IS PERSISTENT MEMORY Persistent?

A SIMPLE AND INEXPENSIVE TEST OF FAILURE- ATOMIC UPDATE MECHANISMS

TERENCE KELLY

TRUST VS. VERIFY

Preserving the integrity of application data is a paramount duty of computing systems. Failures such as power outages are major perils: A sudden crash during an update may corrupt data or effectively destroy it by corrupting metadata.

Applications protect data integrity by using update mechanisms that are atomic with respect to failure. Such mechanisms promise to restore data to an application-defined consistent state following a crash, enabling application recovery.

Unfortunately, the checkered history of failure-atomic update mechanisms precludes blind trust. Widely used relational databases and key-value stores often fail to uphold their transactionality guarantees.²⁴ Lower on the stack, durable storage devices may corrupt or destroy data when power is lost.²⁵ Emerging NVM (non-volatile memory) hardware and corresponding failure-atomic update mechanisms^{7,8} strive to avoid repeating the mistakes

of earlier technologies, as do software abstractions of persistent memory for conventional hardware.^{10,11} Until any new technology matures, however, healthy skepticism demands firsthand evidence that it delivers on its integrity promises.

Prudent developers follow the maxim, “Train as you would fight.” If requirements dictate that an application must tolerate specified failures, then the application should demonstrably survive such failures in pre-production tests and/or in game-day failure-injection testing on production systems.¹ Sudden whole-system power interruptions are the most strenuous challenge for any crash-tolerance mechanism, and there’s no substitute for extensive and realistic power-failure tests. In the past, my colleagues and I tested our crash-tolerance mechanisms against power failures,^{3,17,20,21} but we did not document the tribal knowledge required to practice this art.

This article describes the design and implementation of a simple and cost-effective testbed for subjecting applications running on a complete hardware/software stack to repeated sudden whole-system power interruptions. A complete testbed costs less than \$100, runs unattended indefinitely, and performs a full test cycle in a minute or less. The testbed is then used to evaluate a recent crash-tolerance mechanism for persistent memory.¹⁰ Software developers can use this type of testbed to evaluate crash-tolerance software before releasing it for production use. Application operators can learn from this article principles and techniques that they can apply to power-fail testing their production hardware and software.

Of course, power-failure testing alone is but one link

The techniques presented in this article, together with other complementary assurance measures, can help diligent developers and operators keep application data safe.

in the chain of overall reliability and assurance thereof. Reliability depends on thoughtful design and careful implementation; assurance depends on verification where possible and a diverse and thorough battery of realistic tests.^{2,13} The techniques presented in this article, together with other complementary assurance measures, can help diligent developers and operators keep application data safe.

The remainder of this article is organized as follows: The next section briefly reviews persistent memory and corresponding crash-tolerance mechanisms, emphasizing the software that will be tested later in the article. Subsequent sections describe the power-interruption testbed and test results on the persistent memory crash-tolerance mechanism. All software described in this article is available at https://queue.acm.org/downloads/2020/Kelly_powerfail.tar.gz.

PERSISTENT MEMORY

Whereas *non-volatile* memory is a type of hardware, *persistent* memory is a more general hardware-agnostic abstraction, which admits implementation on conventional computers that lack NVM.¹¹ The corresponding persistent memory style of programming involves laying out application data structures in memory-mapped files, allowing application logic to manipulate persistent data directly via CPU instructions (`LOAD` and `STORE`).

