD#17	A-4	5.60	MLC	'11	Y
HDD#1	F	0.33	5.4K	'08	-
HDD#2	G	1.64	15K	-	-

Table IV. Number of Faults Applied to Each Device During Concurrent Random Writes Workload

Device ID	SSD#1	SSD#2	SSD#3	SSD#4	SSD#5	SSD#6	SSD#7	SSD#8	SSD#9	SSD#10
# of Faults	136	1360	8	186	105	105	250	200	200	100
Device ID	SSD#11	SSD#12	SSD#13	SSD#14	SSD#15	SSD#16	SSD#17	HDD#1	HDD#2	—
# of Faults	171	100	100	233	103	50	50	1	24	—

5. EXPERIMENTAL EXAMINATIONS

We examined the behavior of our selected SSDs under three scenarios: (1) power fault during concurrent random writes, (2) power fault during concurrent sequential writes, and (3) power fault during single-threaded sequential writes.

For each scenario, we perform a certain number of testing iterations. Each iteration injects one power fault into the target device. Since the workload of concurrent random writes is the most stressful to the SSDs, we conducted most tests using this workload type. Table IV summarizes the total numbers of power faults we have injected into each device during the workload of concurrent random writes. As shown in the table, we have tested more than 100 power faults on each of the SSDs, except for SSD#3, which exhibited non-recoverable metadata corruption after a mere eight power faults (see Section 5.7 for more details).

As for the other two workloads, we conducted a relatively small number of tests (i.e., each workload 20 times on two drives). One drive did not show any abnormal behavior with these workloads, while the other exhibited similar behavior as with the concurrent random workload. This verifies that concurrent random writes is the most stressful workload for most SSDs, making them more susceptible to failures caused by power faults. In this section, we accordingly focus on analyzing the results exposed by power failures during concurrent random writes.

Table V. Summary of Observations. "Y" Means the Failure was Observed With any Device, While "N" Means the Failure Was Not Observed

Failure	Seen?	Devices exhibiting that failure
Bit Corruption	**Y**	SSD#11, SSD#12, SSD#15
Flying Writes	N	—
Shorn Writes	**Y**	SSD#5, SSD#14, SSD#15
Unserializable Writes	**Y**	SSD#2, SSD#4, SSD#7, SSD#8, SSD#9, SSD#11, SSD#12, SSD#13, SSD#16, HDD#1
Read disturb	N	—
Metadata Corruption	**Y**	SSD#3
Dead Device	**Y**	SSD#1
None	**Y**	SSD#6, SSD#10, SSD#17, HDD#2

5.1. Overall Results

Table V shows the summarized results. In our experiments, we observed five out of the seven expected failure types, including bit corruption, shorn writes, unserializable writes, metadata corruption, and dead device. Surprisingly, we found that *14 of 17* devices exhibit failure behavior contrary to our expectation. This result suggests that our proposed testing framework is effective in exposing and capturing the various durability issues of SSDs under power faults.

Every failed device lost some amount of data or became massively corrupted under power faults. For example, three devices (SSDs no. 11, 12, and 15) suffered bit corruption, three devices (SSDs no. 5, 14, and 15) showed shorn writes, and many SSDs (SSDs no. 2, 4, 7, 8, 9, 11, 12, and 13) experienced serializability errors. The most severe failures occurred with SSD#1 and SSD#3. In SSD#3, about one third of data was lost due to one third of the device becoming inaccessible. As for SSD#1, all of its data were lost when the whole device became dysfunctional. These two devices manifested these unrecoverable problems before we started serializability experiments on them, so we could not include them in our serializability evaluation.

In the following sections, we discuss each potential failure and provide our analysis on possible reasons behind the observations.

5.2. Bit Corruption

We observed random bit corruption in three SSDs. This finding suggests that the massive chip-level bit errors known to be caused by power failures in flash chips [Tseng et al. 2011] cannot be completely hidden from the device-level in some devices. One common way to deal with bit errors is using ECC. However, by examining the datasheet of the SSDs, we find that two of the failed devices have already made use of ECC for reliability. This indicates that the number of chip-level bit errors under power failure could exceed the correction capability of ECC. Some FTLs handle these errors by returning the old value of the page [Personal Communication 2012]. However, this could lead to unserializable writes.

5.3. Flying Writes

We did not observe any flying writes in our experiments. This result indicates that the FTLs evaluated are capable of keeping a consistent mapping between logical and physical addresses even under power fault.

