Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy

Amy Tai, ***Princeton University and VMware Research;*** **Andrew Kryczka and Shobhit O. Kanaujia,** ***Facebook;*** **Kyle Jamieson and Michael J. Freedman,** ***Princeton University;*** **Asaf Cidon,** ***Columbia University***

https://www.usenix.org/conference/atc19/presentation/tai

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy

Amy Tai
Princeton University and VMware Research

Andrew Kryczka
Facebook Inc.

Shobhit O. Kanaujia
Facebook Inc.

Kyle Jamieson
Princeton University

Michael J. Freedman
Princeton University

Asaf Cidon
Columbia University

Abstract

Due to its high performance and decreasing cost per bit, flash storage is the main storage medium in datacenters for hot data. However, flash endurance is a perpetual problem, and due to technology trends, subsequent generations of flash devices exhibit progressively shorter lifetimes before they experience uncorrectable bit errors. In this paper, we present an approach for addressing the flash lifetime problem by allowing devices to operate at much higher bit error rates. We present DIRECT, a set of techniques that harnesses distributed-level redundancy to enable the adoption of new generations of denser and less reliable flash storage technologies. DIRECT does so by using an end-to-end approach to increase the reliability of distributed storage systems.

We implemented DIRECT on two real-world storage systems: ZippyDB, a distributed key-value store in production at Facebook that is backed by and supports transactions on top of RocksDB, and HDFS, a distributed file system. When tested on production traces at Facebook, DIRECT reduces application-visible error rates in ZippyDB by more than 100× and recovery time by more than 10,000×. DIRECT also allows HDFS to tolerate a 10,000–100,000× higher bit error rate without experiencing application-visible errors. By significantly increasing the availability of distributed storage systems in the face of bit errors, DIRECT helps extend flash lifetimes.

1 Introduction

Flash has become the dominant storage medium for hot data in datacenters [64, 72], since it offers significantly lower latency and higher throughput than hard disks. Many storage systems are built atop flash, including databases [6, 11, 15, 44], caches [5, 36, 57, 58, 78], and file systems [48, 67].

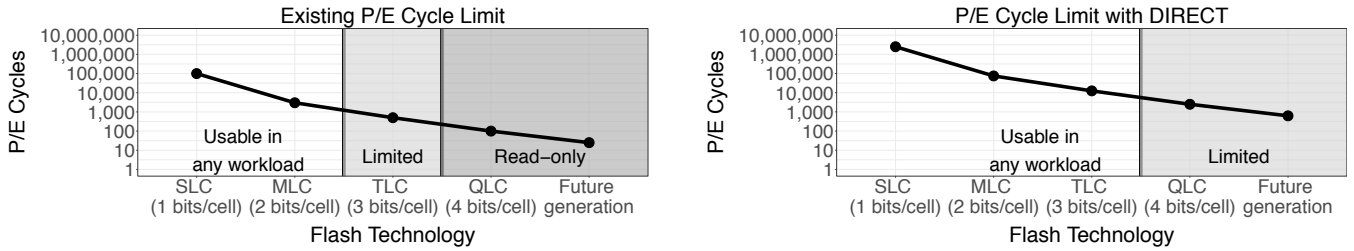
However, a perennial problem of flash is its limited endurance, or how long it can reliably correct raw bit errors. As device writes are the main contributor to flash wear, its lifetime is measured in the number of writes or program-erase (P/E) cycles the device can tolerate before exceeding an uncorrectable bit error threshold. Uncorrectable bit errors are errors that are exposed externally and occur when there are too many raw bit errors for the device to correct.

In hyper-scale datacenters, operators constantly seek to reduce flash wear by limiting flash writes [21, 64]. At Facebook, for example, a dedicated team monitors application writes to ensure they do not prematurely exceed manufacturer-defined device lifetimes. Even worse, each subsequent flash generation tolerates a smaller number of writes before reaching end-of-life (see Figure 1a) [42]. Further, given the scaling challenges of DRAM [49, 56] and the increasing cost gap between DRAM and flash [2, 37, 38], many operators are migrating services from DRAM to flash [7, 37], increasing the pressure on flash lifetime.

There is a variety of work that attempts to extend flash lifetime by delaying the onset of bit errors [6, 12, 30, 36, 47, 59, 61, 62, 63, 77, 82, 83]. This paper takes a contrarian approach. We observe that flash endurance can be extended by *allowing* devices to go beyond their advertised uncorrectable bit error rate (UBER) and embracing the use of flash disks that exhibit much higher error rates; Google recently released a whitepaper suggesting a similar approach [28]. We can do so without sacrificing durability because datacenter storage systems replicate data on remote servers, and this redundancy can correct bit error rates orders of magnitude beyond the hardware error correction mechanisms implemented on the device. However, the challenge with higher flash error rates is maintaining availability and correctness.

We introduce Distributed error Isolation and RECovery Techniques (DIRECT), which is a set of three simple general-purpose techniques that, when implemented, enable distributed storage systems to achieve high availability and correctness in the face of uncorrectable bit errors:

1. **Minimize data error amplification.** DIRECT detects errors using existing error detection mechanisms (e.g., checksums) and recovers data from remote servers at the smallest possible granularity.
2. **Minimize metadata error amplification.** A corruption in local metadata (e.g., database index), often requires a large amount of data to be re-replicated. DIRECT avoids



(a) Existing hardware-based error correction.

(b) Augmenting existing error correction with DIRECT.

Figure 1: For each generation of flash bit density, the average number of P/E cycles after which the uncorrectable bit error rate falls below the manufacturer specified level (10^{-15}). Beyond MLC, the number of flash writes the application can issue is limited [29]. With current hardware-based error correction, QLC technology and beyond can only be used for applications that are effectively read-only [8, 22, 76]. DIRECT enables the adoption of denser flash technologies by handling errors in the distributed storage application. We use the model from §3 to compute the UBER tolerated by DIRECT, while the UBER to P/E conversion was derived from data in a Google study [72].

this by adding redundancy locally to local metadata.

3. **Ensure safe recovery semantics** by treating recovery operations as write operations. DIRECT serializes recovery operations on corrupted data against concurrent operations with respect to the system's consistency guarantees.

The difficulty of implementing DIRECT depends on two properties of the underlying storage system. The first property is whether the system is physically or logically replicated. Physically-replicated systems replicate *data blocks* between servers, while logically-replicated systems replicate the *commands* (e.g., write, update, delete). In physically-replicated systems, a certain object is stored in the same block or file on another server and therefore can be recovered efficiently by simply re-replicating the remote data block. This does not work for logically-replicated systems, where physical blocks are not identical across replicas. The second property is whether the data store supports versioning. In systems with versioning, we need to guarantee the recovered object does not override a more up-to-date version.

We demonstrate how to generalize DIRECT techniques by implementing them in two popular systems that are representative of two different classes of storage systems: (1) the Hadoop Distributed File System (HDFS), which is a physically-replicated storage system without versioning, and (2) ZippyDB, a distributed system that implements logical replication and transactions on top of RocksDB, a popular key-value store that supports key versioning. Objects in HDFS are physically-replicated, so it is straightforward for DIRECT to find the corrupt object in another replica and recover it at a high granularity (§4.1). On the other hand, recovery is challenging in ZippyDB since the corrupted region of one replica is stored in a different location on another replica, so the recovered key-value pairs might not have consistent versions ZippyDB (§4.2).

DIRECT Limitations and Lessons Learned. We chose to implement DIRECT by retrofitting existing datacenter storage applications, rather than as a general-purpose application library. The latter design would be particularly challenging since DIRECT depends on application-specific details such as file layout and recovery semantics. Note that the storage systems we retrofitted (HDFS and RocksDB) and their relatives serve as the base layer for many storage services and databases (e.g., MyRocks, Ozone, HBase, Cassandra). Furthermore, we learned that to implement the first and third DIRECT techniques, storage systems must have a key requirement: we must be able to infer logical objects from the physical location (on the application's file format) of the bit error. In §6 we discuss PostgreSQL, which does not satisfy this requirement, and therefore is difficult to retrofit with DIRECT.

DIRECT leads to significant increases in device lifetime, since systems can maintain the same probability of application-visible errors (durability) at much higher device UBERs. In Figure 1b, we estimate the number of P/E cycles gained with DIRECT using an empirical UBER vs P/E cycle comparisons in a Google study [72]. Depending on workload parameters and hardware specifications, DIRECT can increase the lifetime of devices by 10-100×. This allows datacenter operators to replace flash devices less often and adopt lower cost-per-bit flash technologies that have lower endurance. DIRECT also provides the opportunity to rethink the design of existing flash-based storage systems, by removing the assumption that the device fixes all corruption errors. Furthermore, while this paper focuses on flash, DIRECT's principles also apply in other storage mediums, including NVM, hard disks, and DRAM.

