CrashMonkey: A Framework to Systematically Test File-System Crash Consistency

Ashlie Martinez and Vijay Chidambaram
University of Texas at Austin

Abstract

Modern file systems employ complex techniques to ensure that they can recover efficiently in the event of a crash. However, there is little infrastructure for systematically testing crash consistency in file systems. We introduce CrashMonkey, a simple, flexible, file-system-agnostic test framework to systematically check file systems for inconsistencies if a failure occurs during a file-system operation. CrashMonkey is modular and flexible, allowing the users to easily specify different test workloads and custom consistency checks.

1 Introduction

One of the core features a file system provides is keeping data safe in the event of a power loss or a crash [19]. Over the years, a number of techniques, such as journaling [11, 20], soft updates [14], and copy-on-write [12], have been developed to solve the crash consistency problem, with new solutions and file systems still being developed [3, 4, 10, 13, 17]. Such techniques are complex and their implementation touches most parts of the file system. As a result, most modifications to the file system will impact its crash-consistency guarantees in one way or another: a famous example is the introduction of delayed allocation in ext4 causing wide-spread data loss in the event of a crash [16]. Recently introduced features such as Direct Access (DAX) [9] also significantly impact file-system crash consistency.

Despite the importance and complexity of crash consistency, there currently does not exist infrastructure to systematically test the crash consistency of file systems. While developers run regression test suites (such as xfstests [6]) before committing code, these suites do not systematically test for crash consistency. Part of the reason why is because running crash-consistency tests is hard; such tests were historically performed by power cycling the server, a slow and inefficient process. A faster method is to power cycle a virtual machine instead, but even a virtual machine requires several seconds to boot up. Another challenge is that many bugs are exposed only by crashes at specific points in the write path; the window in which a crash would reveal these bugs is extremely small. Therefore, just randomly crashing the system is unlikely to uncover these bugs. As a result, developers today simply do not include crash-consistency tests as part of the development cycle. Given the increasing complexity of file systems, this leads to crash-consistency bugs being discovered only in production.

We introduce *CrashMonkey*, a framework to systematically test the crash consistency of different file systems. CrashMonkey is file-system agnostic, not depending upon the internal features of the file system being tested. CrashMonkey is flexible, allowing the user to provide custom test workloads, and provide custom notions of consistency (*e.g.,* specifying whether file contents need to be checked). CrashMonkey is efficient: it does not require the restart of the whole system, and only remounts the file system during testing.

CrashMonkey builds on prior work [18, 21] to provide these properties. CrashMonkey *constructs* disk states which may result from a crash during the given test workload. CrashMonkey achieves this by logging all IO to the disk, so that it knows exactly which blocks are already on storage at the time of the crash. While respecting write barriers, CrashMonkey constructs different crash states that are the result of blocks being reordered in the storage stack. CrashMonkey then mounts the file system being tested on the crash state, allowing the file system to perform recovery operations. CrashMonkey can then inspect the state of the file system and determine if it has recovered correctly. Constructing and testing a crash state is orders of magnitude faster than power-cycling a virtual machine.

A significant challenge in effectively testing crash consistency is figuring out which crash states to test; for a workload which writes N blocks to disk, 2^N crash states are possible (in the worst case). Since testing all possible states is not feasible, we can use file-system domain expertise to heuristically explore crash states that are likely to lead to bugs. For example, a crash state where some of the metadata blocks are lost in the crash is more likely to lead to bugs than a crash state where some of the data blocks are lost. The user can also guide CrashMonkey to exercise new and immature features of the file system such as snapshotting or deduplication.

Our hope is that CrashMonkey becomes part of the workflow for file-system developers, similar to xfstests today. Rapid file-system development should not preclude testing that file systems ensure the safety of user data in the event of a crash.

2 Current Approaches

We briefly discuss current approaches to testing file-system crash consistency, and the problems and limitations of each approach.

Current Test Frameworks. xfstests is a file-system test suite that contains a large number of regression tests for bugs that have been found over the years. xfstests does contain a few crash-consistency tests [6]; however, the crash-consistency test cases either randomly crash the machine or use the device-mapper error device to fail all writes to specific disk sectors. Thus, xfstests does not test if a file system remains consistent after a crash occurs midway through a file-system operation.