5.4. Shorn Writes

In our experiments, we use 4KB as the default record size, which is the typical block size used in the kernel. We did not expect a shorn write within a 4KB record should or could occur for two reasons: First, 4KB is the common block size used by the kernel as

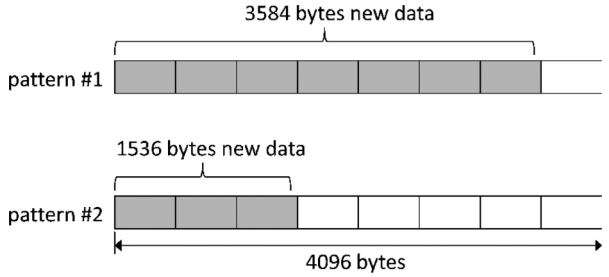

Fig. 9. Two examples of shorn writes observed.

well as the page size used in the SSDs (which can be protected by the associated ECC). In other words, most drives use a 4KB programming unit, which would not allow shorn writes to occur within a 4KB block. Second, SSDs usually serve a write request by sending it to a newly erased block whose location differs from the location that stores the old data so a partially new and partially old record is unlikely to appear.

Contrary to our expectations, we observed shorn writes on three drives: SSDs no. 5, 14, and 15. Among the three, SSD#5 and SSD#14 are the most expensive ones—supposedly "enterprise-class"—in our experiments. Figure 9 shows two examples of shorn-write patterns observed. In pattern #1, the first 3,584 bytes of the block are occupied by the content from a new record, while the remaining 512 bytes are from an old record. In pattern #2, the first 1,536 bytes are new, and the remaining 2,560 bytes are old. In all patterns observed, the size of the new portion is a multiple of 512 bytes. This is an interesting finding indicating that some SSDs use a sub-page programming technique internally that treats 512 bytes as a programming unit, contrary to manufacturer claims. Also, a 4KB logical record is mapped to multiple 512-byte physical pages. As a result, a 4KB record could be partially updated while keeping the other part unchanged. In addition, because SSDs includes on-board cache memory for buffering writes, the shorn writes may also be caused by alignment and timing bugs in the drive's cache management.

In the 441 testing iterations on the three drives, we observed 72 shorn writes in total. This shows that shorn writes is not a rare failure mode under power fault. Moreover, this result indicates even SLC drives may not be immune to shorn writes since two of these devices use SLC. A further analysis of the shorn writes reveals that all of the shorn writes happened on the last record that committed before the power fault.

5.5. Unserializable Writes

We have detected unserializable writes in nine tested SSDs, including nos. 2, 4, 7, 8, 9, 11, 12, 13, and 16. Figure 10 shows the average number of serialization errors observed per power fault for each SSD. Each serialization error implies at least one write was dropped or mis-ordered. The SSDs are sorted in the increasing order of their cost per capacity ($/GB) from left to right. No strong relationship between the number of serialization errors and a SSD's unit price stands out except for the fact that the most expensive SLC drives, SSD#5 and SSD#14, did not exhibit any serialization errors. In addition, we have observed that SSDs with the same model have a similar average number of serialization errors (e.g., SSD#8 and SSD#9, SSD#11 and SSD#12). This suggests that the failure behavior is closely related to the design and implementation of each device model.

Figure 11 shows how the number of serialization errors observed per power fault varied over time for selected SSDs. Notice that SSD#2 and SSD#4 had hundreds of serialization errors result from each fault. In our experiments, the kernel is made

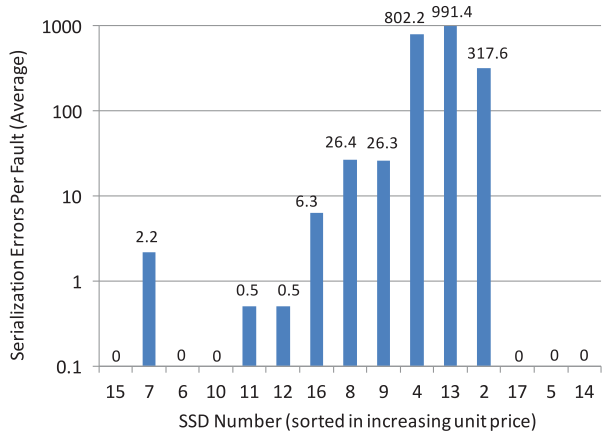

Fig. 10. Average number of serialization errors observed per power fault for each SSD. The x-axis shows the devices from cheapest unit price ($/GB) to most expensive.