In summary, this paper makes several contributions:

1. We observe flash lifetime can be extended by allowing devices to operate at much higher bit error rates.
2. We propose DIRECT, general software techniques that enable storage systems to maintain performance and high availabily despite high hardware bit error rates.
3. We design and implement DIRECT in two storage systems, HDFS and ZippyDB, that are representative of physical and logical replication, respectively. Applying DIRECT results in significant end-to-end availability improvements: it enables HDFS to tolerate bit error rates that are 10,000×-100,000× greater, reduces application-

visible error rates in ZippyDB by more than 100×, and speeds up recovery time in ZippyDB by 10,000×.

2 Motivation

What Limits Flash Endurance? Flash chips are composed of memory cells, each of which stores an analog voltage value. The flash controller reads the value stored in a certain memory cell by sensing the voltage level of the cell and applying quantization to determine the discrete value in bits. The more bits stored in a cell, the narrower the voltage range that maps to each discrete bit, so more precise voltage sensing is required to get a correct read. A primary way to reduce cost per bit is to increase the number of bits per cell, which means that even small voltage perturbations can result in a misread.

Multiple factors cause voltage drift in a flash cell. The dominant source, especially in datacenter settings where most data is "hot," is the program-erase (P/E) cycle, which involves applying a large high voltage to the cell in order to drain its stored charge, thus wearing the insulating layer in the flash cell [30]. This increases the voltage drift of subsequent values in the cell, which gradually leads to bit errors.

3D NAND is a recent technology that has been adopted for further increasing flash density by stacking cells vertically. While 3D NAND relaxes physical limitations of 2D NAND (traditional flash) by enabling vertical stacking, 3D NAND inherits the reliability problems of 2D NAND and further exacerbates them, since a cell in 3D NAND has more adjacent (vertical) neighbors. For example, voltage retention is worse, because voltage can now leak in three dimensions [54, 63, 65]. Similarly, disturb errors that occur when adjacent cells are read or programmed are also exacerbated [50, 75].

Existing Hardware Reliability Mechanisms and Limitations. To correct bit errors, flash devices use error correcting codes (ECC), which are implemented in hardware. After the ECC pass, there could still be incorrect bits on the page. To address this, SSDs also employ internal RAID across the dies of a flash device [16, 19]. After applying coding and RAID within the device, there will remain a certain rate of *uncorrectable bit errors* (UBER). Together, ECC and internal RAID mechanisms can drive the error rates of SSDs from the raw bit error rate of around 10^{-6} down to the 10^{-17} to 10^{-20} UBER range typical of enterprise SSDs [14]. "Commodity" SSD devices typically guarantee an UBER of 10^{-15}.

While it is possible to create stronger ECC engines, the higher the corrective power of the ECC, the more costly the device [4, 10]. Furthermore, the level of internal RAID striping is constant across generations, because the number of dies inside a flash device remains constant. This means that the corrective power of RAID is fixed.

Similarly, while RAID across devices [53, 69, 74] can add redundancy, a main design goal of DIRECT is to avoid adding unnecessary overhead. We avoid turning to RAID because it is inflexible since its recovery power is fixed at deployment time, and, more importantly, it imposes storage and write overheads, in particular generating additional flash writes that further reduce endurance.

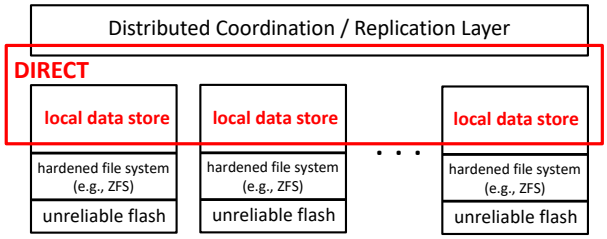


Figure 2: DIRECT fixes errors in the local data store, sometimes requiring interaction with the distributed coordination layer.

Implications of Limited Flash Endurance. Flash technology has already reached the point where its endurance is inhibiting adoption and operation in various datacenter use cases. For example, QLC was recently introduced as the next generation flash cell technology. However, in the worst case, it can only tolerate ∼150 P/E cycles [8, 22, 76], so it can only be used for read-heavy use cases, e.g., a 2 TB QLC drive with a lifetime of 150 P/E cycles can only write at a rate of 2 MB/s or less in order to preserve its advertised lifetime of 5 years. In the best case, some QLC devices can tolerate ∼1000 P/E cycles for completely sequential write workloads, due to an internal SLC cache [8]. But since datacenter applications like databases and analytics that deal with hot data typically need to update objects frequently, the adoption of QLC has been more limited and is the reason that Facebook has avoided QLC flash. Subsequent cell technology generations will suffer from even greater problems.

Operational issues also often dictate a device's usage lifetime. Flash is typically only used for its advertised lifetime to simplify operational complexity [72]. Further, in a hyperscale datacenter where it is common to source devices from multiple vendors, the most conservative estimate of device lifetime across vendors is typically chosen as the lifetime for a fleet of flash devices, so that the entire fleet can be installed and removed together. If the distributed storage layer could tolerate much higher device error rates, then datacenter operators would no longer have to make conservative and wasteful estimates about entire fleets of flash devices.

Finally, because of the increase in DRAM prices due to its scaling challenges and tight supply [2, 38, 49, 56], datacenter operators are migrating services from DRAM to flash [7, 37]. This means flash will be responsible for many more workloads, further exacerbating its endurance problem.

3 DIRECT Design

DIRECT is a set of techniques that enables a distributed storage system to maintain high availability and correctness in the face of high UBER.

We define a distributed storage system as a set of many local stores coupled with a distributed protocol layer that replicates data and coordinates between the local stores. Figure 2

shows an ideal storage stack that runs on unreliable flash (flash that exposes high UBERs). Note that there is existing work on how to make local file systems tolerate corruption errors (we survey these in §6), so our efforts in this paper focus on hardening the application-level storage system.

We observe that redundancy already exists in distributed storage systems in the form of distributed replication [27, 32, 40], which maintains multiple remote copies of each piece of data, or distributed erasure coding [45, 71, 80], which maintains remote parity bits for each piece of data. However, many systems do not systematically use this redundancy to recover individual bit errors [39], even though it can significantly boost resilience to bit errors. We focus on using distributed replicas to correct bit errors, but the DIRECT principles presented in the paper also apply to systems that use erasure coding; for example, error amplification — the number of bits required to recover an error — can be reduced in erasure coding schemes by reducing the size of a stripe.

Distributed Redundancy. Consider the following example of a physically replicated storage system, such as HDFS. Suppose the minimum unit of recovery is a data block[1], which is replicated in each of three data stores. If the block has size B, and the uncorrectable bit error rate (UBER) is E, then the expected number of errors in the block will be $B \cdot E$. Since the block is replicated across R different servers, the only way that the storage system would encounter an application-observable read error is when at least one error exists in *every* copy of the block. Therefore, the probability of an application-level read error can be expressed as:

$$\mathbb{P}[error] = (1-(1-E)^B)^R \approx (B \cdot E)^R$$

where we assume $B \cdot E << 1$ and use a Taylor series approximation.

Then, for an UBER of $E = 10^{-15}$, a block size of $B = 128\ MB$ (typical of distributed file systems), and a replication factor of $R = 3$, the probability of a read error is 10^{-18} (files are measured in bytes, while UBER is in bits).

However, with relatively large blocks, the probability of encountering at least one error in all block replicas quickly increases as UBER increases. For example, for an UBER of $E = 10^{-10}$, the expected number of errors in a single block will be $B \cdot E = 0.1$ for 128 MB blocks (Table 1). Then in this case $\mathbb{P}[error] \approx 0.001$. We observe that reducing $B \cdot E$, by reducing B, will dramatically reduce the probability of error.

Minimizing Error Amplification of Data Blocks. DIRECT captures this intuition with the notion of *error amplification* (B in the previous example), or the number of bytes required to recover a bit error. DIRECT observes that *the lower the error amplification, the lower the probability of error and the faster recovery can occur*. This similarly implies a shorter period of time spent in degraded durability and thus higher availability.

