Understanding the Robustness of SSDs under Power Fault

Mai Zheng[†] Joseph Tucek[*] Feng Qin [†] Mark Lillibridge[*]
[†] *The Ohio State University* [*] *HP Labs*

Abstract

Modern storage technology (SSDs, No-SQL databases, commoditized RAID hardware, etc.) bring 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 paper, 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 fifteen commodity SSDs from five different vendors using more than three thousand fault injection cycles in total. Our experimental results reveal that thirteen out of the fifteen tested SSD devices exhibit surprising failure behaviors under power faults, including bit corruption, shorn writes, unserializable writes, metadata corruption, and total device failure.

1 Introduction

Compared to traditional spinning disk, flash-based solid-state disks (SSDs) offer much greater performance and lower power draw. Hence SSDs are already displacing spinning disk in many datacenters [17]. However, while we have over 50 years of collected wisdom working with spinning disk, flash-based SSDs are relatively new [1], and not nearly as well understood. Specifically, the behavior of flash memory in adverse conditions has only been studied at a component level [26]; given the 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 paper considers the behavior of SSDs[1] 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 [18, 13, 16, 4] 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, 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 15 SSDs from 5 different vendors to more than three thousand fault injection cycles in total. Surprisingly, we find that *13 out of the 15* 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, two of the fifteen 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 [27] 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 OS, device driver, bus controller, or the block device itself have an opportunity to perform a clean shutdown.

[1] In this paper, "SSD" means "flash-based SSD"

Originally published in the 11th USENIX conference on File and Storage Technologies (FAST'13)

- **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, flying writes, and shorn writes) 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 fifteen different SSDs and two hard drives.** Using our implementation of the proposed testing framework, we have evaluated the failure modes of fifteen commodity SSDs as well as two traditional spinning-platter hard drives for comparison. Our experimental results show that SSDs have counter-intuitive behavior under power fault: of the tested devices, only two SSDs (of the same model) 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.

SSDs offer the promise of vastly higher performance operation; our results show that they 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 [25]. 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/three bit "triple-level cell" (TLC).

In terms of higher-level characteristics, MLC flash is more complex, slower, and less reliable comparing to SLC. A common trick to improve the performance of MLC flash is to consider the two bits in each physical cell to be from separate logical pages [24]. This trick is nearly universally used by all MLC flash vendors [19]. However, since writing[2] to a flash cell is typically a complex, iterative process [15], 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 2 MB) that can only be zero-ed all together. A page is a physically contiguous set of cells (typically 4 KB) 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 (4 KB) 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 [11, 15, 10, 22] 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 floating gate, 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.

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 a Flash Translation Layer (FTL), to make the device appear as if it can do update-in-place. The mechanisms for this are typically reminiscent of a journaling filesystem [21]. The specifics of a particular

[2] 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.

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

Table 1: Brief description of the types of failures we attempted to detect.

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 [14]. Some FTLs are quite sophisticated, and will even compress data in order to reduce the amount of wear imposed on the underlying flash [12].

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 [19]. 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 [26]. Erase operations are also susceptible to power loss, since they take much longer to complete than program operations [2]. The result of corruption at the cell-level is typically an 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 [19].

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 1 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 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. We also expected to see operations about a device's page size "shear" (e.g., we expected to see only 4 KB of an 8 KB update), but at that level the behavior of the drive is still correct. Because an FTL is a complex piece of software, and corruption of its internal state could be problematic, we expected to see metadata corruption and dead devices. Finally, 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.

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 faults (e.g., bit corruption, flying writes, etc.) 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 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

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

Table 2: Description of header fields and why they are included.