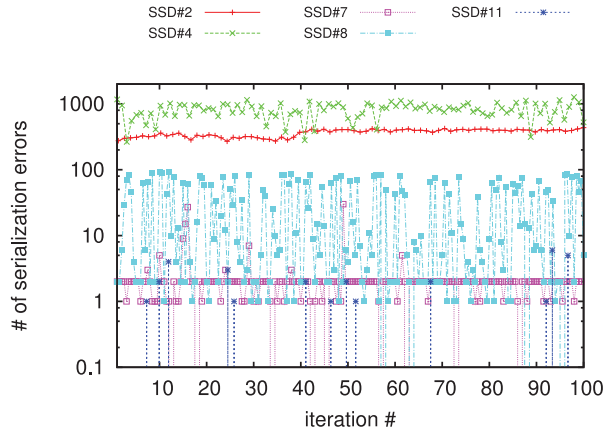

Fig. 11. Number of serialization errors observed in each testing iteration for selected SSDs.

to send commands to the device being tested to flush its write cache at the end of each write request, so a write should be committed on the device before returning. In spite of this, we still observed a large number of serialization errors. This suggests that these SSDs do not try to commit writes immediately as requested by the kernel. Instead, they probably keep most of the recent write requests in volatile buffers for performance reasons. On the other hand, SSDs no. 7, 8, and 11 have only a relatively small number of serialization errors during power faults, suggesting that those SSDs try to commit most of their write requests on time.

Several correlated device design choices may contribute to the serialization errors observed. First, the devices may make extensive use of on-drive cache memory and ignore the flush requests from the kernel as discussed above. As a result, writes in the device's cache will not survive a power fault if one occurs. Second, the devices serve write requests in parallel. Therefore, a write request committed later does not necessarily guarantee that an earlier write request is committed safely. This counterintuitive behavior can lead to severe problems in higher-level storage components.

In a serialization error, a write could be either have its result overwritten by an earlier write or simply be lost. We see three possible scenarios that might cause a lost write:

Table VI. Characteristics of Unserializable Writes

Device ID	SSD#2	SSD#4	SSD#7	SSD#8	SSD#9	SSD#11	SSD#12	SSD#13	SSD#16
Total	21,002	72,645	381	5,348	5,326	88	82	96,530	317
Multi-Workers	19	0	0	105	106	1	1	265	1
Same block	0	0	0	0	0	0	0	9	0

Total means the total number of serialization errors observed on the device. Multi-workers means the number of errors involving multiple worker threads. Same block means the number of errors happened on the same block in a given run.

First, the write may not have been completely programmed into the flash memory at the time of the fault. The program operation for NAND flash may take several program-read-verify iterations before the flash memory page reaches the desired state. This iterative procedure could be interrupted by a power fault leaving the page under programming in an invalid state. Upon restart, the FTL can identify the page as invalid using the valid/invalid bit of the metadata associated with the page, and thus continue to map the logical address being written to the page that contains the old version of the data.

Second, the data to be written may have been successfully programmed into the flash memory before exhausting the emergency power, but the FTL did not get around to updating the page's valid/invalid bit to valid. As a result, the mapping table of the recovered device still maps the written address to the old page even though the new data has been saved in the flash memory.

Third, the data may have been successfully programmed into the flash memory and the page may have been marked as valid, but the page that contains the old record may not have been marked as invalid. As a result, in the restarted device there are two valid pages for the same logical address. When rebuilding the mapping table, the FTL may be unable to distinguish them and thus may fail to recognize the new page as the correct one. It may thus decide to keep the old one and mark the new one as invalid. It is possible to solve this problem by encoding version information into pages' metadata [Gal and Toledo 2005].

5.5.1. Characteristics of Unserializable Writes. We analyze the characteristics of unserializable writes, which is the most prevalent errors we have observed, in more detail. Specifically, we study two characteristics: whether a serialization error involves two worker threads and, for a give trace, how many serialization errors happened at the same block. The two characteristics allow us to infer whether the serialization errors were more likely caused by the competition of multiple threads or simply caused by missing updates within a single thread.

Table VI shows the characteristics. The first row (i.e., Total) shows the total number of serialization errors observed on each device. The second row (i.e., Multi-workers) row shows the number of errors involving multiple threads. The last row (i.e., Same block) shows the number of errors happened on the same block in a given run. We can see that seven of the nine devices have errors involving multiple worker threads. However, the number of these errors is quite small compared to the total number of errors observed on the corresponding device. Also, only one device (i.e., SSD#13) has errors happened at the same disk block. This result is reasonable since we use hash function and modulo operation to calculate the location of write operations (Section 3.1) and thus the chance of mapping different writes from different worker threads to the same location is small. Also, this result may imply that many of the serialization errors did not involve multiple worker threads. In other words, serialization error could happen in a single thread, and could likely be caused by missing updates in one thread (instead of overwritten from another worker thread).