[1]Note that in HDFS while errors can be detected using checksums at a smaller granularity than the block size, actual recovery and replication is conducted at the granularity of a block.

	Probability of Application-Observable Error	
UBER	**Block Recovery**	**Chunk Recovery**
10^{-10}	$1 \cdot 10^{-3}$	$3 \cdot 10^{-10}$
10^{-15}	$1 \cdot 10^{-18}$	$1 \cdot 10^{-28}$

Table 1: Probability of application-observable error comparing block-by-block recovery to chunk-by-chunk recovery, with an UBER of 10^{-10}, and 10^{-15}. Finer granularity recovery provides significantly higher protection against corruptions.

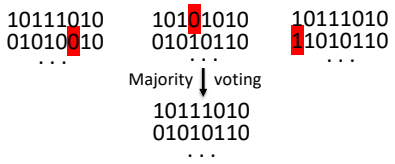


Figure 3: Even if the same chunk is corrupted on all replicas, bit-by-bit majority voting can reconstruct the correct chunk, by taking the majority vote of each bit across all chunks.

In the example above, suppose the system can recover data at a finer granularity, for example, at chunk size $C = 64$ KB. Then a read error would occur if all three replicas of the same *chunk* have at least one bit error. The revised probability of read error becomes:

$$\mathbb{P}[error] = 1-(1-(1-(1-E)^C)^R)^{\frac{B}{C}}$$

Assuming $E \cdot C << 1$, Taylor series approximation leads to $(1-(1-E)^C)^R) \approx (E \cdot C)^R$, and assuming this value is much smaller than $\frac{B}{C}$, the probability of an application-observable error when correcting chunk-by-chunk is:

$$\mathbb{P}[error] \approx (E \cdot C)^R \cdot \frac{B}{C}$$

When $C = 64$ KB and $E = 10^{-10}$, this probability is $3 \cdot 10^{-10}$, which is much lower than the probability when recovering at the block level (see Table 1).

We can further reduce B by using bit-by-bit majority voting, i.e., the recovered value of a bit in the chunk is the majority vote across the three chunk replicas (Figure 3). Bit-by-bit majority voting further reduces the application-observable error beyond chunk-by-chunk recovery, because the only way an application-observable error would occur is if an error occurs in the same bit across two chunks or more.

In a physically-replicated system like HDFS, minimizing error amplification is straightforward because corrupted blocks (and even bits) can be directly recovered from remote replicas. For a logically-replicated system like ZippyDB, however, blocks are not identical across replicas. This makes minimizing error amplification more challenging, since DIRECT cannot simply recover from a remote physical chunk. For example, bit-by-bit majority voting is not possible in ZippyDB, because the replicas do not store the same physical bits. For such systems, DIRECT must instead first isolate the region where the error might have occurred and then retrieve objects one-by-one from the other servers (see §4.2).

Minimizing Metadata Error Amplification. Error amplification can be even more severe if the error occurs in local metadata. For example, a corrupt index in a key-value store can prevent a data store from starting up, which can mean re-replication of hundreds of GBs of data. Thus even though the likelihood of errors in metadata is lower than in data blocks (metadata typically takes up less space than data), it still requires protection. Hence DIRECT either locally duplicates metadata or applies local software error correction.

Safe Recovery Semantics. DIRECT must also ensure recovery operations preserve the correctness of the distributed storage system, which might be dealing with concurrent write and read operations.

This is relatively straightforward in systems that do not support versioning or transactions, such as HDFS, since an object is up-to-date as soon as it is recovered from a remote replica. Systems like RocksDB which support versioning are more challenging, because if the system re-writes an object from a remote replica, recovery might overwrite a newer version with a stale version. In particular, the versions of the corrupted key-value pairs *are not known*, because (a) the corruption prevents the data from being read and (b) due to logical replication, the data's location does not provide information on its version. Hence to correctly recover corrupted key-value pairs, the system must locate some consistent (up-to-date) version of each pair. To do this, DIRECT forces recovery operations to go through a fault-tolerant log (for ZippyDB we use its existing Paxos log), which can provide correct ordering (§4.2.3).

DIRECT Techniques. To summarize, DIRECT includes the following techniques.

1. Systems must reduce error amplification of data objects and fix corruptions from remote replicas.
2. Systems must reduce local metadata error amplification, which is much higher than data error amplification.
3. Systems must ensure safe recovery semantics.

Note that the first and second techniques apply exclusively to the local data store and affect *performance*, while the third technique may require that the local data store interact with the distributed coordination layer to ensure *correctness*.

4 Implementing DIRECT

To demonstrate the use of the DIRECT approach, we integrate it into two systems: HDFS, a popular distributed file system, and ZippyDB, a distributed key-value store backed by RocksDB. The techniques used to implement DIRECT in HDFS can be applied to other physically replicated systems, such as GFS [40], Windows Azure Storage [31], and RAMCloud [66], which write objects into large immutable blocks that are replicated across several servers. Similarly, the techniques used to implement DIRECT in ZippyDB and RocksDB can be applied to other logically replicated systems, such as Cassandra [79], MongoDB [9], and CockroachDB [1]. In these systems a distributed coordination layer manages the replication of objects across different servers and uses versioning to execute transactions.

4.1 HDFS-DIRECT

4.1.1 HDFS Overview.

HDFS is a distributed file system that is designed for storing large files that are sequentially written and read. Files are divided into 128MB blocks, and HDFS replicates and reads at the block level.

There are three types of HDFS servers: NameNode, JournalNode, and DataNode. The NameNode and JournalNodes store cluster metadata by running a protocol similar to Multi-Paxos; we note that this protocol can tolerate bit errors by writing an additional entry per Paxos entry (for more information, see PAR [20]). DataNodes (the local data stores in Figure 2) store HDFS data blocks, and they respond to client requests to read blocks. If a client encounters errors while reading a block, it will continue trying other DataNodes from the offset of the error until it can read the entire block. After an error on a DataNode, the client will not try that node again. If there are no more DataNodes and the block is not fully read, the read fails and that block is considered missing.

Additionally, HDFS has a configurable background "block scanner" that periodically scans data blocks and reports corrupted blocks for re-replication. But the default scan interval is three weeks, and the scanner still recovers at the 128 MB block granularity. If there is a bit error in every replica of a block, then HDFS cannot recover the block.

4.1.2 Implementing DIRECT

Minimizing Error Amplification of Data Blocks. We leverage the observation that HDFS checksums every 512 bytes in each 128 MB data block. Corruptions thus can be narrowed down to a 512 byte chunk; verifying checksums adds no overhead, because by default HDFS will verify checksums during every block read. For streaming performance, the smallest-size buffer that is streamed during a data block read is 64 KB, so we actually repair 64 KB everytime there is a corruption. To mask corruption errors from clients, we repair a data block synchronously during a read. Under DIRECT, the full read (and recovery) protocol is the following.

Each 128 MB block in HDFS is replicated on three DataNodes, call them A, B, C. An HDFS read of a 128 MB block is routed to one of these DataNodes, say A. A will stream the block to the client in 64 KB chunks, verifying checksums before it sends a chunk. If there is a checksum error in a 64 KB chunk, then A will attempt to repair the chunk by requesting the 64 KB chunk from B. If the chunk sent by B also contains a corruption, then A will request the chunk from C.

If C *also* sends a corrupted chunk, then A will attempt to construct a correct version of the chunk through bit-by-bit majority voting: the value of the bit in the chunk is the majority vote across the three versions provided by A, B, and C. After reconstructing the chunk via majority voting (Figure 3), A will verify the checksums again; if the checksums fail, then the

read fails. Majority voting allows HDFS-DIRECT to tolerate on the order of $10^4 - 10^5$ times more bit errors than HDFS. In fact, as we show in Section 5.1, UBERs can be as high as 10^{-5} before majority voting failures are detectable in our experimental framework

Safe Recovery Semantics. Safety is straightforward in HDFS because data blocks are immutable once written, so there are never updates that will conflict with chunk recovery. Before a client does a block read, it first contacts the NameNode to get the DataNode IDs of all the DataNodes on which the block is replicated. In HDFS-DIRECT, when a client sends a block read request to a DataNode, it also sends this set of IDs. Because blocks are immutable and do not contain versions, these IDs are guaranteed to be correct replicas of the block, *if they exist*. It could be that a concurrent operation has deleted the block. In this case, if chunk recovery cannot find the block on another DataNode because it has been deleted, then it cannot perform recovery, so it will return the original checksum error to the client. This is correct, because there is no guarantee in HDFS that concurrent read operations should see the instantaneous deletion of a block.

