DOI:10.1145/3570521

Article development led by acmqueue
queue.acm.org

Keeping data safe in the presence of crashes is a fundamental problem.

BY RAMNATTHAN ALAGAPPAN

Research for Practice: Crash Consistency

FOR THIS RESEARCH FOR PRACTICE entry, we asked Ram Alagappan, an assistant professor at the University of Illinois Urbana Champaign, to survey recent research on crash consistency—the guarantee that application data will survive system crashes. Unlike memory consistency, crash consistency is an end-to-end concern, requiring not only that the lower levels of the system (for example, the file system) are implemented correctly, but also that their interfaces are *used* correctly by applications.

Alagappan has chosen a collection of papers that reflects this complexity, traversing the stack from applications all the way to hardware. The first paper focuses on the file system—upon which applications that hope to provide crash consistency must rely—and uses bug-finding techniques to witness violations of interface-level guarantees. The second moves up the stack, rethinking the interfaces that file systems provide to application programmers to make it easier to write crash-consistent programs. In the last, the plot thickens with the new challenges that persistent memory brings to crash consistency. It explores how to mitigate those challenges using cache-coherent accelerators. I learned a lot reading these selections, and I am sure that you will too.

—Peter Alvaro

Peter Alvaro is an associate professor of computer science at the University of California Santa Cruz, where he leads the Disorderly Labs research group (disorderlylabs.github.io).

A critical challenge that storage systems face is how to update persistent data correctly despite system crashes (caused by a power loss or kernel bugs). At a high level, the problem is that the system may crash at any time when it is in the middle of updating its persistent structures, leaving the data in an inconsistent state. A storage system is deemed crash-consistent if it can recover the persistent data it stores to a meaningful state after crashes.

Crash consistency is of paramount importance for two main reasons:

- System crashes are inevitable. Even well-managed data centers suffer from occasional power-loss events; further, increasing software complexity means more bugs and, thus, crashes. As a result, every storage system, including local file systems, storage applications that run atop them, and persistent-memory programs, must ensure crash consistency.
- Crash consistency is critical from a user and application perspective. A storage system that can lose or corrupt data upon a crash can be disastrous, leading to a loss of trust and millions of dollars in revenue.

Achieving crash consistency is challenging. Storage systems usually execute a carefully crafted sequence of modifications to ensure they can safely move from one consistent state to another, despite crashes. Doing so correctly, how-

ever, is full of nuance and challenging even for seasoned programmers.

This article discusses three ways the systems research community strives to improve the state of affairs: Finding and fixing crash-consistency bugs; developing new abstractions that ease crash consistency; and exploiting new hardware to implement crash consistency. I have chosen papers in each of these categories. By no means are these the only papers or even the general ways that improve crash consistency, but they do provide a good overview of the problem and solution space.

Finding and Fixing Crash-Consistency Bugs

J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, V. Chidambaram.
Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th Unix Symposium on Operating Systems Design and Implementation*; https://www.usenix.org/system/files/osdi18-mohan.pdf

Perhaps the most pragmatic way to improve crash consistency of storage systems is to find bugs in crash-consistency code and fix them. This paper by Mohan et al. finds crash-consistency bugs in local file systems, the building block of many storage systems. A file system is crash-consistent if it can safely recover its internal metadata (such as inodes and bitmaps) and user data that was explicitly persisted (using `fsync` or similar operations) after a crash.

One challenge in testing file systems for crash consistency is that there are innumerable workloads, and crashes can occur at any point during the workload. A testing approach that exhaustively explores this search space is impractical. This paper addresses the problem by first studying existing crash-consistency bugs in file systems. A key observation from the study is that workloads with just three or fewer file-system operations can trigger most crash-consistency bugs.

The authors also realize that it is sufficient to inject crashes only after persistent points (that is, operations such as `fsync` that explicitly persist data). This choice makes the correctness criterion very clear: While there are no guarantees for updates that are not explicitly persisted, ones that have been persisted (via `fsync` or similar operations) must be safe. While this strategy does not guarantee finding all bugs, it offers a practical way to expose serious ones, making the approach useful.

The authors devise testing tools based on these insights and apply the tools to many file systems. One neat aspect of these tools is that they work in a black-box fashion: No file-system code modification is required for testing, so they readily apply to many file systems. The results show that even popular file systems can lose persisted data in the event of a crash. For example, `Btrfs` can lose a renamed file after a crash. The existence of such severe bugs in mature systems shows how building crash-consistent systems is a challenging task. Fortunately, fixing the bugs once they are found is often straightforward. For example, file-system developers were able to fix some of the bugs found by the authors' testing tools.