Table VII. Different Behavior of SSDs on Two Kernels. Each Cell Shows the Percentage of Faults with Serialization Errors Among All Faults Injected

Kernel version	SSD#2	SSD#7	SSD#8	SSD#11	SSD#13	SSD#16
2.6.32	100%	97.6%	98%	32.8%	100%	96%
2.6.33-rc1	4%	32%	34%	0%	0%	0%

Table VIII. Different Behavior of SSDs on Two Kernels. Each Cell Shows the Average Number of Serialization Errors Per Power Fault, Averaging Over All Power Faults with Serialization Errors

Kernel version	SSD#2	SSD#7	SSD#8	SSD#11	SSD#13	SSD#16
2.6.32	316.7	2.3	27.3	1.6	991.4	6.6
2.6.33-rc1	88	1.3	1.3	0	0	0

Table IX. Number of Faults and Read Iterations Applied to Devices for Read Disturb Experiments

Device ID	SSD#2	SSD#7	SSD#8	SSD#9	SSD#12	SSD#13
# of Faults	2	2	10	20	10	10
# of Read Iterations per Fault	5,242	5,800	5,000	5,000	5,000	5,000

5.5.2. Serializability on Newer Kernel. While evaluating our new circuit, we observed an interesting behavior that some devices that exhibited many serialization errors on kernel 2.6.32 have much fewer errors on newer kernels. We analyze the kernel patches since 2.6.32 and pinpoint a patch that could change the behavior of SSDs in terms of serializability. The patch fixes the disk cache flushing on fsync [Hellwig 2009].

Table VII compares the different behavior of SSDs on two different kernels. 2.6.32 is the kernel without the patch, while 2.6.33-rc1 is the kernel with the patch. As we can see, the percentage of faults that incur serialization errors is much reduced in the newer kernel. For example, SSD#2 exhibit serialization errors in every fault on kernel 2.6.32. However, on kernel 2.6.33-rc1, only 4% of faults injected on SSD#2 incur serialization errors. As for SSD#11, SSD#13, and SSD#16, they do not have any serialization error on the new kernel.

Table VIII further compares the average number of serialization errors observed per power fault, averaging over all power faults with serialization errors (i.e., excluding power faults that do not incur serialization errors). We can see that the average number of serialization errors occurred in each fault is also reduced significantly. For example, SSD#2 shows more than 300 errors in each fault on kernel 2.6.32, while on the new kernel there is only 88 errors per fault.

The above result implies that many of the serialization errors for SSDs are likely caused by the bug in kernel 2.6.32 instead of the device itself. On the other hand, even after fixing the kernel bug, some devices (e.g., SSD#2, SSD#7, and SSD#8) still exhibit serialization errors, although the number of errors are reduced greatly. Note that some devices (e.g., SSD#6, SSD#10, SSD#17) can behave correctly even without the kernel patch, which implies that it is feasible for SSDs to workaround the kernel bug and achieve serializability.

5.6. Read Disturb

As shown in Table IX, we have performed read disturb experiments on six SSDs, which involved 54 injected faults in total. For each fault, we perform at least 5,000 read iterations, each of which read a range of blocks near the last operations before the fault. At the time of writing, we have not observed any read disturb at the device level. This indicates that although the flash memory cells that are affected during the power fault could become less reliable in terms of read disturb, at device level the errors may

be detected or corrected by ECC code. Thus, the read disturb is not significantly more severe after power failure.

5.7. Metadata Corruption

SSD#3 exhibited an interesting behavior after a small number (8) of tests. SSD#3 has 256GB of flash memory visible to users, which can store 62,514,774 records in our experimental setting. However, after eight injected power faults, only 69.5% of all the records can be retrieved from the device. In other words, 30.5% of the data (72.6GB) was suddenly lost. When we tried to access the device beyond the records we are able to retrieve, the process hangs, and the I/O request never returns until we turn off the power to the device.

This corruption makes 30.5% of the flash memory space unavailable. Since it is unlikely that 30.5% of the flash memory cells with contiguous logical addresses are broken at one time, we assume corruption of metadata. One possibility is that the metadata that keeps track of valid blocks is messed up. Regardless, about one third of the blocks are considered bad sectors. Moreover, because the flash memory in SSDs is usually over-provisioned, it is likely that a certain portion of the over-provisioned space is also gone.

5.8. Dead Device

Our experimental results also show total device failure under power faults. In particular, SSD#1 "bricked"—that is, can no longer be detected by the controller—and thus became completely useless. All of the data stored on it were lost. Although the dysfunction was expected, we were surprised by the fact that we have only injected a relatively small number of power faults (136) into this device before it became completely dysfunctional. This suggests that some SSDs are particularly susceptible to power faults.