ZFS runs the file-system back-end in user space, and kills the file system at random points using `kill` [15]. While ZFS tests are similar to CrashMonkey, they are specific to the ZFS file system. Modifying an existing file system, say ext4, to be tested in a similar manner would require significant engineering effort. In contrast, CrashMonkey is file-system agnostic and can work without additional effort on new file systems.

The Linux POSIX file-system test suite (fstests) focuses on file-system correctness and detecting regression bugs[1] [8]. As the name implies, the test suite focuses on testing the POSIX interface of file systems and does not test crash consistency.

Block Order Breaker. In previous work with the University of Wisconsin-Madison, we developed the Block Order Breaker (BOB) [18]. Similar to CrashMonkey, BOB logs IOs to disk and creates different crash states. However, BOB has a different (and narrower) purpose: to show that different file systems implement file-system operations such as append and rename in drastically different ways. As such, BOB does not provide hooks for the user to provide different workloads or custom tests for consistency. BOB does not also have to solve the challenge of finding interesting crash states among the large number of possibilities.

Replay Framework. The logging and replay framework from Zheng *et al.* [21] shares our goal of building a framework to test crash consistency. Their work is focused on testing whether databases provide ACID guarantees, while CrashMonkey focuses on whether file systems are consistent after a crash. Their framework works only on iSCSI disks, while CrashMonkey works on any block storage device.

[1]Unfortunately, it appears this project was abandoned sometime around 2011

3 Background

The Linux kernel submits IO requests to the block device in the form of `bio` structures. The `bio` structure contains details about the IO request such as the data to be written (`bi_io_vec`), data location on the device (`bi_sector`), and flags (`bi_rw`) associated with the request. The `bio` flags order the IO request with respect to other requests submitted to the device. The Forced Unit Access (`FUA`) flag indicates that the request should not complete without making the associated data persistent on durable media. The `flush` flag indicates the storage cache must be flushed prior to writing the `bio` containing the `flush` flag. The `sync` flag specifies that the process that issued the `bio` is waiting for the `bio` to complete, and, therefore, indicates the request should be handled within a reasonable amount of time. The `sync` flag only serializes `sync` flagged `bio`s within a single process.

File systems carefully order `bio`s to ensure the file system remains consistent if the system crashes in the middle of an operation. For example, the ext3 file system first writes a journal transaction to storage before writing the commit block [1, 20]. The file system sends a `bio` with a `flush` flag to ensure that the transaction is persisted before it writes the commit block (with the `FUA` flag) to storage. If the file system crashes in the middle of this operation, it can read the journal and see that there is an unfinished transaction [5, 20].

At the block device driver level, device drivers often rearrange `bio` requests to reduce seek times and improve performance. Flags like `flush` and `FUA` are used to prevent such reordering. The `sync` flag does not affect this ordering since it only indicates that the process that issued the request is waiting for the request to finish.

Finally, devices themselves can cache and reorder requests they receive. `FUA` and `flush` operations are understood by devices, but `sync` requests are not. Therefore, devices could potentially persist `sync` requests out of order, making `sync` requests alone incapable of properly ordering file-system journal writes. Since CrashMonkey seeks to emulate devices, it ignores ordering restrictions arising from `sync` flags. When constructing crash states, CrashMonkey honors the ordering imposed by `FUA` and `flush` flags.

4 CrashMonkey

We present CrashMonkey, a framework to systematically test file-system crash consistency. CrashMonkey is made up of three parts. The first part consists of the test workloads and customized validation tests constructed by the user. The second part is the user-space test harness which directs test execution and interprets and rearranges logged `bio`s. The final part of our framework is a

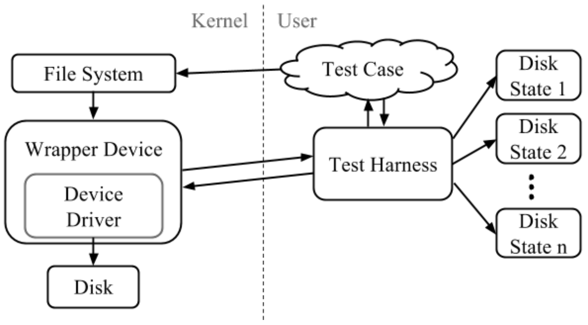

Figure 1: **CrashMonkey Architecture**. *The figure shows the major components of the* CrashMonkey *framework: the test cases, the wrapper device, and the user-space test harness.*

simple kernel block device wrapper which logs all `bios` sent to the wrapped device. Figure 1 shows an overall view of the CrashMonkey architecture.