Minimizing Metadata Error Amplification. Each server in HDFS has local metadata files that must be correct, otherwise it cannot start. These files include a VERSION file, as well as special files on the NameNode and JournalNode. Metadata files are not protected in HDFS, thus a single corruption will prevent the server from starting. DIRECT adds a standard CRC32 checksum at the beginning of each file and replicates the file twice so that there are three copies of the file on disk.

4.2 ZippyDB-DIRECT

4.2.1 ZippyDB Overview

We also implemented DIRECT on a logically replicated system, ZippyDB, a distributed key-value store used within Facebook that is backed by RocksDB (i.e., RocksDB is the local data store in Figure 2), which is a versioned key-value store.

ZippyDB runs on tens of thousands of flash-provisioned servers at Facebook, which makes it an ideal target for DIRECT. ZippyDB provides a replication layer on top of RocksDB. ZippyDB is logically separated into shards, and each shard is fully replicated at least three ways. Each shard has a primary replica as well as a number of secondary replicas, wherein each replica is backed by a separate RocksDB instance residing on some server. Each ZippyDB server contains hundreds of shards, including both primary and secondary replicas. Hence, each ZippyDB server actually contains a large number of separate RocksDB instances.

ZippyDB runs a Paxos-based protocol for shard operations to ensure consistency. The primary shard acts as the leader for the Paxos entry, and each shard also has a Paxos log to persist each Paxos entry. Writes are considered durable when they are committed by a quorum of shards, and write operations are applied to the local RocksDB store in the order that they are committed. A separate service is responsible for monitoring the primary and triggering Paxos role changes.

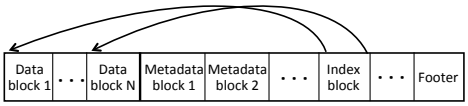


Figure 4: RocksDB SST file format. Index block entries point to keys within data blocks. Therefore, consecutive index entries form a key range. DIRECT modifies this file format by writing each metadata block at least twice in-line.

4.2.2 RocksDB Overview

RocksDB is a local key-value store based on a log-structured merge (LSM) tree [68]. RocksDB batches writes in-memory—each write receives a sequence number that enables key versioning—and flushes them into immutable files of sorted key-value pairs called sorted string table (SST) files. SST files are composed of individually checksummed blocks, each of which can be a data block or a metadata block. The metadata blocks include index blocks whose entries point to the keys at the start of each data block (see Figure 4) [13].

SST files are organized into levels. A key feature of RocksDB and other LSM tree-backed stores is background compaction, which periodically scans SST files and compacts them into lower levels, as well as performs garbage collection on deleted and overwritten keys.

4.2.3 Implementing DIRECT

ZippyDB has high error amplification since a single bit error can cause migration of terabytes of data: if a compaction encounters a corruption, an entire server, which typically has hundreds of gigabytes to terabytes of data, will shutdown and attempt to drain its RocksDB shards to another machine. Meanwhile, this sudden crash causes spikes in error rates and increases the load on other replicas while the server is recovering. To make matters worse, the new server could reside in a separate region, further delaying time to recovery.

Minimizing Error Amplification of Data Blocks. We observe that checksums in RocksDB are applied at the data block level, so a data block is the smallest recovery granularity. Data blocks are lists of key-value pairs, and key-value pairs are replicated at the ZippyDB layer. A corrupted data block can be recovered by fetching the pairs in the data block from another replica. However, this is challenging for two reasons.

First, compactions are non-deterministic in RocksDB and depend on a variety of factors, such as available disk space and how compaction threads are scheduled. Hence, *two replicas of the same RocksDB instance will have different SST files*, making it impossible to find an exact replica of the corrupted SST file and the corrupted data block, unlike in HDFS. Second, because the block is corrupted, it is impossible to know the exact key-value pairs that were stored in that block.

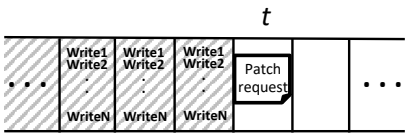


Figure 5: To serialize a patch properly, we add it as a request in the Paxos log. If the patch request is serialized at point t, then it must reflect all entries $t' < t$ (shaded). Furthermore, the patch request is not batched with any writes to ensure atomicity.

Therefore, not only do we not know what data to look for on the other replica, we also don't know where to find it.

Instead of repairing the exact keys that are lost, we rewrite a larger key range that covers the keys in the corrupted block. The key range is determined from index blocks, which are a type of metadata block in SST files that has an entry for the first key of each data block (Figure 4). Hence, consecutive index block entries form a key range which is guaranteed to contain the lost keys. Note that relying on these index entries requires that the index block, a metadata block, be error-free. See below for how we ensure the index block is uncorrupted.

Unfortunately, just knowing the key range is not enough: the existence of key versions in RocksDB and quorum replication in ZippyDB compounds the problem. In particular, a key must be recovered to a version greater than or equal to the lost key version, which could mean deleting it as key versions in RocksDB can be deletion markers. Additionally, if we naïvely fetch key versions from another replica, we may violate consistency.

Safe Recovery Semantics. To guide our recovery design, we introduce the following correctness requirement. Suppose we learn from the index entries that we must re-replicate key range $[a,b]$. This key range is requested from another replica, which assembles a set of fresh key-value pairs in $[a,b]$, which we call a patch.

Safety Requirement: *Immediately after patch insertion, the database must be in a state that reflects some prefix of the Paxos log. Furthermore, this prefix must include the Paxos entries that originally updated the corrupted data block.*

In other words, patch insertion must bring ZippyDB to some consistent state *after* the versions of the corrupted keys; otherwise, if the patch inserts prior versions of the keys, then the database will appear to go backwards.

Because the Paxos log serializes updates to ZippyDB, the cleanest way to find a prefix to recover up to is to serialize the patch insertion via the Paxos log. Then if patch insertion gets serialized as entry t in the log, the log prefix the patch must reflect is all Paxos entries $t' < t$, as shown in Figure 5. Serializing a patch at index t tells us exactly how to populate the patch. In particular, each key in the patch must be recovered to the largest entry $s < t$ such that s is the index of a Paxos entry that updates that key.

Furthermore, patch insertion must be atomic. Otherwise, it could be interleaved with updates to keys in the patch, which would violate the safety requirement, because then the version

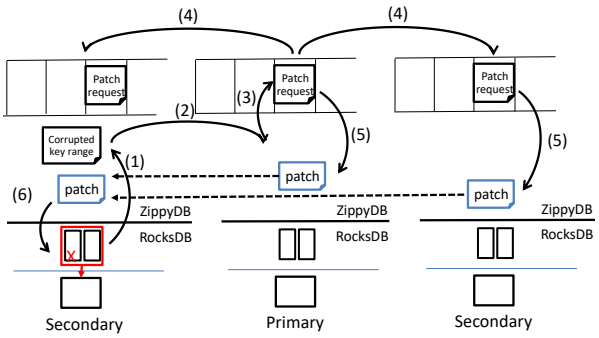


Figure 6: The process of recovering a corrupted RocksDB data block: (1) RocksDB compaction iterator determines the corrupted key range(s) based on the index blocks of the SST files and reports it to ZippyDB. (2) ZippyDB reports this error to the primary of that replica. (3) Primary shard adds patch request to Paxos log. (4) Paxos engine replicates the request to all replicas. (5) Each replica tries to process the patch request. If the processing shard is *not* the corrupted shard (which it knows because the patch request contains the shard ID of the corrupted shard), then it prepares a patch from its local RocksDB state and sends it to the corrupted shard. If the processing shard *is* the corrupted shard, then it waits for a patch from any replica. (6) Corrupted shard applies the fresh patch to its local RocksDB store.

of the key in the patch would not reflect a prefix of t. This is actually a subtle point because ZippyDB batches many writes into a single Paxos entry, as shown in Figure 5. If patch insertion is batched with other writes, then the patch will not reflect the writes that are in front of it in the batch. Hence, we force the patch insertion to be its own Paxos entry.

Minimizing Metadata Error Amplification. There are two flavors of metadata in RocksDB: metadata files and metadata blocks in SST files. Metadata files are only read during startup and then cached in memory. We can easily protect them with local replication, which adds a minimal space overhead (on the order of kilobytes per server). We protect metadata blocks by writing them several times in-line in the same SST file. In our implementation, we write each metadata block twice[2]. Protecting metadata enables us to isolate errors to a single data block, rather than invalidating an entire SST file.

