An Energy-Aware Debugger for Intermittently Powered Systems

Development and debugging support is a prerequisite for the adoption of intermittently operating energy-harvesting computers. This work identifies and characterizes intermittence-specific debugging challenges that are unaddressed by existing debugging solutions. This work addresses these challenges with the Energy-Interference-Free Debugger (EDB), the first debugging solution for intermittent systems. This article describes EDB's co-designed hardware and software implementation and shows its value in several debugging tasks on a real RF-powered energy-harvesting device.

Alexei Colin
Carnegie Mellon University

Graham Harvey
Alanson P. Sample
Disney Research Pittsburgh

Brandon Lucia
Carnegie Mellon University

Energy-harvesting devices are embedded computing systems that eschew tethered power and batteries by harvesting energy from radio waves,[1,2] motion,[3] temperature gradients, or light in the environment. Small form factors, resilience to harsh environments, and low-maintenance operation make energy-harvesting computers well-suited for next-generation medical, industrial, and scientific sensing and computing applications.[4]

The power system of an energy-harvesting computer collects energy into a storage element (that is, a capacitor) until the buffered energy is sufficient to power the device. Once powered, the device can operate until energy is depleted and power fails. After the failure, the cycle of charging begins again. These charge–discharge cycles power the system intermittently, and consequently, software that runs on an energy-harvesting device executes intermittently.[5] In the intermittent execution model, programs are frequently, repeatedly interrupted by power failures, in contrast to the traditional continuously powered execution model, in which programs are assumed to run to completion. Every reboot induced by a power failure clears volatile state (such as registers and memory), retains nonvolatile state (such as ferroelectric RAM), and transfers control to some earlier point in the program.

Intermittence makes software difficult to write and understand. Unlike traditional systems, the power supply of an energy-harvesting computer changes high-level software behavior, such as control-flow and memory consistency.[5,6] Reboots complicate a program's possible behavior, because they are implicit discontinuities in the program's control flow that are not expressed anywhere in the code. A reboot can happen at any point in a program and cause control to flow unintuitively back to a previous point in the execution.

Published by the IEEE Computer Society

The previous point could be the beginning of the program, a previous checkpoint,[5,7] or a task boundary.[6] Today, devices that execute intermittently are a mixture of conventional, volatile microcontroller architectures and nonvolatile structures. In the future, alternative architectures based on nonvolatile structures may simplify some aspects of the execution model, albeit with lower performance and energy efficiency.[8]

Intermittence can cause correct software to misbehave. Intermittence-induced jumps back to a prior point in an execution inhibit forward progress and could repeatedly execute code that should not be repeated. Intermittence can also leave memory in an inconsistent state that is impossible in a continuously powered execution.[5] These intermittence-related failure modes are avoidable with carefully written code or specialized system support.[5–7,9,10] Unaddressed, these failure modes represent a new class of intermittence bugs that manifest only when executing on an intermittent power source.

To debug an intermittently operating program, a programmer needs the ability to monitor system behavior, observe failures, and examine internal program state. With the goal of supporting this methodology, prior work on debugging for continuously powered devices has recognized the need to minimize resources required for tracing[11] and reduce perturbation to the program under test.[12] A key difference on energy-harvesting platforms is that interference with a device's energy level could perturb its intermittent execution behavior. Unfortunately, existing tools, such as Joint Test Action Group (JTAG) debuggers, require a device to be powered, which hides intermittence bugs. Programmers face an unsatisfying dilemma: to use a debugger to monitor the system and never observe a failure, or to run without a debugger and observe the failure, but without the means to probe the system to understand the bug.

This article identifies the key debugging functionality necessary to debug intermittent programs on energy-harvesting platforms and presents the Energy-Interference-Free Debugger (EDB), a hardware–software platform that provides that functionality (see Figure 1). First, we observe that debuggers designed for continuously powered devices are not effective for energy-harvesting devices, because they interfere with the target's power supply. Our first contribution is a hardware device that connects to a target energy-harvesting device with the ability to monitor and manipulate its energy level, but without permitting any significant current to flow between the debugger and the target.