The main attraction of persistent memory programming is simplicity: It requires neither separate external persistent stores such as relational databases nor translation between the in-memory data format and a different persistence format. An added benefit of

persistent memory on conventional hardware is storage flexibility. Persistent application data ultimately resides in files, and the durable storage layer beneath the file system can be chosen with complete freedom: Even if persistent application data must be geo-replicated across high-availability elastic cloud storage, applications can still access it via `LOAD` and `STORE`. Not surprisingly, persistent memory in all of its forms is attracting increasing attention from industry and software practitioners.¹⁶

The right crash-tolerance mechanism for persistent memory on conventional hardware is FAMS (failure-atomic `msync()`).¹⁷ Whereas the conventional Posix `msync()` system call pushes the modified pages of a file-backed memory mapping down into the backing file *without* any integrity guarantees in the presence of failure, FAMS guarantees that the state of the backing file always reflects the most recent successful `msync()` call. FAMS allows applications to evolve persistent data from one consistent state to the next without fear of corruption by untimely crashes. My colleagues and I have implemented FAMS in the Linux kernel,¹⁷ in a commercial file system,²⁰ and in user-space libraries.^{9,23} At least two additional independent implementations of FAMS exist.^{4,5,22} While this article emphasizes Posix-like environments, analogous features exist on other operating systems. For example, Microsoft Windows has an interface similar to `mmap()` and has implemented a failure-atomic file-update mechanism.¹⁷

This article uses the power-failure testbed described in the next section to evaluate the most recent FAMS implementation: `famus_snap` (failure-atomic `msync()` in user space via snapshots).¹⁰ The `famus_snap`

implementation is designed to be audited easily. Whereas previous FAMS implementations involved either arcane kernel/file-system code or hundreds of lines of user-space code, `famus_snap` weighs in at 51 nonblank lines of straightforward code, excluding comments. It achieves brevity by leveraging an efficient per-file snapshotting feature currently available in the BtrFS, XFS, and OCFS2 file systems.¹² A side benefit of building FAMS atop file snapshotting is efficiency: Snapshots employ a copy-on-write mechanism and thus avoid the double write of logging.²⁰ While in principle `famus_snap` may seem so clear and succinct that its correctness can be evaluated by inspection, in practice its correctness depends on the file-system implementation of snapshotting, and therefore whole-system power-failure testing is in order.

POWER-FAILURE TESTBED

The most important requirement for a power-failure testbed is that software running on the host computer must be able to cut the host's power abruptly at times of its own choosing. Power must then somehow be restored to the host, which must respond by rebooting and starting the next test cycle. The host computer should be rugged, able to withstand many thousands of power interruptions, and it should be able to perform power-off/on cycles rapidly. It should also be affordable to developers with modest budgets, and it should be cheap enough to be expendable, as the stress of repeated power cycling may eventually damage it. Indeed, you should positively *prefer* to use the flimsiest hardware possible: By definition, such hardware increases the likelihood of test failure,

therefore successful tests on cheap machines inspire the most confidence. The remainder of this section describes the host computer, auxiliary circuitry, and software that together achieve these goals.

Host computer

Choosing a good host computer isn't easy. Renting from a cloud provider would satisfy the frugality objectives, but unfortunately, cloud hardware is so thoroughly mummified in layers of virtualization that, by design, customer software cannot physically disconnect power from the bare metal. High-end servers expose management interfaces that allow them to be rebooted or powered off remotely, but such shutdowns are much gentler than abrupt power cutoffs; if you could somehow abruptly cut a high-end server's power, you would risk damaging the expensive machine. Laptops are cheap and throwaways abundant, but laptops lack BIOS features to trigger an automatic reboot upon restoration of external power. Workstations and desktop PCs have such BIOS features, but they're bulky, power-hungry, and they boot slowly.

Single-board computers such as the Raspberry Pi are well suited for our purposes: They are small, rugged, cheap enough to be expendable, and draw very little power. The Pi runs the Linux operating system and nearly all Linux software. It boots quickly and automatically when powered on. Its GPIO (general-purpose input/output) pins enable unprivileged software to control external circuitry conveniently. Most importantly, it's a minimalist no-frills machine. If software and storage devices pass power-failure tests on a Pi, they would likely fare no worse on

more expensive feature-rich hardware. The testbed in this article uses the Raspberry Pi 3 Model B+, which will be in production until at least 2026.¹⁸

The main downsides of single-board computers are restrictions on CPU capabilities and peripheral interfaces. For example, the Pi 3B+ CPU doesn't support Linux "soft dirty bits," and storage is limited to the onboard microSD card and USB-attached devices. It's possible, however, to connect a wide range of storage devices via, for example, SATA-to-USB adapters. Overall, the attractions of single-board computers for the present purpose outweigh their limitations.

