Nonvolatile Memory is a Broken Time Machine

Benjamin Ransford
University of Washington

Brandon Lucia
Microsoft Research

Abstract

Energy harvesting enables intermittently powered devices to compute without built-in power. But frequent power failures, combined with nonvolatile memory intended to protect computational state, introduce strange control flow that turns sequential code into unwieldy concurrent code: programs must grapple with their own state from previous interrupted runs. This paper describes the *broken time machine* problem for these devices and outlines potential solutions from the perspective of safe concurrent programming.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Checkpoint/restart

Keywords Nonvolatile memory, intermittent power

1. Introduction

Embedded devices with general-purpose CPUs are increasingly powering emerging applications like the "internet of things," wearable computing, and implantable medical devices. Recent enhancements in energy harvesting make it possible to power such devices solely from energy in their environments, reducing size and weight [3], but at a cost of unpredictable, intermittent power—breaking the familiar abstraction of a constant power supply. When not enough energy is available, an energy-harvesting device is forced to power down and wait for better conditions. Powering down destroys volatile state including execution context, but nonvolatile data (in flash, FRAM, etc.) are retained; eventually execution resumes from the start of the program.

This position paper explains how frequent reboots make programming complicated and how nonvolatile memory compounds that complexity. Our position is that key challenges remain in providing system support for application correctness on intermittently powered devices.

MSPC'14, June 13, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2618128.2618136

2. Broken Time Machines

Losing state impedes progress under adverse power conditions. It also complicates program analysis by introducing an implicit edge in the control-flow graph (CFG) from each program statement to the beginning of the program. Fig. 1a shows a program that extends a buffer by one character. Without power failures, this code is sequential. If the power fails in `append()`, control flows back to its call site in `main()`. Arcs represent the control flow edges that are both *implicit* (not reflected in the code) and *non-local* (cross function boundaries). Implicit, non-local control flow is hard to reason about.

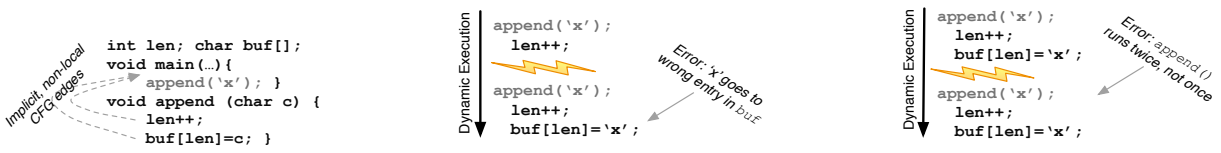
Recent work [4, 7] observed that by periodically checkpointing execution context to nonvolatile memory, it is possible to make progress despite power failures. Checkpointing preserves volatile state, but introduces yet more implicit CFG edges, from each instruction to checkpoints.

In addition to complicating control flow, checkpoints combined with explicit manipulation of other nonvolatile data can lead to inconsistent program states after a power failure. After a failure, restoring a checkpoint may result in two kinds of inconsistency. *NV-internal inconsistency* occurs if data structures in nonvolatile memory are partially updated before an interruption. *NV-external inconsistency* occurs if nonvolatile memory is updated after one checkpoint, but an interruption occurs before the next checkpoint.

Fig. 1b shows how NV-internal inconsistency affects a dynamic execution of the code from Fig. 1a. `len` and `buf` are nonvolatile and should be updated atomically. In the figure, the power fails after `len` is incremented, but before `buf` is updated, violating the update's atomicity. Control flows back to `main()` and again into `append()`. At next boot, the function increments `len` again, causing an inconsistency: `len` was incremented twice and `buf` was never updated. When `buf` is updated, `append()` writes its second entry, not its first.

Fig. 1c shows NV-external inconsistency in an execution of the code in Fig. 1a. The execution calls and fully executes `append()`. Power fails just before the function returns. The execution restarts in `main()` and fully executes `append()`. By the end, two characters were appended to `buf`, despite the presence of only one call to `append()`. `len` and `buf` were updated atomically; the cause of the error in this case is that the persistent variables reflect updates made *after* the point where execution resumes after failure. On restart, the variables' values are from an interrupted, hypothetical future.[1]

[1] Hence the metaphor of a broken time machine, a classic comedy trope.