Second, we observe that basic debugging techniques, such as assertions, `printf` tracing, and LED toggling, are not usable on intermittently powered devices without system support. Our second contribution is the

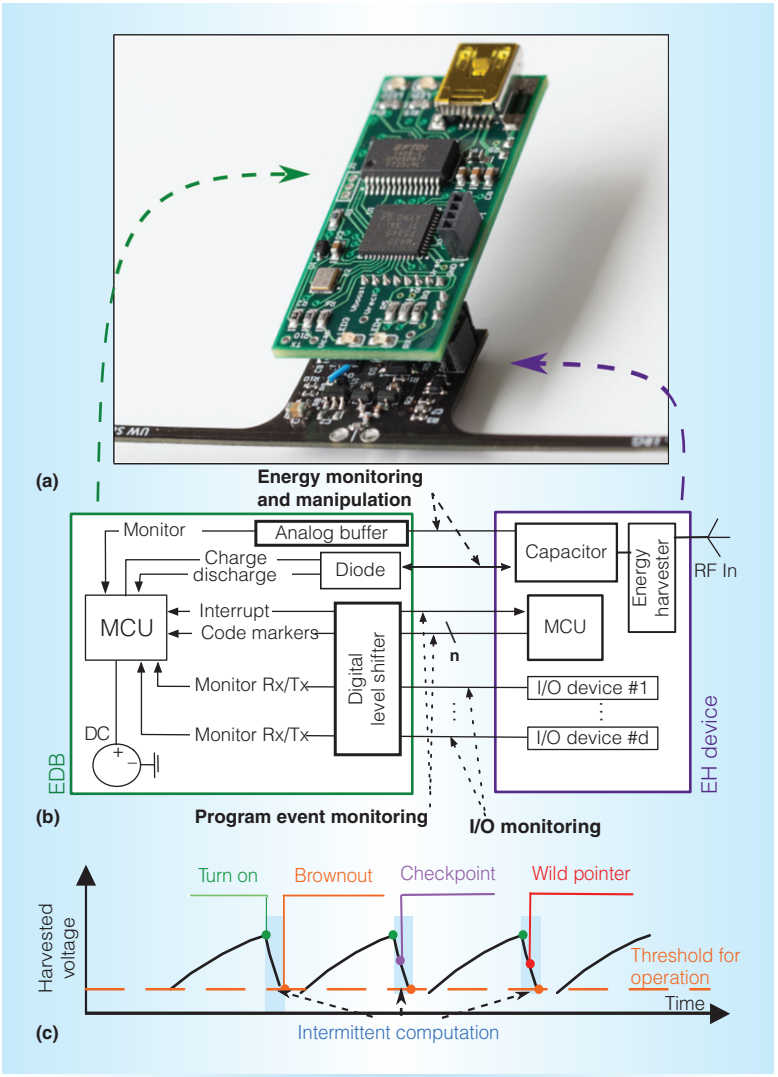
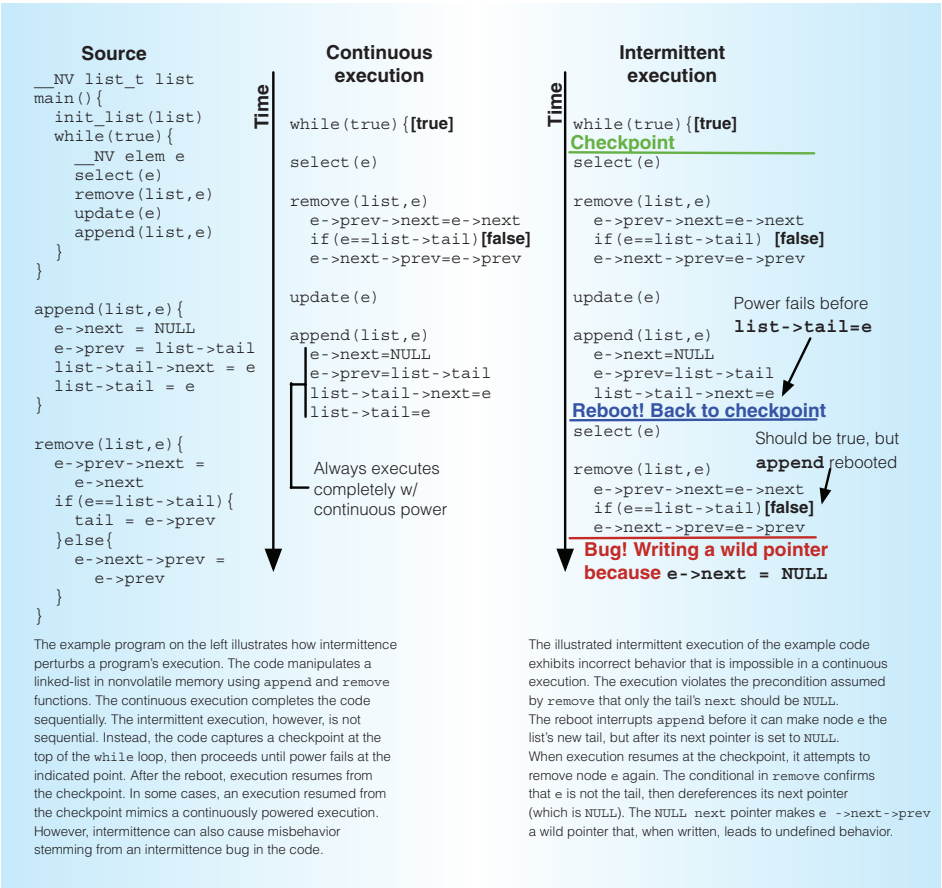


Figure 1. The Energy-Interference-Free Debugger (EDB) is an energy-interference-free system for monitoring and debugging energy-harvesting devices. (a) Photo. (b) Architecture diagram. (c) The charge–discharge cycle makes computation intermittent.