Better Abstractions to Ease Crash Consistency

T.S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau.
Application crash consistency and performance with CCFS. In *Proceedings of the 15th Usenix Conference on File and Storage Technology*; https://www.usenix.org/system/files/conference/fast17/fast17_pillai.pdf

While file systems implement mechanisms to keep their internal metadata

IMAGE BY CHUMAKOV OLEG

crash-safe, they do little to protect application data. Applications, thus, do so on their own by modifying their data via a carefully implemented update protocol (a sequence of system calls such as writes, `fsync`s, and renames). Unfortunately, while the high-level ideas to construct such protocols (for example, write-ahead logging) are well understood, *implementing* them in a crash-consistent manner on modern file systems is surprisingly difficult.

The problem is that the exact semantics of how the file system will persist the issued operations is underspecified. Specifically, file systems may reorder operations for efficiency reasons; thus, when the system recovers from a crash, a later write may have reached the disk before an earlier one. As a result, applications must reason about all possible reordered on-disk states after a crash, an arduous task even for experienced programmers.

With just a moderately complex update protocol, developers must manually reason about a multitude of states. One way to avoid reordering would be to persist the system calls in the application-issued order. However, forcing every operation synchronously to the storage is prohibitively expensive.

This paper introduces a new abstraction called *streams* to ease the construction of crash-consistent update protocols without any performance penalty. The key idea is that writes within a stream are always persisted in the issued order, obviating the need to reason about reordering in the recovery protocol. Writes from different streams can be reordered, however, a fact that file-system implementations can exploit to realize higher performance. Applications can take advantage of the stream abstraction with little code modification: They just need to issue one system call setstream at the beginning. All updates issued in the established stream will then be persisted to storage.

Crash-consistent File System (CCFS) implements the stream abstraction. It also implements new mechanisms to avoid false dependencies across streams. Several applications—such as Git, LevelDB, and Apache ZooKeeper—are crash-consistent on CCFS, but they can lose or corrupt data when run on ext4, the journaling file system for Linux. Further, CCFS approximates the performance of the reordering ext4 file system. Overall, this paper shows that a new file-system abstraction and a careful implementation can ease crash consistency for applications without forgoing performance.

Exploiting Emerging Hardware to Implement Crash Consistency

A. Bhardwaj, T. Thornley, V. Pawar, R. Achermann, G. Zellweger, R. Stutsman. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*; https://dl.acm.org/doi/pdf/10.1145/3538643.3539752

Persistent memory (PM) offers an interface and performance like that of DRAM (dynamic random-access memory). It can be accessed via load and store instructions, and its performance can approximate DRAM's performance. Unlike DRAM, however, PM is nonvolatile: Data stored on PM can be recovered after a crash. Thus, persistent structures on PM must be updated in a crash-consistent manner. Emerging accelerators connected over a cache-coherent link (for example, CXL) can offer a new way to implement (black-box) crash consistency for PM.

Most existing PM systems ensure crash consistency through WAL (write-ahead logging). Sometimes the developers handcraft the WAL protocol; other times, it is automated via a compiler pass or software library such as Intel's PMDK (Persistent Memory Development Kit) to log all PM stores. In either case, logging imposes overhead because of the extra log writes and ordering constraints (via instructions to use the hardware caches). Sometimes such update tracking and logging can also be done by the hardware (using write protection). This approach incurs huge trap overheads, however, and updates can be tracked only at page granularity.

This paper observes that PM updates can be interposed and logged with low overhead using cache-coherent accelerators. It envisions a system that works (at a high level) as follows. A persistence accelerator device (PAX) with persistent memory is attached via a cache-coherent interconnect to the host. A process can map and access this memory using regular load and store instructions. The device, however, intercepts requests for cache lines from the CPU. Loads are simply proxied to the PM on the device. Stores are more interesting because the device needs to ensure crash consistency upon stores. Upon a store, the device gets a message from the host CPU about which cache line will be modified. This allows the device to perform undo logging; in particular, the device fetches the old version (of the cache line being modified) from PM and logs the address and old value. If a crash occurs, the undo log can be used to roll back to the old (consistent) version. The proposed design reduces logging costs using asynchronous log writes and grouping updates.

This approach offers two main advantages: First, it imposes low overhead (for example, no traps, tracking at cache-line granularity); second, it provides a black-box way to transform a volatile data structure into its crash-consistent persistent counterpart with no code changes.

Overall, this is a new and exciting direction in realizing crash consistency in PM devices. More broadly, this paper provides a glimpse into how emerging hardware can be exploited to implement storage functionality.

Conclusion

Keeping data safe in the presence of crashes is a fundamental problem in storage systems. Although the high-level ideas for crash consistency are relatively well understood, realizing them in practice is surprisingly complex and full of challenges. The systems research community is actively working on solving this challenge, and the papers examined here offer three solutions.

Another promising approach that is getting traction in the systems community is to use software verification to prove crash consistency. This approach is particularly well suited for new storage systems built from scratch. It would be interesting to see which of these approaches—or a combination of them—would be widely adopted in practice. C

Ramnatthan Alagappan is an assistant professor at the University of Illinois Urbana Champaign, USA. He was previously a postdoctoral researcher at VMware Research.