As with the HDFS JournalNode, we can protect against errors in the ZippyDB Paxos log with an additional entry [20].

4.2.4 DIRECT Recovery in ZippyDB

ZippyDB-DIRECT triggers a recovery procedure when RocksDB encounters a corruption error during compaction. For user reads, ZippyDB does not synchronously recover corrupted blocks, unlike in HDFS. Instead, it returns the error to the client, which will retry on a different replica, and ZippyDB will then trigger a manual compaction involving the file of the corrupted data block.

Figure 6 depicts this process. Importantly, we do not release a compaction's output files until the recovery procedure

[2]For increased protection, metadata blocks can be locally replicated more than twice or protected with software error correction.

finishes; otherwise, stale key versions may reappear in the key ranges still undergoing recovery. Fortunately, because compaction is a background process, we can wait for recovery without affecting client operations.

Step (1) is implemented entirely within RocksDB. A RocksDB compaction iterator will record corrupted key ranges as they are encountered and withhold them from appearing in the compaction's result. At the end of the iterator's lifetime, ZippyDB is notified about the corrupted key range. Note that the compaction may encounter multiple corrupt key ranges, which are batched into a single patch request.

In step (2), the patch is reported to the primary. Step (3) must go through the primary because the primary is the only shard that can propose entries to the Paxos log. Note this does not mean primaries cannot recover from corrupted data blocks. The patch request in the Paxos log is simply a no-op that reserves a point of reference for the recovery procedure and includes information necessary for recovery, such as the corrupted key ranges and the ID of the corrupted shard. Any replica that encounters the patch request in the log is by definition up-to-date to that point in the Paxos log, which means any replica that isn't the corrupted replica is eligible to send a patch to the corrupted replica. In step (4), ZippyDB waits for the Paxos log to replicate the Paxos entry as well as for other replicas to consume the log until they encounter the patch request.

In step (5), an uncorrupted replica assembles a patch on the specified key range(s) with a RocksDB iterator. To do this, the uncorrupted replica opens a range scan iterator on each key range. Note that this replica might encounter a bit corruption while assembling the patch. In practice the probability of this is small because the number of keys covered by the patch is on the order of kilobytes (§5.2), and any scans to find such keys would predominantly look through metadata blocks, such as bloom filter or index blocks. However, if a replica does encounter a corruption while assembling a patch, it simply does not send a patch. Therefore, for the patch request to fail, *both* (or more, if the replication factor is more than 3) uncorrupted replicas will have to encounter a bit corruption, and this probability is low.

Step (6) is also implemented at the RocksDB level. When a replica applies a patch, simply inserting all the key-value pairs present in the patch is insufficient because of deleted keys. In particular, any key present in the requested key range and *not* present in the patch is an implicit delete. Therefore, to apply a patch, the corrupted shard must also delete any keys that it can see that aren't present in the patch. This case is possible because RocksDB deletes keys by inserting a tombstone value inline in SST files, but such a tombstone might have already been compacted away on the replica providing the patch. Hence the corrupted data block may contain tombstone operators that delete a key, and these must be preserved.

4.2.5 Transactions and Invalidating Snapshots

ZippyDB uses RocksDB snapshots to execute transactions. RocksDB snapshots are represented by a sequence number, s. Then, for as long as the snapshot s is active, RocksDB will not compact any version, s', of a key where s' is the greatest version of the key such that $s' < s$. If RocksDB invalidates a snapshot, then the ZippyDB transaction using that snapshot will abort and retry.

A subtle side-effect of a corrupted data block is snapshot corruption. For example, suppose the RocksDB store has a snapshot at sequence number 100 and the corrupted data block contains a key with sequence number 90. Because the data block is corrupted, it cannot be read, so we do not know whether the corruption affects snapshot 100. Then for safety, we need to invalidate all local snapshots, because any of them could have been affected by the corrupted key range. In practice, this is reasonable because most ZippyDB transactions (and hence RocksDB snapshots) have short lifetimes.

More generally, any transactional system that relies on versioning (e.g., MyRocks that is built on RocksDB), through either optimistic concurrency control or multi-version concurrency control (MVCC) can similarly deal with corruptions by aborting ongoing transactions.

4.3 Cascading Errors

In both HDFS-DIRECT and ZippyDB-DIRECT, the system will visit multiple replicas if necessary to resolve a bit error. However, currently DIRECT ignores and does not try to fix any cascading errors encountered during this process. For example, if a replica tries to assemble a patch in ZippyDB-DIRECT and fails because the iterator encounters a corruption, the replica will simply cleanup, ignore the patch request, and move to executing the next request in the Paxos log. We ignore cascading errors for simplicity but can easily incorporate recovery of cascaded errors in the future.

5 Evaluation

This section evaluates DIRECT by answering the following questions: (1) By how much does DIRECT decrease application-level errors in both HDFS and ZippyDB? In HDFS, how far can DIRECT drive UBER while avoiding application-level errors? (2) How much does DIRECT decrease time to recovery from corruption errors in ZippyDB?

Note we do not measure recovery time in HDFS because DIRECT handles bit errors *synchronously*, which means read errors only propagate to the application level if DIRECT cannot fix them. Therefore, "recovery time" can be measured by its effect on read latency. On the other hand, in ZippyDB, DIRECT handles bit errors asynchronously because recovery procedures must go through the coordination layer, as described in Section 4.2.

Experimental Setup. To evaluate ZippyDB, we set up a cluster of 60 Facebook servers that capture and duplicate live traffic from a heavily loaded service used in computing

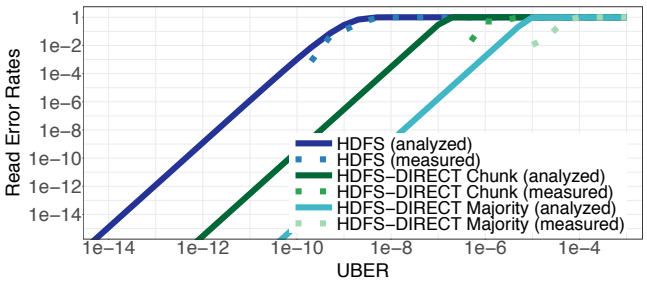


Figure 7: Read error rate for HDFS with varying UBER. HDFS-DIRECT Chunk is based on chunk-by-chunk recovery, while HDFS-DIRECT Majority is computed on bit-by-bit majority. The analyzed data is computed using the formulas in §3.

user feeds. To evaluate HDFS, we set up an HDFS cluster on machines with 8 ARMv8 cores at 2.4 GHz, 64 GB of RAM, and 120 GB of flash. The cluster has three DataNode machines, and four machines act as HDFS clients. The machines are connected with 10Gb links. HDFS experiments have a load and read phase: in the load phase, we load the cluster with 500, 128MB files with random data. In the read phase, clients randomly select files to read. After the load phase, we clear the page cache.

Error Injection. To simulate UBERs, we inject bit errors into the files of both systems. In ZippyDB, we inject errors with a custom RocksDB environment that flips bits as they are read from a file. In HDFS, we run a script in between the load and read phases that flips bits in on-disk files and flushes them. For an UBER of μ, e.g. $\mu = 10^{-11}$, we inject errors at the rate of 1 bit flip per $1/\mu$ bits read. We tested with UBERs higher than the manufacturer advertised 10^{-15} to test the system's performance under high error rates, and so that we can measure enough bit errors during an experiment time of 12 hours rather than several days (or years)[3].

Baselines. We compare against unmodified HDFS and ZippyDB, both systems used in production for many years. Although unmodified HDFS does compute checksums for chunks, it does not recover at that granularity. HDFS-DIRECT leverages these checksums during recovery, which allows it to recover blocks synchronously within client reads. In unmodified ZippyDB, when a RocksDB instance encounters a compaction error, the entire ZippyDB server crashes. While this may seem like an overly aggressive baseline, due to the difficulty of recovering an individual object in a logically-replicated system, we did not find an alternative baseline that was correct and easier to implement. One possible strawman is to recover the individual RocksDB instance that encountered a bit error. Even this approach has significant error amplification (tens of GBs per bit error), and suffers from high implementation complexity, as ZippyDB has no existing logic for recovering individual RocksDB instances.

[3] Note that an UBER 10^{-11} is $10,000\times$ *higher* than 10^{-15}.