The dead device we observed could be the result of an unrecoverable loss of metadata (e.g., the whole mapping table is corrupted), a buggy implementation of firmware (e.g., the unrecoverable state could be caught by an assertion without proper failure handling), or hardware damage due to irregular voltage (e.g., a power spike) during the power loss. To pinpoint the cause, we measured the current at the interface of the dead drive. It turns out that the "bricked" device consumes a level of power similar to a normal device. This suggests that part of the functionality of device is still working, although the host cannot detect it at all.

5.9. Comparison with Hard Drives

For comparison purposes, we also evaluated the reliability of two hard disk drives (HDDs) under power faults. Our testing framework detected unserializable writes with HDD#1, too. This indicates that some low-end hard drives may also ignore the flush requests sent from the kernel. On the other hand, HDD#2 incurred no failures due to power faults. This suggests that this high-end hard drive is more reliable in terms of power fault protection.

5.10. Internal Operations

To further verify the internal operations of SSDs, we measured the current waveform of a SSD during and after a write operation. As shown in Figure 12, during the write operation (i.e., from 0.3 to around 1.27 on x-axis) the current value oscillates between 0.05 to 1.15 amperes (A). After the write operation, however, the current is still unstable. There are short periods of high current draw around 1.31 and 1.59s, which implies that the write has triggered intensive internal operations (e.g., garbage collection or wear-leveling). Consequently, even if the write operation was completed successfully,

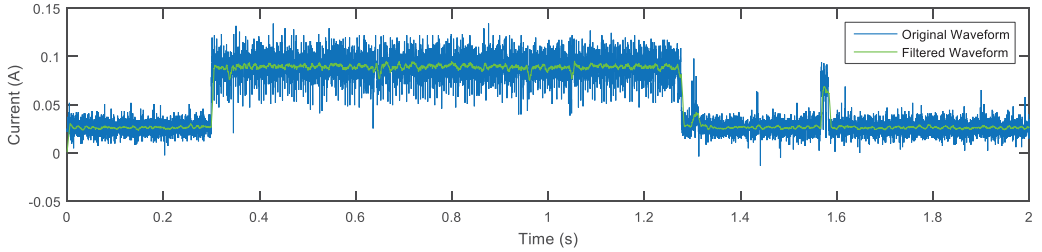

Fig. 12. The current waveform of a SSD during and after a write operation.

Table X. Throughput of Selected Devices Under Our Synchronous Workloads. SSDs 6, 10, 14, and 15 Do Not Have Serialization Errors, while SSDs 2, 7, and 8 Have Serialization Errors

Device ID	SSD#6	SSD#10	SSD#14	SSD#15	SSD#2	SSD#7	SSD#8
MB/s	34.5	35.1	10.1	43.9	75.8	64.4	66.5

a sudden loss of power later may still interrupt the internal operations and lead to severe corruption as discussed in this section.

5.11. Power Loss Protection

Based on the manufacturers' manuals as well as our examination on the internal circuits of the devices, we have verified that five of the devices tested have built-in power loss protection mechanisms (e.g., a supercapacitor for emergency energy). The five devices are SSDs 6, 7, 10, 15, and 17. We can see from Figure 10 that the power loss protection indeed has a discernible impact on the failure behavior. For example, SSDs 6, 10, 15, and 17 do not have any serialization errors in our experiments. On the other hand, SSD 7 still has a couple of serialization errors under each power fault on average, which implies that adding a supercapacitor may not be enough to avoid the errors due to the limited backup energy and the complicated internal operations. Instead, an effective protection requires careful co-design of both hardware and firmware.

5.12. Performance

Table X shows the throughput (MB/s) of selected devices under our synchronous workloads. We divide the results into two groups. The first group of devices (i.e., SSDs 6, 10, 14, and 15) does not have any serialization error, while the second group (i.e., SSDs 2, 7, and 8) has serialization errors even after adding the kernel patch. We can see that the throughput of the first group is generally lower than that of the second group. This may imply that the first group prioritizes reliability over performance over and performs more internal operations to enforce the persistence of writes, while the second group may sacrifice reliability for better performance.

6. RELATED WORK

In this section, we briefly describes related work that has not been covered in the previous sections.