Power-interruption circuits

In the past, my colleagues and I tested our crash-consistency mechanisms using AC power strips with networked control interfaces.^{3,17,20,21} These power strips tend to be fussy and poorly documented. Our previous test systems included a separate control computer in addition to the computer that hosted the software and storage devices under test; the control machine used the power strip to cut and restore power to the host. In retrospect, these earlier test frameworks seem unnecessarily complex, rather like "buying a car to listen to the radio." The minimalist power-interruption circuits described in this section are cheaper, more elegant, support more strenuous tests, and enable the host machine to control power cutoffs directly.

The testbed presented here uses electromechanical relays to physically disconnect power from the host computer. Relays faithfully mimic the effects of abrupt power interruptions and completely isolate the host.

Power-supply circuitry can contain a surprising amount of residual energy, enough to enable even server-class computers to shut down somewhat gracefully when utility mains power fails.¹⁵ Our power-interruption circuit therefore interposes between the host computer and its power supply, which eliminates the possibility that residual energy in the power supply might somehow enable an orderly host shutdown rather than a sudden halt.

It turns out that a remarkably simple circuit suffices to disconnect power momentarily from the host computer, which reliably triggers an immediate reboot. The circuit, shown in figure 1, is built around a monostable (nonlatching) relay. When sufficient current energizes the relay's coil, movable poles switch from their normally closed position to their normally open position. (Figure 1 follows the convention found on many relay datasheets: The normally *closed* contacts are shown closest to the coil)

FIGURE 1: **RELAY-ONLY CIRCUIT**

and you are to imagine that current in the coil pushes the relay's poles up toward the normally *open* contacts.) We use a relay¹⁹ whose contacts can carry enough power for the Pi and whose coil can be operated by a Pi's 3.3-volt GPIO pins without exceeding their 16-milliamp current limit. The relay's 180Ω coil, together with the 31Ω internal resistance of a GPIO pin,¹⁴ appropriately limits the current.

As shown in figure 1, the Pi's power supply is routed through the relay's normally closed contacts. When software on the Pi uses a GPIO pin to energize the relay's coil, power to the Pi is cut as the relay's poles jump away from the normally closed contacts. The GPIO pin on the now-powerless Pi then stops pushing current through the coil, so the poles quickly fall back to the normally closed position, restoring power to the Pi and triggering a reboot. When current ceases to flow through the relay coil, the magnetic field in the coil collapses, releasing energy that could harm the delicate circuitry on the Pi that controls the GPIO pin. A customary *protection diode* connected in parallel to the relay's coil prevents such damage. (For a description of diode protection for inductive loads, see *The Art of Electronics*.⁶ My circuits use 1N4001 diodes.)

The main worry surrounding the circuit of figure 1 is that it restores power to the host computer very quickly, which might somehow mask failures that a longer power outage would expose.

Figure 2 shows a second circuit, called PiNap, that interrupts power to the Pi for several seconds.

As shown in figure 2(a), the circuit includes two capacitors: One helps to switch the relay and the other takes over the role of energizing the relay's coil

FIGURE 2: PINAP CIRCUIT AND ITS STAGES OF OPERATION

while the Pi is powered off. Figure 2(b) depicts normal operation: The external 5.1 V source supplies power to the Pi through the relay's normally closed contacts; it also charges both capacitors. Figure 2(c) shows the transient situation as software on the host computer energizes the relay's coil via a GPIO pin: The external power supply is immediately disconnected from the host, but energy from capacitor C2 enables the GPIO pin to push the relay's poles all the way up to close the normally open contacts. Then, as shown in figure 2(d), capacitor C1 discharges through the coil, keeping the Pi powered off for a few seconds. When the energy in C1 is spent, the relay's poles drop back to the normally closed contacts, restoring power to the Pi, which reboots for the next test cycle.