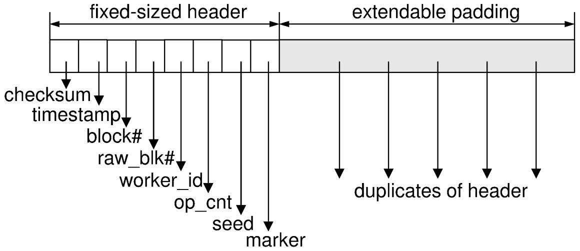

Figure 1: Record format.

that header to pad the record out to the desired length. Table 2 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 that 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 we can 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 4 KB. 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 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, it would be undetectable. Another approach is to fill the padding of the record with random numbers. This in-

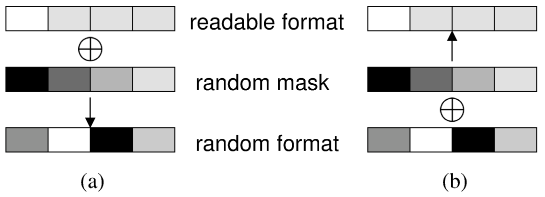

Figure 2: Randomization of record format. (a) Generate the random format by XORing the readable format with the random mask. (b) Recover from the random format by XORing it with the random mask again.

deed 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 it 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.

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 has 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).

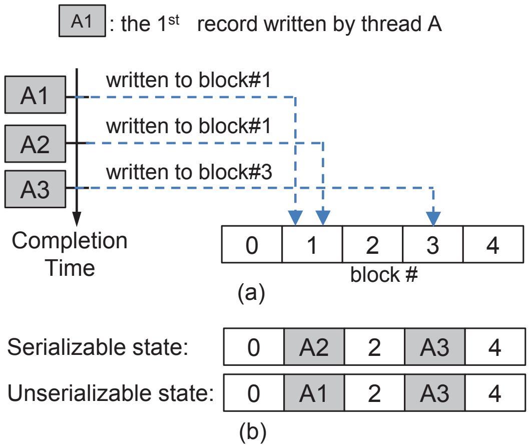

Figure 3: Serializability of writes within one thread. (a) Thread A synchronously writes three records with both the 1st and 2nd records written to the same address (block#1). (b) In a serializable state, the on-device records reflect the order of synchronous writes; that is, later records (e.g., $A2$)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

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

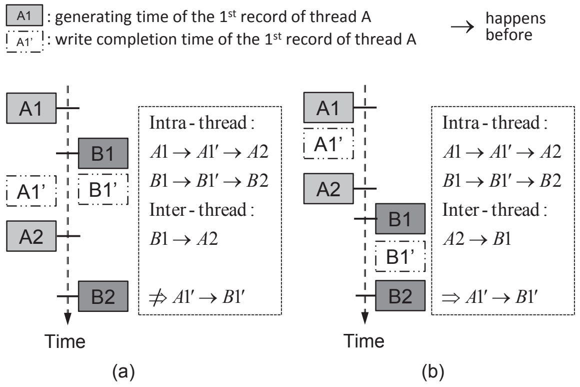
A1 : generating time of the 1st record of thread A
A1' : write completion time of the 1st record of thread A
→ happens before
A1
B1
A1'
B1'
A2
B2
Time
Intra - thread :
$A1 \rightarrow A1' \rightarrow A2$
$B1 \rightarrow B1' \rightarrow B2$
Inter - thread :
$B1 \rightarrow A2$
$\nRightarrow A1' \rightarrow B1'$
(a)
A1
A1'
A2
B1
B1'
B2
Time
Intra - thread :
$A1 \rightarrow A1' \rightarrow A2$
$B1 \rightarrow B1' \rightarrow B2$
Inter - thread :
$A2 \rightarrow B1$
$\Rightarrow A1' \rightarrow B1'$
(b)

Figure 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 $B1'$.

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*.

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 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 (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, 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.* [8]). Given the information 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.

Our algorithm works roughly as follows: the record format allows us to reconstruct the full operation stream (who wrote what where). 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 locations to which they wrote. We expect to see the result of one 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". We count instances of the third case as serialization errors. The actual algorithm is somewhat more complicated; we omit the details and correctness argument for space reasons.

3.3 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.