UBER	HDFS Thruput [GB/s]	HDFS-DIRECT Thruput [GB/s]
10^{-7}	0.00 ± 0.00	2.09 ± 0.08
10^{-8}	0.00 ± 0.00	2.56 ± 0.09
10^{-9}	2.46 ± 0.08	2.55 ± 0.07
10^{-10}	2.89 ± 0.10	2.84 ± 0.07
No errors	2.83 ± 0.07	2.88 ± 0.07

Table 2: Throughput of HDFS and HDFS-DIRECT. At UBER of 10^{-8}, HDFS throughput collapses due to bit errors.

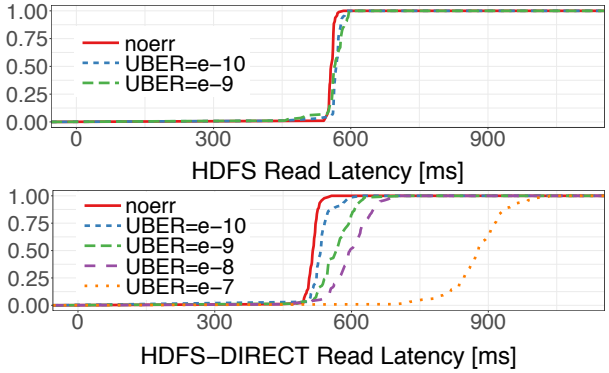


Figure 8: Read latencies (128 MB) of HDFS and HDFS-DIRECT. All reads fail in HDFS an UBER of 10^{-8} and higher. (Note that all latencies in HDFS-DIRECT are shifted slightly to the left, and this is due to temporal variations in our shared, experimental testbed.)

5.1 HDFS

UBER Tolerance. The main advantage of HDFS-DIRECT over HDFS is the ability to tolerate much higher UBERs with chunk-level recovery and majority voting. Figure 7 reports block read error rates of HDFS with varying UBERs. In HDFS, read errors are also considered *data loss*, because the data is unreadable (and hence unrecoverable) even after trying all 3 replicas. The figure shows the measured read error on our HDFS experimental setup, within the UBER range in which we could effectively measure errors, as well as the computed read error based on the computation presented in §3. We compared unmodified HDFS, with chunk-by-chunk recovery and bit-by-bit majority. The experimental read error is collected by running thousands of file reads and measuring how many fail. The measured results are relatively close to the analytical results, and in fact experience even fewer errors than the analytical model (the measured curves are shifted to the right of the analytic curves). We believe the primary reason is that the Taylor's approximation used in the analytical model does not hold for high UBERs. As expected, bit-by-bit majority (green lines) reduces the read error rate even further due to its lower error amplification (it can recover bit-by-bit). Both our analysis and the experimental results show that HDFS-DIRECT can tolerate a $10,000\times$–$100,000\times$ higher UBER and maintain the same read error rate.

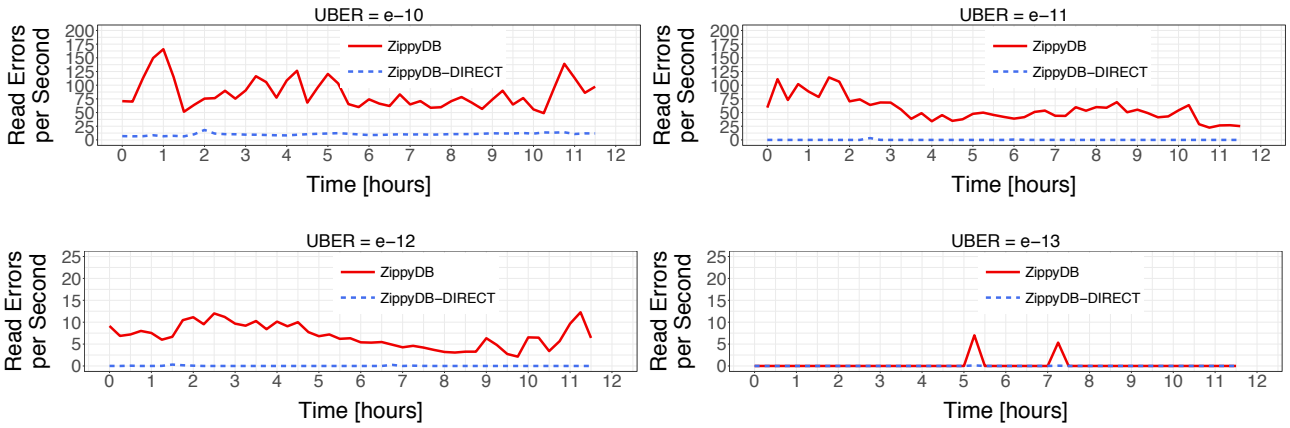


Figure 9: Read error rates over time in ZippyDB and ZippyDB-DIRECT, under varying UBERs.

UBER	Time to Complete Benchmark (s)
10^{-7}	177.4 ± 2.5
10^{-8}	169.4 ± 2.1
No errors	166.2 ± 1.8

Table 3: Time for HDFS-DIRECT to complete TeraSort benchmark. Note that we do not present results for unmodified HDFS, because for the UBERs tested, HDFS cannot complete any reads.

Overhead of DIRECT. Because DIRECT corrects bit errors synchronously in HDFS, error correction poses an overhead on reads that encounter bit errors. Table 2 shows the throughput of both systems, measured by saturating the DataNodes with four, 64-threaded clients that are continuously reading random files. The throughput of HDFS goes to zero at an UBER of 10^{-8}, because it cannot complete any reads due to corruption errors. Such failures do not occur in HDFS-DIRECT, although its throughput decreases modestly due to the overhead of synchronously repairing corrupt chunks during reads.

For HDFS-DIRECT, we are also interested in latency incurred by synchronous chunk recovery. We compare the CDF of read latencies of 128 MB blocks for different UBERs in Figure 8. The higher the UBER, the more chunk recovery requests that need to be made during a block read and the longer these requests will take. The results in Figure 8 (and Table 2) highlight the fine-grained tradeoff between performance and recoverability that is exposed by DIRECT. We also report HDFS read latencies, but there is little difference across UBERs because only latency for successful block reads are included; again, we do not report results for UBERs higher than 10^{-8}, since at those error rates HDFS cannot successfully read any blocks.

We also note that the latencies reported are the time it takes to read an entire 128 MB *file*, which is composed of many (64K) chunks. Hence Figure 8 hides a small *chunk* tail latency introduced by DIRECT. For example, chunks can now encounter errors on 0, 1, 2, or 3 of its replicas. Most chunks will encounter 0 errors, but some may encounter errors on all 3 of its replicas, which will require a relatively costly majority voting round. However, these disparate chunk latencies are hidden in the file latency, because all files have almost the expected number of errors (128 MB·$UBER$).

Interestingly, these overheads become minimal when we run an end-to-end benchmark. We ran the TeraSort benchmark, a canonical Hadoop benchmark. We configured TeraSort to generate and sort 20 GB of data. Table 3 shows the time it takes HDFS-DIRECT to complete the TeraSort benchmark. Note that at an UBER of 10^{-8}, the time it takes to complete the benchmark is similar to when there are no errors (in fact, we do not report results for UBERs lower than 10^{-8} because they are so similar to results when there are no errors). Even at an UBER of 10^{-7}, the performance overhead is relatively low, because TeraSort is dominated by sort time in the mappers and reducers, rather than the time it takes to read the data into memory. These results suggest that even at very high UBERs, DIRECT imposes a low recovery overhead in workloads that are not disk-bound.

5.2 ZippyDB

UBER Tolerance. One main difference between unmodified ZippyDB and ZippyDB-DIRECT is that ZippyDB-DIRECT avoids crashing when encountering a bit error. To characterize how many server crashes are mitigated with DIRECT, we measured the average rate of compaction errors per hour *per server*, over 12 hours. The results are shown in Table 4. Figure 9 shows the read error rate over time of both systems, and Table 4 also shows the number of read errors as a percentage of all reads. Note that the error rate patterns across UBERs are different because they are run during different time intervals, so each UBER experiment sees different traffic. We did try to ensure read/write QPS and query distribution remain steady throughout the experiments[4].

The error rate is much higher for ZippyDB than ZippyDB-DIRECT because not only do clients see errors from regular read operations, but also they experience the spike in errors when a server shuts down due to a compaction corruption.

[4]Unfortunately, there is no tracing system set up for ZippyDB, so we were unable to capture and replay traces for consistency.