Simple calculations determine the specifications

of all components in the PiNap circuit. The relay is the same as in the circuit of figure 1, for the same reasons.¹⁹ Capacitor C1 is charged to 5.1 V so you first choose the resistor to reduce the voltage drop across the relay coil to approximately 3 V; by Ohm's law 120Ω is appropriate. Now you choose C1 such that the RC (resistor-capacitor) time constant of C1 and the resistor is on the order of a few seconds; anything in the neighborhood of 10–40 millifarads (10,000–40,000 μF) will do. Electrolytic capacitors in this range are awkwardly large—roughly the size of a salt shaker—so I use a much smaller 22,000 μF supercap. Finally, capacitor C2 must be capable of holding enough energy to keep the coil energized while the relay's poles are in flight (roughly one millisecond, according to the relay datasheet); 220 μF or greater does the trick.

Figure 3 shows a closeup of PiNap on a breadboard. The relay is the white box near the center. Capacitor C2 is the cylinder on the right edge; the black rectangle at the top right is supercap C1. The resistor and protection diode are the small cylinders oriented vertically and horizontally, respectively. All of the components fit comfortably on the U.S. quarter at the bottom of the photo, and the complete circuit occupies the top half of a breadboard the size of a playing card. The host computer's GPIO and ground pins are connected via the red and black vertical jumpers flanking the relay, respectively. External 5.1 V power enters the breadboard via the inner rails on either side of the breadboard and exits via the outer rails.

Some relays and capacitors require correct DC polarity; mine do. Sending current the wrong way through a polarity-sensitive relay coil will fail to switch the poles,

FIGURE 3: PINAP CIRCUIT ON BREADBOARD

and the protection diode will provide a short-circuit path. Incorrect polarity can cause a polarized capacitor to malfunction. Pay close attention to polarity when assembling circuits.

Figure 4 shows the complete testbed on a sheet of U.S. letter-size graph paper: the PiNap breadboard is at bottom center, the Raspberry Pi with USB-attached storage device at right, and the power supply at left. The Pi's USB mouse and keyboard have been removed for clarity. To interpose PiNap between the power supply and the Pi, I cut the power cord and soldered breadboard-friendly 22 AWG solid copper wires to its stranded wires; this was by far the slowest step in assembling the testbed hardware. Everything shown in figure 4 can be purchased for less than \$100 U.S.

Two final tips for building this testbed:

FIGURE 4: **COMPLETE PINAP TESTBED**

➔ First, the relay's pin rows are spaced too closely to span the center furrow of a breadboard, so I fashioned an adapter from a wire-wrap socket (visible between the relay and breadboard in figure 3). I have also tried the alternative of clamping and/or soldering wires to the relay's pins, but the makeshift adapter shown in figure 3 is neater and seems less likely to damage the relay.

➔ Second, the output voltage on some of the Raspberry Pi 3B+'s GPIO pins fluctuates during boot. The power-interruption circuits require a pin that remains at zero volts until software running on the Pi sets it to output logical "HI" (3.3 V). Physical pin 40 works well as a GPIO output pin, and physical pin 39 provides zero-volt ground. These pins are nearest the Pi's USB ports; see the jumpers in figure 4.

Omitted from this article for both brevity and aesthetics

is a third power-interruption circuit that I designed and built before the circuits of figures 1 and 2. It used two relays, an integrated circuit timer chip, several resistors, capacitors, and diodes, and a separate 12 VDC power supply in addition to the Pi's 5.1 V power supply. My first circuit worked reliably in thousands of tests, and in some ways it is easier to explain than PiNap, but it is costlier, more complex, and harder to assemble than the circuits presented in this article. The main contribution of my first circuit was to showcase undersimplification, inspiring a search for leaner alternatives.