4.1 Anatomy of a Test Case

Listing 1 shows the simple C++ interface all test cases in our framework will inherit. By forcing all test cases to inherit from a base class, we establish a set interface which then allows the test harness to dynamically load and run tests without recompiling the entire test harness.

The `setup()` method is responsible for populating the test device with any files or directories that will be used in the test. The setup method puts the test file system into a known good state before the test workload runs.

The `run_workload()` method explicitly defines the workload being tested. We explicitly define the workload in each test case because we feel the `check_test_eval()` method, which determines if workload data is consistent in various test evaluations, is intimately tied to the workload. If multiple tests have similar workloads with small differences in parameters, we encourage test authors to create parameterized library code or take advantage of C++'s inheritance system to reduce code duplication. Currently, the harness supports workloads consisting of any valid set of C++ commands. In the future we hope to provide users with a library of utility functions to help create workloads.

4.2 User-Space Test Harness

The user-space test harness contains the bulk of the functionality for CrashMonkey. The test harness is meant to be flexible and easy to add new test cases to. A set of command line flags change the file system being tested, the device to test on, and mount options for the file system under test. The responsibilities of the test harness can be split into 3 steps: pre-profiling setup, profiling,

```
class test_case {
  public:
    virtual ~test_case() {};
    virtual int setup() = 0;
    virtual int run_workload() = 0;
    virtual int check_test_eval() = 0;
};
```

Listing 1: interface for individual test cases

and testing (where we evaluate different disk images). Each of these steps also corresponds to a function defined by the `test_case` interface shown in Listing 1.

Setup. During the setup phase before profiling, the test harness formats the test device, allows the current test case to run any pre-profiling setup defined in `setup()`, and creates a snapshot of the disk. This stage runs without wrapper module logging. Once the test case has finished setting things up, we snapshot the disk. This snapshot is used to restore the disk to a known state at the start of each test evaluation.

Profiling. After all the pre-profiling work is completed, the test harness profiles the workload specified by the test case. Logging is enabled in the wrapper device and the test case's `run_workload()` method is called. The wrapper module logs all `bios` sent to the test device during workload execution. Once the workload has completed, logging is disabled and the file system under test is unmounted. We then transfer all logged data from the wrapper module to user space through an `ioctl` interface. In the future, we plan on exploring alternate interfaces to transfer data for large workloads.

Constructing Crash States. We construct a crash state by combining the snapshot of the initial state of the disk with a subset of the logged `bio` requests. We start with the snapshot and then apply different `bio` requests to the disk state, such that the ordering rules set by `FUA` and `flush` flags are respected.

To help construct crash states for a sequence of IO requests, we define a *disk epoch*: a disk epoch consists of all disk operations up to and including the first `flush` or `FUA` operation in the `bio` sequence. Disk operations can therefore be broken down into a series of disk epochs, each containing some number of `bios`.

We construct each crash state in the following manner:

- If there are N epochs in the logged IO, we first select a random epoch X. All IO requests in epochs $> X$ are dropped.
- For each epoch from 1 to X, we determine if multiple IO requests in the epoch write to the same storage location; if so, we randomly re-order the IO requests within the epoch. If all write requests within an epoch write to different locations on storage, re-

arranging them will not have any effect on the final disk state.

- For the last epoch X, we drop a random subset of write requests. Thus, we simulate a scenario where the system crashed in the middle of writing out epoch X: all previous epochs are fully persisted (with write requests re-ordered where possible), and epoch X is partially written to storage.

As with test cases, if users want a more targeted or robust reordering algorithm, they can easily write their own C++ module that adheres to our predefined interface and pass that module to the test harness at runtime.

Selecting Crash States. If the logged IO has N disk blocks, there are 2^N possible crash states if all N blocks are in the same epoch. The FUA and flush flags constrain ordering, and hence reduce the number of possible crash states. Even so, evaluating all possible crash states for a large sequence of logged IO requests is impossible. We need to focus on crash states which may expose a crash-consistency bug. We believe focusing on file-system metadata write requests can help find interesting crash states; in general, states where metadata IO is missing or re-ordered is more likely to result in inconsistency than missing data blocks [3]. Fortunately, metadata write requests are often tagged differently from data write requests by the file system (*e.g.,* the sync flag). We believe this is a good starting point to identify heuristics for finding interesting crash states.