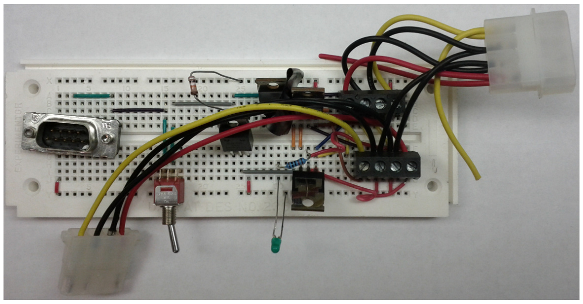

Figure 5: The hardware used to cut power in our experiments

3.4 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. Figure 5 shows a photograph of the control circuit. This separation of power supplies enables us to test the target device intensively without jeopardizing other circuits in the host systems. 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., 30 seconds). 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.

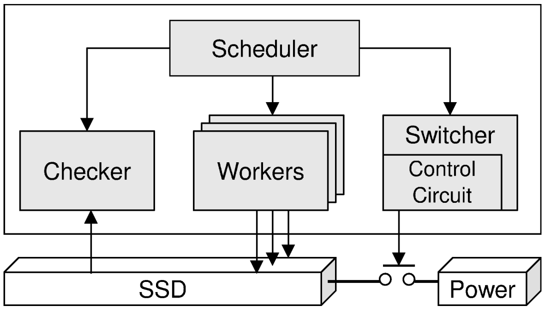

Figure 6: Framework for testing SSDs using injected power faults.

3.5 Putting It Together

This section briefly summarizes the work flow of our proposed framework. As shown in Figure 6, 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., 30 seconds). Meanwhile, the Switcher waits for a random amount of time within the working period (e.g., 8 seconds) 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 when the SSD is being actively exercise by a workload. Of course, the 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.

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
HDD#1	F	0.33	5.4K	’08	-
HDD#2	G	1.64	15K	-	-

Table 3: 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 RPM of the disk. ”P” indicates presence of some power-loss protection (e.g. a super capacitor).

4 Experimental Environment

4.1 Block Devices

Out of the wide variety of SSDs available today, we selected fifteen representative SSDs (ten different models) from five different vendors.[3]The characteristics of these SSDs are summarized in Table 3. More specifically, the prices of the 15 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 multi-level cell (MLC) and single-level cell (SLC). Based on our examination and manufacturer’s statements, four 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 3.00 GHz CPU and 2 GB of main memory. The operating system is Debian Linux 6.0 with Kernel 2.6.32. The SSDs and the hard drives are individually connected to a LSI Logic SAS1064 PCI-X Fusion-MPT

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

Device ID	SSD#1	SSD#2	SSD#3	SSD#4	SSD#5	SSD#6	SSD#7	SSD#8	SSD#9
# of Faults	136	1360	8	186	105	105	250	200	200
Device ID	SSD#10	SSD#11	SSD#12	SSD#13	SSD#14	SSD#15	HDD#1	HDD#2	-
# of Faults	100	151	100	100	233	103	1	24	-

Table 4: Number of faults applied to each device during concurrent random writes workload.

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, HDD#1
Metadata Corruption	**Y**	SSD#3
Dead Device	**Y**	SSD#1
None	**Y**	SSD#6, SSD#10, HDD#2

Table 5: Summary of observations. "Y" means the failure was observed with any device, while "N" means the failure was not observed.

SAS Controller. The power supply to the SSDs is controlled by our custom circuit, which is connected to the host machine via its serial port.

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.

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 4 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 5.5 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.

5.1 Overall Results

Table 5 shows the summarized results. In our experiments, we observed five out of the six expected failure types, including bit corruption, shorn writes, unserializable writes, metadata corruption, and dead device. Surprisingly, we found that *13 out of 15* 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 (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

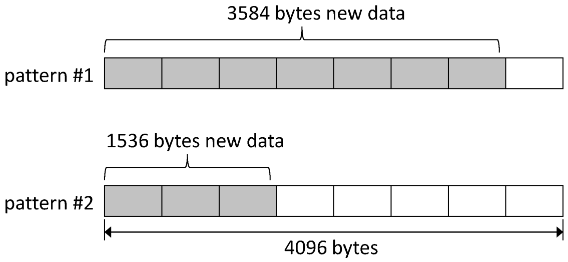

Figure 7: Two examples of shorn writes observed.

third of the device becoming inaccessible. As for SSD#1, all of its data was 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 observed failure and provide our analysis on possible reasons behind the observations. The only failure we did not observe is flying writes, which indicates that the FTLs are capable of keeping a consistent mapping between logical and physical addresses.