System configuration and test software

The particulars of configuring the host computer and test software to test the `famus_snap` library against power failures are somewhat tedious. A brief high-level summary of test setup procedures is provided in this section. Detailed instructions are part of the source-code tarball that accompanies this article. (Ambitious readers: Log in to your Pi as default user “`pi`,” untar in home directory `/home/pi/`, and see the `README` file.)

The host hardware configuration is relatively straightforward. The host computer must be attached to the external circuit of figure 1 or figure 2 via GPIO and ground pins, with the Pi's power routed through the circuit. A storage device connects via one of the Pi's USB ports. Note that configuring the host for testing *destroys all data on the storage device*.

Host computer software configuration involves several steps. The host runs the Raspbian variant of Linux; I have used several versions of this operating system from 2018

and 2019. Several nondefault software packages must be installed, notably `xfsprogs`, which is used to create a new XFS file system on the USB-attached storage device. XFS is used because `famus_snap` relies on efficient reflink snapshots, and XFS is one of a handful of file systems that support this feature. It is possible to configure a Pi to boot in a stripped-down fashion (e.g., without starting a graphical user interface). A lean boot might be faster, but the difference is small and a default boot is fast enough (less than one minute).

The sample code tarball contains additional software specific to the `famus_snap` tests. The main power-failure test program is called `pft`. It maps a backing file into memory, repeatedly fills the in-memory image with a special pseudo-random pattern, and calls the `famus_snap` analog of `msync()` to commit the in-memory image to a snapshot file. The goal of these tests is to see if power failures corrupt the application-level data of `pft`; thus, the pseudo-random pattern is designed so that corruption is easy to detect.

The `cron` utility is set up to run a script called `rab` every time the Pi boots, which happens when the external circuit restores power. The main job of `rab` is to invoke a test script, `pft_run`. Script `pft_run` starts test program `pft`, waits for a few seconds, and then uses a GPIO pin to activate the external circuit that cuts and restores power to the Pi.

The `rab` script supports an alternative to power-failure tests: It can be used to *reboot* the Pi suddenly while `pft` is running, which is less stressful than a power failure but easier to set up, as the external circuitry is unnecessary.

After a suitable number of off/on cycles have completed, you can check to see whether `pft`'s data have been corrupted by running the `check` script. The `check` script runs `pft` in recovery mode, which inspects the appropriate snapshot files to see if they contain the expected pseudo-random pattern.

RESULTS

The tests of `famus_snap` used all three external power-interruption circuits discussed in this article: the circuits of figures 1 and 2 and the third complex circuit mentioned briefly before. The tests were run on three different storage devices: a cheap (\$30) 64-GB flash thumb drive; an allegedly rugged and rather expensive (\$220) 512-GB flash memory stick; and a moderately priced (\$90) 500-GB portable SSD (solid-state drive). A total of more than 58,000 power-off/on test cycles ran. Each power-off/on cycle takes roughly one minute, which is considerably faster than the five-minute cycle times of test environments that my colleagues and I built in the past.^{3,17,20,21} All tests passed perfectly; not a single byte of data was corrupted.

Inspecting the detritus left behind by tests sheds light on what the software under test was doing when power was cut. The `famus_snap` library creates a snapshot of the backing file when application software calls its `msync()` replacement; the caller chooses the snapshot file names and decides when to delete them. The `pft` test application alternates between two snapshot files; `famus_snap`'s rules state that post-crash recovery should replace the backing file with the most recent readable snapshot file.¹⁰

During a month-long test run that completed 49,626 power-off/on test cycles using the pricey flash memory stick, power cuts left the pair of snapshot files in all four logically possible situations: only one snapshot file exists, and it is suitable for recovery (0.054 percent of tests); both files exist and are full (i.e., the same size as the backing file), but only one is readable (4.7 percent); both snapshot files are full and readable, so recovery must compare their last-mod timestamps (43.8 percent); and one file is full and readable, but the other is undersized and writable (51.4 percent).