Evaluating Crash States. Our test evaluations focus primarily on determining if the file system is consistent in the generated crash state. However, for file systems mounted with options which provide data consistency, we allow users to set a flag denoting that workload data should also be checked. We first run fsck to examine (and possibly repair) the file system, and if fsck finds no errors (or fixes some errors), we run the individual test case's check_test_eval() method to determine if the data is consistent. Thus the evaluation could yield three different results: irrecoverable file system (fsck reports unfixed errors), recovered file system with bad data (user check fails), and a fully consistent file system (fsck and user tests pass). If the test harness is simply checking for file-system consistency, we just run fsck, yielding a binary result of consistent or not.

User Tests. Custom consistency checks supplied by the user can test that the file system does not lose data if there was a crash (since an empty file system is still a consistent file system). We explain user tests with an example workload (shown in Listing 2). If the system crashes after "foo" has been printed, then file1 should contain A. Similarly, if the crash point represents a point after printing bar, file2 should contain B. We plan to provide an API in CrashMonkey so that users can set flags in CrashMonkey instead of printing to stdout. For each crash state, CrashMonkey would inspect the flags to determine what data should be available in each crash state during evaluation.

```
write(file1, A)
printf(foo)
write(file2, B)
printf(bar)
```

Listing 2: Example Custom Workload

4.3 Kernel Block Device Wrapper

The kernel block device driver is a simple block device that logs all bios sent to the device it monitors. The wrapper device intercepts all bios sent to the disk and logs the flags, data, and location on disk that bio references. The wrapper gets information about device size and allowed bio flags from the block device it monitors. By mirroring the accepted bio flags and device size, we can transparently insert a monitoring device between the kernel and the original device driver. Neglecting to mirror these flags in the monitoring device could cause some bios to change flags when they reach the wrapper device, or, in some cases, reject bios that extend past the end of the disk the wrapper device presents the system.

Implementing the wrapper device as a kernel module allows us to dynamically insert it into the kernel, thus avoiding lengthy kernel re-compilations. Adhering to the kernel's block device interface instead of integrating the wrapper module into a file system also ensures the logging mechanism is file-system agnostic.

4.4 Implementing CrashMonkey

Memory Usage. If CrashMonkey simply stores all logged write requests in kernel memory, the system could run out of memory when logging large workloads. CrashMonkey could instead asynchronously write logged data out to disk. We plan to examine the device-mapper interface in the Linux kernel as it provides similar functionality.

Heuristics for Constructing Crash States. Given that CrashMonkey cannot exhaustively evaluate all crash states, we need heuristics to identify interesting crash states. These heuristics are likely to be different for different file systems, and hence should not be hard-coded into CrashMonkey. The user should be able to easily guide the crash state construction of CrashMonkey.

Potential Workloads. Workloads that individual test cases implement should be carefully chosen to exercise different parts of the file system under test's underlying functionality. Similar to xfstests, some tests could be common to all file-system types, but other tests should be tailored to specifically test implementation details of a specific file system. For example, only tests that write large amounts of data could detect a potential flaw in the journaling and recovery logic of ext4 extents.

xfstests provides a good repository of open-source tests that exercise file-system corner cases; these could be adapted for CrashMonkey. However, we need to be careful in adopting these tests since many tests may result in similar IO patterns.

4.5 Challenges

Over the course of this project, we ran into several snags and pitfalls, both in kernel space and in user space. One of the biggest challenges of this project has been getting the kernel block device driver working properly. The majority of the available material related to the kernel target version 2.6 of the Linux kernel. Since the release of the third editions of *Understanding the Linux Kernel* and *Linux Device Drivers* [2, 7], no definitive guides about writing device drivers or interacting with newer kernel structures have been made. That, combined with sparse comments in the kernel code and misleading function names made it very hard to track down the `generic_make_requests_function()`, which silently removes flags on `bios` sent to devices that do not support those flags. This led to significant confusion when something as simple as appending text into a file on ext4 failed to produce any `bios` with `flush` or `FUA` flags.