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 [26] 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 [19]. However, this could lead to unserializable writes.

5.3 Shorn Writes

In our experiments, we use 4 KB as the the default record size, which is the typical block size used in the kernel. We did not expect a shorn write within a 4 KB record should or could occur for two reasons: First, 4 KB is the common block size used by the kernel as well as the page size used in the SSDs (which can be protected by the associated ECC). In other words, most drives use a 4 KB programming unit, which would not allow shorn writes to occur within a 4 KB block. Second, SSDs usually serve a write request by sending it to a newly erased block whose location is different from the location that stores the old data so a partially new and partially old record is unlikely to appear.

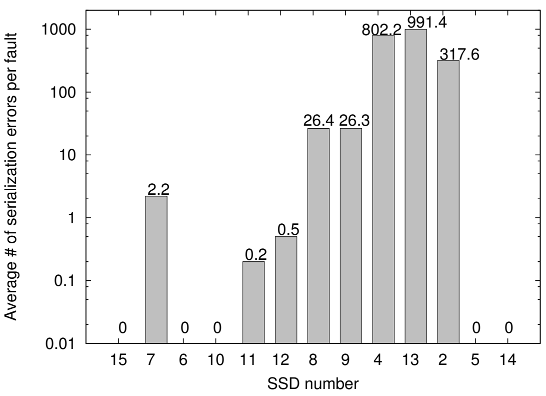

Figure 8: Average number of serialization errors observed per power fault for each SSD. The x-axis shows the devices from cheapest unit price ($/GB) to most expensive.

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 7 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 1536 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 subpage programming technique internally that treats 512 bytes as a programming unit, contrary to manufacturer claims. Also, a 4 KB logical record is mapped to multiple 512-byte physical pages. As a result, a 4 KB record could be partially updated while keeping the other part unchanged.

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.

5.4 Unserializable Writes

We have detected unserializable writes in eight tested SSDs, including no. 2, 4, 7, 8, 9, 11, 12, and 13. Figure 8 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 misordered. The SSDs are sorted in the increasing order of

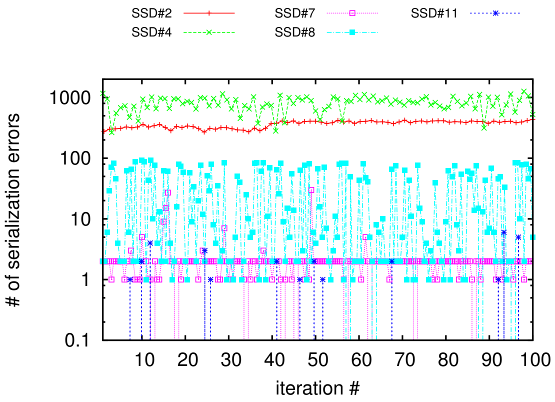

Figure 9: Number of serialization errors observed in each testing iteration for selected SSDs.

their cost per capacity ($/GB) from left to right. No 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.

Figure 9 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 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 very 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 counter-intuitive 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: 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 [6].

5.5 Metadata Corruption

SSD#3 exhibited an interesting behavior after a small number (8) of tests. SSD#3 has 256 GB of flash memory visible to users, which can store 62,514,774 records in our experimental setting. However, after 8 injected power faults, only 69.5% of all the records can be retrieved from the device. In other words, 30.5% of the data (72.6 GB) was suddenly lost. When we try 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 as 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.

5.6 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 was 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 either an unrecoverable loss of metadata (e.g., the whole mapping table is corrupted), 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.

5.7 Comparison with Hard Drives

For comparison purposes, we also evaluated the reliability of two hard drives under power faults. Hard drive I/O operations are much slower than that of SSDs. Accordingly, due to limited time, we applied only a few tests to HDD#1 and HDD#2.

Our testing framework detects 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.