These results conform to my expectations based on the relative amounts of time that the `pft` application and the `famus_snap` library spend in different states. One way to alter the balance of test coverage would be to trigger power failures from *within* either the `pft` application or the `famus_snap` library, analogous to the inline “crashpoint” tests used in the `famus` library⁹ (not to be confused with `famus_snap`).

As Dijkstra famously noted, testing can show the presence of bugs but not their absence. My results don’t prove that `famus_snap` will always uphold its data integrity guarantees, nor that the Raspbian operating system, the XFS file system, the Raspberry Pi computer, or the tested storage devices are reliable under power fault. I can merely report that these artifacts did not avail themselves of numerous opportunities to disappoint. Successful results on a minimalist library such as `famus_snap` and modest hardware such as the Pi furthermore raise the bar for full-featured, expensive hardware and/or software. A commercial relational database running on a server-class host and enterprise-grade storage had better survive tens

of thousands of sudden whole-system power interruptions flawlessly—or the vendors have some explaining to do!

SUMMARY AND CONCLUSIONS

Power failures pose the most severe threat to application data integrity, and painful experience teaches that the integrity promises of failure-atomic update mechanisms can't be taken at face value. Diligent developers and

operators insist on confirming integrity claims by extensive firsthand tests. This article presents a simple and inexpensive testbed capable of subjecting storage devices, system software, and application software to ten thousand sudden whole-system power-interruption tests per week.

The recent `famus_snap` implementation of failure-atomic `msync()` passed tens of thousands of power-failure tests with flying colors, suggesting that all components of the hardware/software stack—test application code, `famus_snap` library code, XFS file system, operating system, storage devices, and

host computer—are either functioning as intended or remarkably lucky.

Future work might adapt the techniques of this

Related articles

➡ Persistent Memory Programming on Conventional Hardware

The persistent memory style of programming can dramatically simplify application software.

Terence Kelly

<https://queue.acm.org/detail.cfm?id=3358957>

➡ Fault Injection in Production

Making the case for resilience testing

John Allspaw, Etsy

<http://queue.acm.org/detail.cfm?id=2353017>

➡ Abstracting the Geniuses

Away from Failure Testing

Ordinary users need tools that automate the selection of custom-tailored faults to inject.

Peter Alvaro and Severine Tymon

<https://queue.acm.org/detail.cfm?id=3155114>

article to design testbeds around other types of single-board computers [e.g., those based on other CPU types]. Arguably the most important direction for future work is the deployment and widespread application of thorough torture-test suites for artifacts that purport to preserve data integrity in the presence of failures. It's ironic that *performance* benchmarks for transaction-processing systems abound, but *crash-consistency* test suites are comparatively rare, as though speed were more important than correctness. The techniques of this article and methods from the research literature²⁴ have identified effective test strategies. It's time to put this knowledge into practice.

References

1. Allspaw, J. 2012. Fault injection in production. *acmqueue* 10(8); <http://queue.acm.org/detail.cfm?id=2353017>.
2. Alvaro, P., Tymon, S. 2017. Abstracting the geniuses away from failure testing. *acmqueue* 15(5); <https://queue.acm.org/detail.cfm?id=3155114>.
3. Blattner, A., Dagan, R., Kelly, T. 2013. Generic crash-resilient storage for Indigo and beyond. Technical Report HPL-2013-75, Hewlett-Packard Laboratories; <http://www.hpl.hp.com/techreports/2013/HPL-2013-75.pdf>.
4. Hellwig, C. 2017. Failure-atomic writes for file systems and block devices. <https://lwn.net/Articles/715918/>.
5. Hellwig, C. 2019. Failure-atomic file updates for Linux. In Linux Piter 2019; Presentation: <https://linuxpiter.com/en/materials/2307>; patches: <https://www.spinics.net/lists/linux-xfs/msg04536.html> and <http://git.infradead.org/>