In user space, we faced issues in selecting block devices and figuring out to how to snapshot devices. Preliminary traces logged different numbers of `bios` sent to block devices backed by LVM, ramdisk, and virtual disks mounted in a VM. LVM seemed like a prime choice because of its ability to quickly create writeable snapshots; however, testing showed that LVM generated more than double the number of `bio` requests than a virtual disk did. Furthermore, the number of `bios` generated for multiple runs of the same test varied greatly. We have not yet identified the root cause of this behavior.

5 Future Work

Parallel Evaluation. Our goal is to optimize CrashMonkey performance so that we can construct and test several crash states per minute. Achieving this might be challenging since evaluating each crash state requires remounting the file system and running the file system's recovery methods. We plan on using multiple threads to achieve the required throughput. One can imagine using a cluster of machines to do the evaluation as well; each evaluation is a stand-alone computation that is independent of other evaluations.

Constructing Crash States. We plan on improving the reordering algorithm and heuristics used to create crash states. Improving the permutation algorithm can help us improve test evaluation coverage by allowing us to identify and skip over disk states that represent similar conditions. This allows us to test a wider range of `bio` request sequences in the same number of test evaluations.

Application-Level Crash Consistency. Since we allow the user to supply custom workloads and consistency checks, the user could use CrashMonkey to test application-level crash consistency, similar to ALICE [18]. However, there is a key difference: ALICE logs systems calls and constructs possible crash states based on a specification of the file system; in contrast, CrashMonkey will produce crash states based on the block IO traffic from the file system. Thus, CrashMonkey will produce a smaller set of crash states, corresponding more closely to the common case behavior of the file system; unless we take special efforts, corner-case file-system behavior (such as low free inode count) may not be triggered for the application workload. Thus, CrashMonkey may be a useful complement to ALICE for testing common-case crash behavior of applications.

6 Conclusion

Keeping data safe after a crash is one of the chief responsibilities of the file system. Unfortunately, due to increased file system complexity and the rush to support new features, this core property has been forgotten. Crash consistency is not systematically tested, and in some cases, not tested at all.

We introduced CrashMonkey, a framework for systematically testing file-system crash consistency. CrashMonkey works across different file systems and storage devices. CrashMonkey allows the user to provide different test workloads and custom consistency checks. Our hope is that CrashMonkey eventually becomes part of the workflow for file-system developers. Although file-system development proceeds at a rapid pace, we believe crash consistency is too important to not test before making significant changes to the file system.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and guidance. This work was supported by generous donations from VMware and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of other institutions.

References

[1] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 0.9 ed. Arpaci-Dusseau Books, 2014.

[2] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, 2005.

[3] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, November 2013).

[4] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)* (San Jose, California, Feb. 2012), pp. 101–116.

[5] Corbet, J. Barriers and journaling filesystems. `https://lwn.net/Articles/283161/`, 2008.

[6] Corbet, J. Toward better testing. `https://lwn.net/Articles/591985/`, 2014.

[7] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005.

[8] Dawidek, P. J. Posix filesystem test suite. `https://github.com/zfsonlinux/fstest`.

[9] Documentation, L. K. Dax. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[10] Fryer, D., Sun, K., Mahmood, R., Cheng, T., Benjamin, S., Goel, A., and Brown, A. D. Recon: Verifying file system consistency at runtime. *Trans. Storage 8*, 4 (Dec. 2012), 15:1–15:29.

[11] Hagmann, R. *Reimplementing the Cedar file system using logging and group commit*, vol. 21. ACM, 1987.

[12] Hitz, D., Lau, J., and Malcolm, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)* (San Francisco, California, Jan. 1994).

[13] Lu, Y., Shu, J., and Wang, W. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 75–88.

[14] McKusick, M. K., Ganger, G. R., et al. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 1–17.

[15] Moore, B. Zfs and the all-singing, all-dancing test suite. `https://web.archive.org/web/20160317060437/https://blogs.oracle.com/bill/entry/zfs_and_the_all_singing`, 2005.

[16] News, L. W. That massive filesystem thread. `https://lwn.net/Articles/326471/`, March 2009.

[17] Pillai, T. S., Alagappan, R., Lu, L., Chidambaram, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 181–196.

[18] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, October 2014).

[19] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Crash Consistency. *Communications of the ACM 58*, 10 (Oct. 2015), 46–51.

[20] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)* (Anaheim, CA, April 2005), pp. 105–120.

[21] Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., and Singh, S. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 449–464.