(a) Power failures result in control flow. **(b) NV-internal inconsistency.** **(c) NV-external inconsistency.**

Figure 1: (a) The arcs show two possible control flows from points in `app()` to a point in a different function. Despite not being explicit in the code these flows may be exercised when power fails. **(b)** The nonvolatile variables `len` and `buf` should be updated atomically. A power failures violates the atomicity of the updates leading to an inconsistent program state. **(c)** Even if `len` and `buf` are updated atomically, (e.g., using system support [2, 5, 11]) inconsistency between nonvolatile state and the execution context after a failure can lead to incorrect behavior.

Prior work on transactional nonvolatile object updates [2, 5, 11] provide atomicity, addressing NV-internal inconsistency, but do not address NV-external inconsistency. Even with atomic updates, the data are out of sync with the execution context just after the restart. We assert that systems must prevent both forms of inconsistency.

Ubiquitous NV memory. Forthcoming nonvolatile memory technology may move all storage (except execution context) into nonvolatile memory [1, 10, 12]. This approach provides "free" persistence, but it also suggests a worrisome distinction, from a consistency perspective. Programs may use state that is *intentionally* nonvolatile, like persistent data structures. But programs are also likely to use state that is *incidentally* nonvolatile, like loop induction variables. Both types persist across failures, and both are vulnerable to the kinds of inconsistency described above.

3. Research Challenges

Our position is that we need new programming models and system support to address correctness and programmability in intermittently powered systems.

System support. One approach to preventing the inconsistencies we describe is to checkpoint all *persistent* state. This approach increases the storage and access cost of nonvolatile variables, which is most likely unacceptable under the tight resource limits of energy-harvesting devices. Future work should study how to reduce these costs.

Another approach is to provide system support for lightweight, application-specific recovery after power loss, inspired by recovery blocks [6]. An effective recovery mechanism facilitates reasoning about nonvolatile and checkpointed state after a failure, and allows writing code to restore consistency. Designing such a mechanism is a research challenge.

A major research challenge is to design system support that eliminates implicit, non-local control flow on power failures. The main idea is to make control flow explicit and provide strong consistency guarantees at certain program points. Task- or transaction-based programming interfaces are a good starting point. Task boundaries are natural control-flow targets, and consistency at task boundaries would be a useful guarantee. Programming and tool support could help programmers place tasks to minimize overhead and reason about implicit control flow and consistency.

Programming support. The consistency problems we discuss often stem from violations of expected atomicity and ordering properties. One avenue for research is to create tools that identify potential consistency problems so that programmers can find and fix them. We envision support in the type system or compiler to address these problems and produce code that is *correct by construction*. Dealing with concurrency problems in general is often too hard for such verification efforts to handle, but energy-harvesting devices exhibit a particular kind of concurrency. Verifying useful correctness properties in this restricted domain is a more achievable research goal than doing so in the general case.

An alternative approach is to require all code to be idempotent, eliminating the consistency problem. Requiring idempotence hinders stateful, long-running applications by limiting their use of nonvolatile storage.

Concurrency. These consistency problems described above resemble those in other concurrent systems, such as multithreading, distributed systems, and databases. Borrowing ideas from these areas may be profitable, but it is unlikely that prior solutions will apply directly for several reasons. Energy harvesting systems experience failures as the common case; most prior work assumes failures are rare. Scarce energy means devices can afford little or no power overhead to deal with correctness, precluding simple ports of prior approaches like transactions [9] or log/replay [8]. Additionally, the combination of periodic checkpointing and pervasive persistence creates new problems, such as NV-external consistency, that have no clear solution. Further study will likely expose other previously overlooked complications.

System support for correctness and reliability in energy-harvesting devices can make them accessible to workaday developers. This paper merely scratches the surface of this timely and rich problem area.

Acknowledgments

We thank the anonymous reviewers and Dan Grossman for feedback. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, Mar. 2011.

[3] S. Gollakota, M. S. Reynolds, J. R. Smith, and D. J. Wetherall. The emergence of RF-powered computing. *Computer*, 47(1), 2014.

[4] H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, Jan. 2014.

[5] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. NVM duet: Unified working memory and persistent store architecture. In *ASPLOS*, Mar. 2014.

[6] B. Randell. System structure for software fault tolerance. In *Programming Methodology*, pages 362–387. Springer New York, 1978.

[7] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, Mar. 2011.

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

[9] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[10] K. Strauss and D. Burger. What the future holds for solid-state memory. *Computer*, 47(1), 2014.

[11] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, Mar. 2011.

[12] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, Dec. 2013.