	Read Errors		Compaction Errors per Hour per Server
UBER	ZippyDB	ZippyDB-DIRECT	ZippyDB
10^{-10}	2.7308%	0.1865%	0.1991 ± 0.1077
10^{-11}	1.9808%	0.0400%	0.0621 ± 0.0455
10^{-12}	0.2650%	0.0008%	0.0038 ± 0.0035
10^{-13}	0.0108%	0.0002%	0.0003 ± 0.0005

Table 4: Read and compaction errors with ZippyDB and ZippyDB-DIRECT. The read errors are a percentage of the total number of reads, and the compaction errors are the number of errors per hour per server. ZippyDB-DIRECT is able to fix all compaction errors in our experiment, while the server crashes in ZippyDB.

Time Spent in Reduced Durability. With DIRECT, we also seek to minimize the amount of time spent in reduced durability to decrease the likelihood of simultaneous replica failures. Figure 10 shows a CDF of the time it takes to recover from compaction errors in ZippyDB-DIRECT. The graph shows the amount of time it takes for replicas to process the Paxos log up until the patch request, as well as the overhead of constructing and inserting the patch. With DIRECT, this recovery time is on the order of *milliseconds*. In contrast, the period of reduced durability in unmodified ZippyDB due to a compaction error is on the order of *minutes*, depending on the amount of data stored in the crashed ZippyDB server. This is due to the high error amplification of ZippyDB, which invalidates 100s of RocksDB shards due to a single compaction bit error. With DIRECT, ZippyDB can reduce its recovery time due to a bit error by around $10,000\times$. Even with a more reasonable baseline that only invalidates the single RocksDB shard that experienced the error, we estimate that DIRECT can reduce the recovery time by around 100x.

We also found that the recovery latency is dependent on the size of the patch required to correct the corrupted key range. Figure 11 presents a CDF of the size of the patches generated during the recovery process. Patch size is also interesting because the recovery mechanism described in Section 4.2.4 recovers a *range* of keys, since the exact keys on the corrupted data block are impossible to identify. As we see in Figure 11, even though recovering a range can in theory increase error amplification, the number of keys required for recovery is still low (a single RocksDB instance contains on the order of millions of key-value pairs). Figure 11 also confirms that, generally, as the UBER increases, patch sizes increase due to more key ranges getting corrupted during a single compaction operation. We note that UBER=10^{-12} yielded an anomalous line, but by the time we analyzed the data, we no longer had access to the experimental system to rerun the results. We speculate that a variety of factors could have caused the anomaly: 1) The corruption error could have occurred on a particularly dense subset of the key space. 2) the corruption error might have occurred during a read to a bottom-level SST file in RocksDB; due to compaction, key-ranges used for recovery grow progressively larger in lower levels of the RocksDB LSM tree.

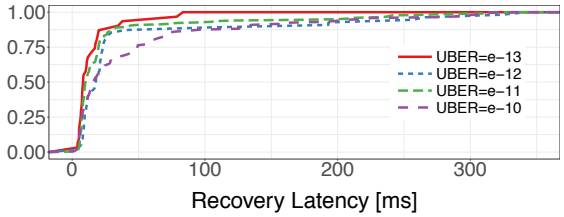


Figure 10: CDF of compaction recovery latencies in ZippyDB-DIRECT. ZippyDB-DIRECT takes milliseconds to recover from corruptions, while ZippyDB takes *minutes*.

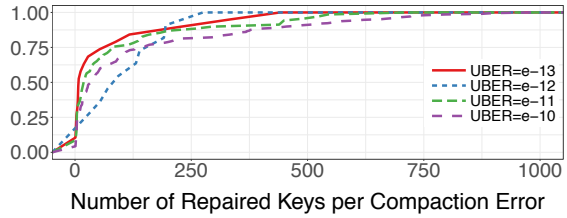


Figure 11: CDF of patch sizes generated during the ZippyDB-DIRECT recovery process. The patch size is small, which means low error amplification.

Measuring Cost. DIRECT trades off higher reliability and longer device lifetime for engineering and operational costs. We are unable to capture these costs in our evaluation but describe the changes to the software stack needed in order to run DIRECT in production. For example, ZippyDB-DIRECT required around 1200 lines of C++ code and HDFS-DIRECT required around 450 lines of Java code. As we discuss in Section 6, a DIRECT stack will also require running a hardened filesystem, such as ZFS, so that the local filesystem can continue functioning after encountering bit errors.

6 Discussion

Local File System Error Tolerance. When devices exhibit higher UBERs, local file systems also experience higher UBERs. DIRECT protects application-level metadata and data, which are data blocks to the local file system. Protecting local file system metadata such as inodes, the FS journal, etc. is beyond the scope of this paper. Several existing file systems protect metadata against bit corruptions [3, 17, 18, 43, 55, 70, 81]. The general approach is to add checksums and locally replicate for error correction. Another approach is to use more reliable hardware for metadata and less reliable hardware for data blocks [55].

Support for DIRECT. Some simple device-level mechanisms would help datacenter operators run devices past their manufacturer defined UBER. First, it would be beneficial if devices have a less aggressive "bad block policy", which is a firmware protocol for retiring blocks once they reach some heuristic-defined level of errors.

Second, it would be helpful if devices return the content of corrupted pages, although this is not a hard requirement This enables distributed storage applications to minimize recovery amplification by recovering data at a granularity smaller than

a device page. For example, to use majority voting, a system operator must use devices that return the content of corrupted pages, such as Open-Channel SSDs [25]. Fortunately, majority voting is optional and only applicable to block-replicated systems, and all other aspects of DIRECT apply if the system operator uses traditional flash devices. In case corrupt pages cannot be read, copies of local metadata must be stored on separate physical pages. Otherwise, a page error could invalidate all copies of the metadata.

Retrofitting PostgreSQL. As we discussed earlier, PostgreSQL is difficult to retrofit with DIRECT. This is because Postgres pages do not have indexing information; indexes in Postgres are stored on separate pages, if at all (no indexes are built without explicit user commands). Postgres checksums are at the page granularity, so if there is a bit error on a data page, DIRECT would need to figure out all the tuples stored on the page in order to both minimize error amplification and do correct recovery (Postgres uses MVCC to support transactions). The only way to determine these tuples and their versions is to comb through the index pages for any pointers to the corrupt page: in particular, we observe that what we need for DIRECT is a *reverse* index, one that maps from pages to tuples, rather than from tuples to pages. Generally, for DIRECT to be efficient in logically-replicated systems, the page layout must provide insight into what tuples are stored on a page. For example, RocksDB builds such a reverse index implicitly in the index blocks of its file format.

Network Partitions. Because DIRECT uses remote redundancy to correct bit errors, network failures can now affect the recovery process. Fortunately, real-world studies have shown that the most common kind of network failure— link failures — do not greatly affect application availability, because there are enough redundant paths built into the network topology [41]. In future work, we plan to both model and evaluate how transient or permanent network failures affect both recovery latency and error rate.

7 Related Work

Our work departs from existing work on data integrity in data storage systems [24, 26, 53, 73] because we expose bit corruptions at the distributed layer, rather than containing them in the storage layer. Furthermore, DIRECT does not stop at identifying corruptions but introduces a principled and performant way of fixing them to achieve high availability.

Software-level Redundancy. DIRECT is related to PAR [20] and PASC [33], which demonstrate how consensus-based protocols can be adapted to address bit-level errors. Unlike both of these works, which only address consensus protocols, our work tackles bit-level errors in general purpose storage systems. We also show how increasing the resiliency to bit errors can significantly reduce storage costs and improve live recovery speed in datacenter environments.

Other related work use different approaches. HARDFS [35] hardens local HDFS nodes by augmenting each node with a lightweight version that verifies its behavior. HDFS-DIRECT generalizes HARDFS, by only applying local protection to metadata and leveraging distributed replicas to recover data. FlexECC [46] and Duracache [60] are flash-based key-value caches that use less reliable disks by treating devices errors as cache misses. D-GRAID is a RAID storage system that gracefully degrades by minimizing the amount of data needed to recover from bit corruptions [74]. AHEAD and EDB-Tree apply software-level error detection and correction to address DRAM corruption in databases [51, 52].

There is a large number of distributed storage systems that use inexpensive, unreliable hardware, while providing consistency and reliability guarantees [23, 34, 40]. However, these systems treat bit corruptions similar to entire-node failures and suffer from high recovery amplification.