Analysis of raw flash memory and SSDs. Much work has been done on analyzing the reliability of raw flash memory chips [Ong et al. 1993; Brand et al. 1993; Suh et al. 1995; Belgal et al. 2002; Kurata et al. 2006; Grupp et al. 2009; Tseng et al. 2011; Grupp et al. 2012; Cai et al. 2012, 2014]. Among them, Tseng et al. [2011] is most closely related to ours. They studied the impact of power loss on flash memory chips, and they find, among other things, that flash memory operations do not necessarily suffer fewer bit errors when interrupted closer to completion. Such insightful observations are

helpful for SSD vendors to design more reliable SSD controllers. Indeed, we undertook this study as the obvious next step. However, unlike their work that is focused on the chip level, we study the device-level behavior of SSDs under power faults. Since modern SSDs employ various mechanisms to improve the device-level reliability, chip-level failures may be masked at the device level. Indeed, our study reveals that their simplest failure, bit corruption, rarely appears in full-scale devices.

Device-level studies are also abundant [Agrawal et al. 2008; Chen et al. 2009; Zhang et al. 2012; Lu et al. 2013; Jimenez et al. 2014; Li et al. 2015; Yadgar et al. 2015; Schroeder et al. 2016]. For example, Agrawal et al. [2008] build a trace-based SSD simulator based on DiskSim [Bucy et al.] and analyze a number of design tradeoffs for SSD performance, such as large allocation pool, overprovisioning, ganging, and rate-limiting wear-leveling. Chen et al. [2009] comprehensively evaluate the performance of SSDs using a set of I/O workloads with different access patterns and analyze the internal characteristics of SSDs and the impacts on systems and applications. While these studies provide valuable insights on designing SSDs-based storage systems, they mainly focus on the performance or lifetime of SSDs. Complementary to these studies, our design and analysis help understand the behavior of SSDs under hostile conditions like power faults.

Analysis of HDDs and HDD-based systems. In terms of traditional HDDs and HDD-based storage systems, Schroeder and Gibson [2007] analyze the disk replacement data of seven production systems over five years. They find that the field replacement rates of system disks were significantly larger than what the datasheet MTTFs suggest. In his study of redundant array of independent/inexpensive disks (RAID) arrays, Gibson [1990] proposes the metric *Mean Time To Data Loss* (MTTDL) as a more meaningful metric than MTTF; whether or not MTTDL is the ideal metric is somewhat disputed [Greenan et al. 2010]. Regardless, we report simply the number of failures we observed while purposefully faulting the drives; a separate study observing in-the-wild failures would also be interesting. Bairavasundaram et al. [2007, 2008] analyze the data corruption and latent sector errors in production systems containing a total of 1.53 million HDDs. Krioukov et al. [2008b] identify the parity pollution problem in HDD-based RAID systems and study the protection measures to avoid such problem. Rajimwale et al. [2011] observe that the write ordering cannot be trusted for HDDs and propose coerced cache eviction (CCE) to force writes to the disk surface despite an untrustworthy disk cache. Differing from these studies, we analyze the reliability of SSDs that have completely different internals. On the other hand, some existing protection mechanisms may also be applicable to SSDs. For example, CCE may eliminate certain unserializable writes caused by improper on-drive caching.

Analysis of general file and storage systems. Much effort has been put towards analyzing the reliability of file and storage systems and improving the robustness [Prabhakaran et al. 2005; Yang et al. 2006; Gunawi et al. 2008; Fryer et al. 2012; Chidambaram et al. 2012; Lu et al. 2013; Zheng et al. 2014; Pillai et al. 2014; Min et al. 2015; Chen et al. 2015]. For example, Prabhakaran et al. [2005] analyze the failure policies of four commodity file systems and propose the IRON file system, a robust file system with a family of recovery techniques implemented. EXPLODE [Yang et al. 2006] uses model checking to find errors in storage software, especially in file systems. Gunawi et al. [2008] study how error codes are propagated in file systems. Fryer et al. [2012] transform global consistency rules to local consistency invariants and provide fast runtime consistency checking to protect a file system from buggy operations. Chidambaram et al. [2012] introduce a backpointer-based No-Order File System to provide crash consistency. Zheng et al. [2014] and Pillai et al. [2014] study the crash consistency of storage applications (e.g., databases) and expose the inconsistent assumptions

about the underlying file systems. Chen et al. [2015] use Coq proof assistant to certify the FSCQ file system. Unlike these studies, our framework bypasses the file system and directly studies the block-level behavior of SSDs, which provides the foundation for designing robust storage software based on SSDs.

