Energy-interference-free System and Toolchain Support for Energy-harvesting Devices

Alexei Colin
acolin@andrew.cmu.edu
Carnegie Mellon University

Alanson P. Sample
alanson.sample@disneyresearch.com
Disney Research, Pittsburgh

Brandon Lucia
blucia@ece.cmu.edu
Carnegie Mellon University

Introduction

Energy-harvesting computers eschew tethered power and batteries by *harvesting* energy from their environment. The devices gather energy into a storage element until they have enough energy to power a computing device. Once powered, the device functions until its energy is depleted, when it browns out and gathers more energy. Software on such computing devices executes *intermittently*, as power is available. An intermittent program execution may be interrupted by a power failure at any point and with each interruption, the volatile state of the device (e.g., register file, RAM) is erased, and its non-volatile state (e.g., FRAM) is retained. Recent work [3] defined and characterized the intermittent execution model, in which a program's execution spans periods of execution perforated by power failures.

It is difficult to write programs that are correct when reboots due to power failures occur at arbitrary points in the code, and failures may occur tens or hundreds of times per second. Intermittence compromises forward progress [6, 7], and can leave memory in an unexpected, inconsistent state that is impossible in a continuously powered execution [3, 5]. These new intermittence-related failure modes are avoidable with carefully written code or specialized system support [3, 6, 2, 4, 1]. Unaddressed, these failure modes represent a new class of *intermittence bugs* that programmers must find, understand, and fix.

To find bugs, programmers need the ability to monitor a system's behavior, observe failures, and hypothesize about the behavior that led to the failure. Unfortunately for developers, this very simple debugging methodology is unusable because existing debuggers and system monitors either *expect* or *provide* power to the device being debugged. This *energy interference* masks the intermittence experienced by real program executions, preventing any intermittence bugs from manifesting as failures. Programmers are left with an unsatisfying dilemma. One option is to use a debugger to monitor the system, but to never observe a failure because the device is powered. The other option is to run intermittently without a debugger and observe the failure, but have no ability to monitor the system to help understand the bug.

Our position is that designers of system and toolchain support for energy-harvesting devices should treat *energy-interference-freedom* and *intermittence* as first-class design concerns in future systems, methodologies, and techniques. From this position, we discuss the design of an *energy-interference-free* platform for monitoring and manipulating the energy and device state of an energy-harvesting device. We see our platform as an essential step toward a toolchain for energy-harvesting devices that supports debugging, testing, and analysis of realistic, intermittent executions.

Intermittence Bugs

Programs that are correct in continuous executions may be incorrect in intermittent executions. For example, code that updates a non-volatile data structure may leave the data structure in an inconsistent state if the update is interrupted by a power failure. The code listed below appends and removes elements from a doubly-linked list. If energy runs out before the `append` function updates the tail pointer, the resulting linked-list will violate the invariant, assumed by the `remove` function, that any node not pointed to by the tail pointer has a valid `next` field. Calling `remove` on the corrupt list leads to a null dereference. This failure can occur in the field using harvested energy, but is not reproducible while using a debugger that powers it continuously.

```
func append(list, e)
  e->next <- nil
  e->prev <- tail
  list->tail->next <- e
  // Energy exhausted here
  list->tail <- e
func remove(list, e)
  e->prev->next <- e->next
  if (e = list->tail) { tail <- e->prev }
  else { e->next->prev <- e->prev }
```

Intermittence can also unintuitively prevent forward progress. The code listed below acquires readings from an accelerometer, featurizes them, runs a classifier, and persistently stores the results. When the energy demands of sensing and computating exceeds the stored energy, the device resets before it stores its results. This failure can repeat indefinitely, preventing the application from producing results. A conventional debugger is not very useful for diagnosing and eliminating this issue, because the code is correct under continuous power.

```
__nonvolatile int nv_result;
main() {
  int i, readings[NUM_READINGS];
  features_t features;
  while (1) {
    for(i = 0; i < NUM_READINGS; ++i)
      readings[i] = read_accelerometer();
    compute_features(&features, readings);
    nv_result = classify(&features); } }
```

978-1-4673-8320-2/15/$31.00 ©2015 IEEE

Limitations of Existing Approaches

Debugging intermittence bugs using existing tools is virtually impossible for several reasons. First, hardware debuggers supply energy to or consume energy from the device-under-test (DUT). Second, developers lack information about the timing of changes in energy state and program events. Third, under intermittence, developers lack visibility into an execution's memory and device state.

Flashing LEDs at points of interest is a ubiquitous strategy for debugging embedded systems that does not work in energy-harvesting devices, due to the high current draw of an LED. For instance, using an LED to indicate when the WISP device [7] is executing code, rather than just charging, increases the WISPs current draw by five times.

Another debugging strategy is to instrument application code to log important program events to non-volatile memory. The resulting trace lacks information about the energy level, unless the developer also spends time, energy, and an ADC channel to log the DUT's energy state. This technique also deprives the application of some precious non-volatile storage space. A variation on the above strategy is to stream the event log to a separate, always-on system (e.g., via UART). Powering and clocking an I/O peripheral to transfer the log uses enough energy that it would change the point where the application fails.

Dedicated debugging equipment, like a JTAG debugger, offers visibility into the device's state but provides continuous power, masking intermittence. JTAG isolator devices [8] exist to decouple debug host power rails from DUT power rails, but do not help with intermittence debugging because the JTAG protocol fails if the DUT powers off.

A mixed-signal oscilloscope can provide an energy trace and a limited set of program events encoded onto spare GPIO pins. This fundamentally limited approach is not a familiar one, and does not accommodate interactive debugging. Moreover, professional oscilloscopes cost tens of thousands of dollars, making them inaccessible to many.

An intermittence-aware toolchain

Today's development tools and techniques either have access to program state and alter energy state, or preserve energy state and have very limited access to the program state. Development on energy-harvesting devices demands a familiar debugger environment that is *energy-interference-free*. Such a debugger must support inspecting program behavior, while simultaneously monitoring and manipulating energy state. The debugger must be able to track energy events and correlate them with program events, even to use them as triggers for interactive debugging. One important challenge is providing a set of features that help understand what the application was doing when its energy was exhausted, and how much progress it can make using a particular amount of stored energy. A complementary challenge is manually reproducing intermittence-induced behavior. A building-block for such a feature is a mechanism for manipulating the amount of energy stored on the device. Application behavior should be robustly agnostic to the energy level and intermittent interruptions, but the development toolchain must be aware of both.

References

[1] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embedded Systems Letters, IEEE*, PP(99):1–1, 2014.

[2] 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.

[3] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 575–585, New York, NY, USA, 2015. ACM.

[4] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*, Mar. 2013.

[5] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 5:1–5:3, New York, NY, USA, 2014. ACM.

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

[7] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

[8] www.segger.com. Segger - the embedded experts - jtag isolator. `https://www.segger.com/jtag-isolator.html`, 2015.