Hardware-level Redundancy. Several studies explore extending SSD lifetime via more aggressive or adaptive hardware error correction. Tanakamuru *et al.* [77] propose adapting codeword size based on the device's wear level to improve SSD lifetime. Cai *et al.* [30] and Liu *et al.* [61] introduce techniques to dynamically learn and adjust the cell voltage levels based on retention age. Zhao *et al.* [83] propose using the soft information with LDPC error correction to increase lifetime. Our approach is different: instead of improving hardware-based error correction, we leverage existing software-based redundancy to address bit-level errors.

8 Conclusion and Future Work

This paper presents DIRECT, a set of general techniques that use the inherent redundancy that exists in distributed storage applications for live recovery of bit corruptions. We showed with implementations of DIRECT in HDFS and ZippyDB that these techniques are widely applicable and, once implemented, can increase the bit error rate tolerance of distributed systems by orders of magnitude.

We envision extending the DIRECT approach in several directions. First, distributed storage systems can control error correction depending on how sensitive particular data is to bit corruptions (e.g. critical metadata). Second, distributed storage systems can control hardware mechanisms that influence the reliability as well as the performance of the device. For example, storing fewer bits per cell may reduce the latency of the device (at the expense of its capacity), and offer higher reliability. Certain applications may prefer to use a hybrid of low latency and low capacity devices for hot data, while reserving the high capacity devices for colder data.

9 Acknowledgements

We thank Mikhail Antonov, Siying Dong, Kim Hazelwood, Binu John, Chris Petersen, Muhammad Waliji, and the extended ZippyDB and RocksDB teams for their support on this project. We also thank our shepherd, Bianca Schroeder, and the anonymous reviewers for their excellent feedback.

References

[1] CockroachDB docs. https://www.cockroachlabs.com/docs/stable/.

[2] DRAM prices continue to climb. https://epsnews.com/2017/08/18/dram-prices-continue-climb/.

[3] Ext4 metadata checksums. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/.

[4] High-efficiency SSD for reliable data storage systems. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T2A_Yang.pdf.

[5] Introducing Lightning: A flexible NVMe JBOF. https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/.

[6] LevelDB. http://leveldb.org.

[7] McDipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[8] Micron 5210 ION SSD. https://www.micron.com/solutions/technical-briefs/micron-5210-ion-ssd.

[9] MongoDB docs. https://docs.mongodb.com/.

[10] Novel 4k error correcting code for QLC NAND. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FE22_Kuo.pdf.

[11] Project Voldemort: A distributed key-value storage system. http://www.project-voldemort.com/voldemort.

[12] RocksDB. http://rocksdb.org.

[13] RocksDB block based table format. https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format.

[14] SanDisk datasheet. https://www.sandisk.com/business/datacenter/resources/data-sheets/fusion-iomemory-sx350_datasheet.

[15] Under the hood: Building and open-sourcing RocksDB. http://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920.

[16] What is R.A.I.S.E? https://www.kingston.com/us/ssd/raise.

[17] XFS reliable detection and repair of metadata corruption. http://xfs.org/index.php/Reliable_Detection_and_Repair_of_Metadata_Corruption.

[18] The Z File System (ZFS). https://www.freebsd.org/doc/handbook/zfs.html.

[19] NAND flash media management through RAIN. Technical report, Micron, 2013.

[20] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32, Oakland, CA, 2018. USENIX Association.

[21] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX Annual Technical Conference*, pages 91–102, 2013.

[22] P. Alcorn. Facebook asks for QLC NAND, Toshiba answers with 100TB QLC SSDs with TSV. http://www.tomshardware.com/news/qlc-nand-ssd-toshiba-facebook,32451.html.

[23] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[24] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[25] M. Bjørling, J. González, and P. Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[26] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Dealing proactively with data corruption: Challenges and opportunities. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 34–39. IEEE, 2011.

[27] D. Borthakur. HDFS block replica placement in your hands now! http://hadoopblog.blogspot.com/2009/09/hdfs-block-replica-placement-in-your.html.

[28] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for data centers. Technical report, Google, 2016.

[29] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526, Dresden, Germany, 2012.

[30] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, pages 551–563, San Francisco, CA, 2015.

[31] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[32] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, 2013.

[33] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Berkeley, CA, USA, 2012. USENIX Association.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220, Oct. 2007.

[35] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 105–118, San Jose, CA, 2013. USENIX.

[36] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.

[37] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13, 2018.

[38] D. Exchange. DRAM supply to remain tight with its annual bit growth for 2018 forecast at just 19.6%. `www.dramexchange.com`.

[39] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[41] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, 2011.

[42] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 17–24, San Jose, CA, 2012.

[43] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 293–306, New York, NY, USA, 2007. ACM.

[44] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 199–212, Santa Clara, CA, 2014.

[45] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure Storage. In *Usenix annual technical conference*, pages 15–26. Boston, MA, 2012.

[46] P. Huang, P. Subedi, X. He, S. He, and K. Zhou. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 489–500, Philadelphia, PA, 2014.

[47] J. Jeong, S. S. Hahn, S. Lee, and J. Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *FAST*, pages 61–74, 2014.

[48] K. Kambatla and Y. Chen. The truth about MapReduce performance on SSDs. In *Proceedings of the 28th Large Installation System Administration Conference*, pages 118–126, Seattle, WA, 2014.

[49] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The memory forum*, pages 1–4, 2014.

[50] H. Kim, S.-J. Ahn, Y. G. Shin, K. Lee, and E. Jung. Evolution of NAND flash memory: From 2D to 3D as a storage market leader. In *Memory Workshop (IMW), 2017 IEEE International*, pages 1–4. IEEE, 2017.

[51] T. Kolditz, D. Habich, W. Lehner, M. Werner, and S. T. de Bruijn. AHEAD: Adaptable data hardening for on-the-fly hardware error detection during database query processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1619–1634, New York, NY, USA, 2018. ACM.

[52] T. Kolditz, T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Online bit flip detection for in-memory B-trees on unreliable hardware. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 5:1–5:9, New York, NY, USA, 2014. ACM.

[53] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, volume 2008, page 127, 2008.

[54] J. Lee, J. Jang, J. Lim, Y. G. Shin, K. Lee, and E. Jung. A new ruler on the storage market: 3D-NAND flash for high-density memory and its technology evolutions and challenges on the future. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 11–2. IEEE, 2016.

[55] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[56] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 1–1. IEEE, 2016.

[57] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 501–512, 2014.

[58] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, Vancouver, BC, 2015.

[59] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011.

[60] R. Liu, C. Yang, C. Li, and G. Chen. DuraCache: a durable SSD cache using MLC NAND flash. In *Proceedings of the 50th Annual Design Automation Conference 2013*, pages 166–171, Austin,TX, 2013.

[61] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, San Jose, CA, 2012.

[62] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016.

[63] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 106–106, New York, NY, USA, 2018. ACM.

[64] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 177–190, Portland, Oregon, 2015.

[65] R. Micheloni et al. *3D Flash memories*. Springer, 2016.

[66] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[67] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale Internet storage systems. *SIGARCH Computing Architecture News*, 42(1):471–484, 2014.

[68] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[69] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, 1988.

[70] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[71] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proceedings of the 39th International Conference on Very Large Data Bases*, pages 325–336, Trento, Italy, 2013.

[72] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 67–80, Santa Clara, CA, 2016.

[73] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36. ACM, 2005.

[74] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *Trans. Storage*, 1(2):133–170, May 2005.

[75] A. S. Spinelli, C. M. Compagnoni, and A. L. Lacaita. Reliability of NAND flash memories: Planar cells and emerging issues in 3D devices. *Computers*, 6(2):16, 2017.

[76] B. Tallis. QLC NAND arrives for consumer SSDs. `https://www.anandtech.com/show/13078/the-intel-ssd-660p-ssd-review-qlc-nand-arrives`.

[77] S. Tanakamaru, M. Fukuda, K. Higuchi, A. Esumi, M. Ito, K. Li, and K. Takeuchi. Post-manufacturing, 17-times acceptable raw bit error rate enhancement, dynamic codeword transition ECC scheme for highly reliable solid-state drives, SSDs. *Solid-State Electronics*, 58(1):2–10, 2011.

[78] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 373–386, Santa Clara, CA, 2015.

[79] The Apache Software Foundation. Apache Cassandra. `http://cassandra.apache.org/`.

[80] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 328–337, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[81] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[82] G. Zemor and G. D. Cohen. Error-correcting WOM-codes. *IEEE Transactions on Information Theory*, 37(3):730–734, 1991.

[83] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 243–256, San Jose, CA, 2013.