- users/hch/vfs.git/shortlog/refs/heads/O_ATOMIC.
6. Horowitz, P., Hill, W. 2015. The Art of Electronics, third edition, pages 38-39 and 818. Cambridge University Press.
 7. Intel. Optane technology; <http://www.intel.com/optanel>.
 8. Intel. Persistent Memory Development Kit; <http://pmem.io/pmdkl>.
 9. Kelly, T. famus: Failure-Atomic msync() in User Space. <http://web.eecs.umich.edu/~tpkelly/famus/>.
 10. Kelly, T. 2019. Good old-fashioned persistent memory. ;login: 44(4), 29-34; https://www.usenix.org/system/files/login/articles/login_winter19_08_kelly.pdf. [Source code for famus_snap library available at https://www.usenix.org/sites/default/files/kelly_code.tgz.]
 11. Kelly, T. 2019. Persistent memory programming on conventional hardware. acmqueue 17(4); <https://dl.acm.org/citation.cfm?id=3358957>.
 12. Linux Programmer's Manual. ioctl_ficlone(); http://man7.org/linux/man-pages/man2/ioctl_ficlone.2.html.
 13. McCaffrey, C. 2016. The verification of a distributed system. acmqueue 13(9); <http://queue.acm.org/detail.cfm?id=2889274>.
 14. McManus, S., Cook, M. 2015. Raspberry Pi, second edition, 281. John Wiley & Sons.
 15. Narayanan, D., Hodson, O. 2012. Whole-system persistence. In Proceedings of the 17th Architectural Support for Programming Languages and Operating Systems (ASPLOS); <https://dl.acm.org/doi/proceedings/10.1145/2150976>
 16. Swanson, S. (organizer). Persistent programming in real life (conference); <https://pirl.nvsl.io/>.

17. Park, S., Kelly, T., Shen, K. 2013. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys); <https://dl.acm.org/citation.cfm?id=2465374>.
18. Raspberry Pi 3 Model B+; <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>.
19. TE Connectivity. Axicom relay, product code IM21TS, part number 1-1462039-5. Vendor datasheets: <https://www.te.com/usa-en/product-1-1462039-5.html>; <https://www.te.com/usa-en/product-1-1462039-5.datasheet.pdf>; https://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtv&DocNm=1-1773734-7_IM_Relay_I_Type&DocType=DS&DocLang=English.
20. Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., Morrey, B. 2015. Failure-atomic updates of application data in a Linux file system. In Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST); <https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf>.
21. Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., Morrey, B. 2015. SQLearn: database acceleration via atomic file update. Technical Report HPL-2015-103, Hewlett-Packard Laboratories; <http://www.labs.hp.com/techreports/2015/HPL-2015-103.pdf>.
22. Xu, J., Swanson, S. 2016. NOVA: A log-structured file system for hybrid volatile/nonvolatile main memories. In Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST); <https://www.usenix.org/system/files/conference/fast16/fast16-papers-xu.pdf>.

23. Yoo, S., Killian, C., Kelly, T., Cho, H. K., Plite, S. 2012. Composable reliability for asynchronous systems. Usenix Annual Technical Conference. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoo>.
24. Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Bill W. Zhao, B. W., Singh, S. 2014. Torturing databases for fun and profit. In Proceedings of the 11th Usenix Symposium on Operating Systems Design and Implementation (OSDI); https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-zheng_mai.pdf [Note that an errata sheet is provided separately.]
25. Zheng, M., Tucek, J., Qin, F., Lillibridge, M. 2013. Understanding the robustness of SSDs under power fault. In Proceedings of the 11th Usenix Conference on File and Storage Technologies (FAST); <https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf>.

Terence Kelly <tpkelly@acm.org> is a Distinguished Member and a Lifetime Member of ACM. He studied computer science at Princeton and the University of Michigan, earning his Ph.D. at the latter in 2002. He then spent 14 years at Hewlett-Packard Laboratories. During his final five years at HPL, he developed software support for nonvolatile memory. Kelly now teaches and evangelizes the persistent memory style of programming. His patents and publications are listed at <http://ai.eecs.umich.edu/~tpkelly/papers/>.

Copyright © 2020 held by owner/author. Publication rights licensed to ACM.