7. CONCLUSIONS AND FUTURE WORK

This article proposes a framework to automatically expose the bugs in block devices such as SSDs that are triggered by power faults. We apply effective workloads to stress the devices, devise a software-controlled circuit to actually cut the power to the devices, and check for various failures in the repowered devices. Based on our carefully designed record format, we are able to detect seven potential failure types. Our experimental results with 17 SSDs from six different vendors show that most of the SSDs we tested did not adhere strictly to the expected semantics of behavior under power faults. We have observed five out of the seven expected failure types, including bit corruption, shorn writes, unserializable writes, metadata corruption, and dead device. Our framework and experimental results should help design new robust storage system against power faults.

The block-level behavior of SSDs exposed in our experiments has important implications for the design of storage systems. For example, the frequency of both bit corruption and shorn writes make update-in-place to a sole copy of data that need to survive power failure inadvisable. Because many storage systems like file systems and databases rely on the correct order of operations to maintain consistency, serialization errors are particularly problematic. Write-ahead logging, for example, works only if a log record reaches persistent storage before the updated data record it describes. If this ordering is reversed or only the log record is dropped, then the database will likely contain incorrect data after recovery because of the inability to undo the partially completed transactions aborted by a power failure.

Our results on different OS kernels suggest that, besides block devices, OS kernels may also play a role in causing data loss. This is perhaps not surprising at all as kernels are complicated and may contain intricate bugs [Erickson et al. 2010; Fonseca et al. 2014]. Moreover, our experiments show that the kernel patch affects different devices in different ways, which suggests that there might be dependency between the implementations of some devices and kernels. Similarly to applications that make inconsistent assumptions about the underlying file systems [Zheng et al. 2014; Pillai et al. 2014], some devices may also make certain assumptions about the kernel behavior. In other words, there might be a semantic gap between devices and kernels that are developed by different groups of developers. With flash devices becoming more and more complicated internally, and more and more kernel components and libraries being optimized for flash aggressively [Bjørling et al. 2013; Marmol et al. 2014; Czerner and Zak 2014; Lu et al. 2014; Lee et al. 2015; Lu et al. 2016], the interaction and dependency between flash devices and the host-side software will likely become more complicated. How to pinpoint the root causes of storage failures in the context of the whole storage stack is thus a challenging and interesting topic to explore in the future.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their invaluable feedback.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, 57–70.

Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An analysis of data corruption in the storage stack. *Trans. Stor.* 4, 3, Article 8 (Nov. 2008), 28 pages. DOI: http://dx.doi.org/10.1145/1416944.1416947

Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY, 289–300. DOI: http://dx.doi.org/10.1145/1254882.1254917

H. P. Belgal, N. Righos, I. Kalastirsky, J. J. Peterson, R. Shiner, and N. Mielke. 2002. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th IEEE International Reliability Physics Symposium (IRPS'02)*.

Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. 2003. Introduction to flash memory. In *Procedings of the IEEE*. 489–502.

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

Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, New York, NY, Article 22, 10 pages.

A. Brand, K. Wu, S. Pan, and D. Chin. 1993. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st IEEE International Reliability Physics Symposium (IRPS'93)*.

John Bucy, Jiri Schindler, Steve Schlosser, and Greg Ganger. DiskSim v4.0. Retrieved from www.pdl.cmu.edu/DiskSim/.

Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. 2012. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*. EDA Consortium, 521–526.

Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. 2014. Neighbor-cell assisted error correction for MLC NAND flash memories. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM, New York, NY, 491–504. DOI: http://dx.doi.org/10.1145/2591971.2591994

Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS*.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 18–37. DOI: http://dx.doi.org/10.1145/2815400.2815402

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

Thomas Claburn. Amazon Web Services Hit By Power Outage. Retrieved from http://www.informationweek.com/cloud-computing/infrastructure/amazon-web-services-hit-by-power-outage/240002170.

Lukas Czerner and Karel Zak. 2014. FSTRIM in Linux. Retrieved from http://man7.org/linux/man-pages/man8/fstrim.8.html. (2014).

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 151–162.

Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 415–431.

Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*.

Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *ACM Comput. Surv.* 37, 2 (2005), 138–163.

Garth Gibson. 1990. *Redundant Disk Arrays: Reliable Parallel Secondary Storage*. Ph.D. Dissertation. University of California, Berkeley.

Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'11)*. ACM, New York, NY, 197–206. DOI: http://dx.doi.org/10.1145/1993806.1993834

Kevin M. Greenan, James S. Plank, and Jay J. Wylie. 2010. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'10)*. USENIX Association, Berkeley, CA, 5.

Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*.

Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. Berkeley, CA.

Christoph Hellwig. 2009. Kernel patch for v2.6.33-rc1. Retreived from http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ab0a9735e06914ce4d2a94ffa41497dbc142fe7f. (2009).

Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX, Berkeley, CA.47–59.

Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. 2008a. Parity lost and parity regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, Article 9, 15 pages.

Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. 2008b. Parity lost and parity regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, Article 9, 15 pages.

Andrew Ku. 2011. Second-Generation SandForce: It's All About Compression. Retrieved from http://www.tomshardware.com/review/vertex-3-sandforce-ssd,2869-3.html.

H. Kurata, K. Otsuga, A. Kotabe, S. Kajiyama, T. Osabe, Y. Sasago, S. Narumi, K. Tokami, S. Kamohara, and O. Tsuchiya. 2006. The impact of random telegraph signals on the scaling of multilevel flash memories. In *Symposium on VLSI Circuits (VLSI'06)*.

Anna Leach. Level 3's UPS burnout sends websites down in flames. Retrieved from http://www.theregister.co.uk/2012/07/10/data_centre_power_cut/.

Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 273–286.

Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.* 42, 6 (Oct. 2008), 36–42. DOI:http://dx.doi.org/10.1145/1453775.1453783

Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. 2015. How much can data compressibility help to improve NAND flash memory lifetime? In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 227–240.

Ren-Shou Liu, Chia-Lin Yang, and Wei Wu. 2012. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX, Berkeley, CA.

Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of linux file system evolution. In *Presented as Part of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX, Berkeley, CA, 31–44.

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, 133–148.

Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX, Berkeley, CA, 75–88.

Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX, Berkeley, CA, 257–270.

Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: A scalable and lightweight flash aware

key-value store. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX, Berkeley, CA.

Robert McMillan. 2012. Amazon Blames Generators for Blackout That Crushed Netflix. Retrieved from http://www.wired.com/wiredenterprise/2012/07/amazon_explains/.

Cade Metz. 2012. Flash Drives Replace Disks at Amazon, Facebook, Dropbox. Retrieved from http://www.wired.com/wiredenterprise/2012/06/flash-data-centers/all/.

Rich Miller. Human Error Cited in Hosting.com Outage. Retrieved from http://www.datacenterknowledge.com/archives/2012/07/28/human-error-cited-hosting-com-outage/.

Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 361–377. DOI:http://dx.doi.org/10.1145/2815400.2815422

T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, and S. Lai. 1993. Erratic erase in ETOX/sup TM/ flash memory array. In *Proceedings of the Symposium on VLSI Technology (VLSI'93)*.

Personal Communication. 2012. Personal communication with an employee of a major flash manufacturer.

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

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 206–220.

Minghai Qin, Eitan Yaakobi, and Paul H. Siegel. 2014. Constrained codes that mitigate inter-cell interference in read/write cycles for flash memories. *IEEE J. Select. Areas Commun.* 32, 5 (2014), 836–846.

Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. Coerced cache eviction and discreet mode journaling: Dealing with misbehaving disks. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN'11)*. IEEE Computer Society, Washington, DC, 518–529. DOI:http://dx.doi.org/10.1109/DSN.2011.5958264

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

Marco A. A. Sanvido, Frank R. Chu, Anand Kulkarni, and Robert Selinger. 2008. NAND flash memory and its role in storage architectures. In *Procedings of the IEEE*. 1864–1874.

Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*.

Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX, Berkeley, CA, 67–80.

Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. 1995. A 3.3V 32Mb NAND flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*.

K. Takeuchi, T. Tanaka, and T. Tanzawa. 1998. A multipage cell architecture for high-speed programming multilevel NAND flash memories. *IEEE Journal of Solid-State Circuits*.

Arie Tal. 2002. Two flash technologies compared: NOR vs NAND. White Paper of M-SYstems. M-Systems Flash Disk Pioneers, Ltd. https://focus.ti.com/pdfs/omap/diskonchipvsnor.pdf.

Veeresh Taranalli, Hironori Uchikawa, and Paul H. Siegel. 2015. Error analysis and inter-cell interference mitigation in multi-level cell flash memories. In *Proceedings of the 2015 IEEE International Conference on Communications (ICC'15)*. 271–276.

Nick Triantos. 2006. Lost Writes in Storage Systems. Retrieved from http://storagefoo.blogspot.com/2006/04/lost-writes.html.

Huang-Wei Tseng, Laura M. Grupp, and Steven Swanson. 2011. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)*.

Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. 2015. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *Proceedings of the13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, Berkeley, CA, Santa Clara, CA, 257–271.

Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 131–146.

Yiying Zhang, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*.

Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX, Berkeley, CA, 449–464.

Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDs under power fault. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, Berkeley, CA, 271–284.

Received January 2015; revised May 2016; accepted August 2016