6 Related Work

Characteristic studies of flash devices. Many studies have been conducted on different aspects of flash memory devices [10, 26, 11]. Among them, Tseng *et al.* [26] is most closely related to ours. Tseng *et al.* studied the impact of power loss on flash memory chips. 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.

Bug detection in storage systems. EXPLODE uses model checking to find errors in storage systems [27]. It systematically explores every possible execution path of the systems in order to detect subtle bugs. In each possible state of the execution, EXPLODE emulates the effect of a crash by committing a subset of the dirty blocks onto disk, running the system's recovery code, and then checking the consistency of the recovered system. Our work has similar goals as EXPLODE, but we focus on block devices rather than file systems. Additionally, we focus on power faults instead of non-fail–stop corruption errors. Furthermore, EXPLODE uses RAM disks as the lowest storage component in experiments, while we study the behavior of real SSD hardware.

Consistency checking of file systems. Much effort has been put towards analyzing the consistency of file systems and designing robust file systems. Prabhakaran *et al.* [20] 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. Fryer *et al.* [5] 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.* [3] introduce a backpointer-based No-Order File System (NoFS) to provide crash consistency. Unlike these studies, our framework bypasses the file system and directly tests the block-level behavior of SSDs. Further, these studies all look to design file systems that are robust to failure while we look to determine what failures actually occur. It would be interesting to test each of these filesystems under our testing framework.

Reliability analysis of mechanical hard disks. Schroeder *et al.* [23] 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 RAID arrays, Gibson [7] 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 [9]. 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.

7 Conclusions

This paper proposes a methodology 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 six potential failure types. Our experimental results with fifteen SSDs from five different vendors show that most of the SSDs we tested did not adhere strictly to the expected semantics of behavior under power faults. We observed five out of the six 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 needs to survive power failure inadvisable. Because many storage systems like filesystems 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.

Because we do not know how to build durable systems that can withstand all of these kinds of failures, we recommend system builders either not use SSDs for important information that needs to be durable or that they test their actual SSD models carefully under actual power failures beforehand. Failure to do so risks massive data loss.

8 Acknowledgments

The authors would like to thank their shepherd, John Strunk, and the anonymous reviewers for their invaluable feedback. This research is partially supported by NSF grants #CCF-0953759 (CAREER Award) and #CCF-1218358.

References

[1] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to Flash Memory. In *Procedings of the IEEE*, pages 489–502, April 2003.

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

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

[4] Thomas Claburn. Amazon web services hit by power outage. http://www.informationweek.com/cloud-computing/infrastructure/amazon-web-services-hit-by-power-outage/240002170.

[5] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, San Jose, California, February 2012.

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

[7] Garth Gibson. *Redundant Disk Arrays: Reliable Parallel Secondary Storage*. PhD thesis, University of California, Berkeley, December 1990.

[8] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 197–206, New York, NY, USA, 2011. ACM.

[9] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. 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, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

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

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

[12] Andrew Ku. Second-generation SandForce: It's all about compression. http://www.tomshardware.

com/review/vertex-3-sandforce-ssd,2869-3.html, February 2011.

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

[14] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, October 2008.

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

[16] Robert McMillan. Amazon blames generators for blackout that crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazon_explains/.

[17] Cade Metz. Flash drives replace disks at Amazon, Facebook, Dropbox. http://www.wired.com/wiredenterprise/2012/06/flash-data-centers/all/, June 2012.

[18] Rich Miller. Human error cited in hosting.com outage. http://www.datacenterknowledge.com/archives/2012/07/28/human-error-cited-hosting-com-outage/.

[19] Personal communication with an employee of a major flash manufacturer, August 2012.

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

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

[22] Marco A. A. Sanvido, Frank R. Chu, Anand Kulkarni, and Robert Selinger. NAND Flash Memory and Its Role in Storage Architectures. In *Procedings of the IEEE*, pages 1864–1874, November 2008.

[23] Bianca Schroeder and Garth A. Gibson. 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)*, 2007.

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

[25] Arie Tal. Two flash technologies compared: NOR vs NAND. In *White Paper of M-SYstems*, 2002.

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

[27] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, November 2006.