The example program on the left illustrates how intermittence perturbs a program's execution. The code manipulates a linked-list in nonvolatile memory using `append` and `remove` functions. The continuous execution completes the code sequentially. The intermittent execution, however, is not sequential. Instead, the code captures a checkpoint at the top of the `while` loop, then proceeds until power fails at the indicated point. After the reboot, execution resumes from the checkpoint. In some cases, an execution resumed from the checkpoint mimics a continuously powered execution. However, intermittence can also cause misbehavior stemming from an intermittence bug in the code.

The illustrated intermittent execution of the example code exhibits incorrect behavior that is impossible in a continuous execution. The execution violates the precondition assumed by `remove` that only the tail's `next` should be `NULL`. The reboot interrupts `append` before it can make node `e` the list's new tail, but after its next pointer is set to `NULL`. When execution resumes at the checkpoint, it attempts to remove node `e` again. The conditional in `remove` confirms that `e` is not the tail, then dereferences its next pointer (which is `NULL`). The `NULL next` pointer makes `e ->next->prev` a wild pointer that, when written, leads to undefined behavior.

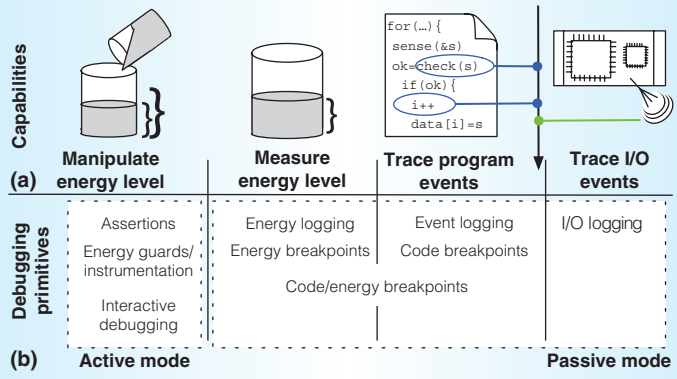
Figure 2. An intermittence bug. The software executes correctly with continuous power, but incorrectly in an intermittent execution.

EDB software system, which was codesigned with EDB's hardware to make debugging primitives that are useful for intermittently powered devices, including energy breakpoints and keep-alive assertions. EDB addresses debugging needs unique to energy-harvesting devices, with novel primitives for selectively powering spans of code and for tracing the device's energy level, code events, and fully decoded I/O events. The whole of EDB's capabilities is greater than the sum of capabilities of existing tools, such as a JTAG debugger and an oscilloscope. Moreover, EDB is simpler to use and far less expensive. We apply EDB's capabilities to diagnose problems on real energy-harvesting hardware in a series of case studies in our evaluation.

Intermittence Bugs and Energy Interference

An intermittent power source complicates understanding and debugging of a system, because the behavior of software on an intermittent system is closely linked to its power supply. Figure 2 illustrates the undesirable consequences of disregarding this link between the software and the power system. The code has an intermittence bug that leads to memory corruption only when the device runs on harvested energy.

Debugging intermittence bugs using existing tools is virtually impossible due to energy interference from these tools. JTAG debuggers supply power to the device under test (DUT), which precludes observation of a realistically intermittent execution, such as the execution on the left in Figure 2. Even JTAG, with a power rail isolator completely masks intermittent behavior, because the protocol requires the target to be powered throughout the debugging session. An oscilloscope can directly observe and trace a DUT's power system and lines, but a scope cannot



libEDB API	Debug console commands
assert (expr)	charge|discharge energy_level
break|watch_point (id)	break|watch en|dis id [energy_level]
energy_guard (begin|end)	trace {energy, iobus, rfid, watch_points}
printf (fmt, ...)	read|write address [value]

(c)

Figure 3. EDB's features support debugging tasks and developer interfaces. (a) Hardware and software capabilities. (b) Debugging primitives. (c) API and debug console commands.

observe internal software state, which limits its value for debugging.

Debugging code added to trace and react to certain program events—such as toggling LEDs, streaming events to a universal asynchronous receiver/transmitter (UART), or in-memory logging—has a high energy cost, and such instrumentation can change program behavior. For example, activating an LED to indicate when the Wireless Identification and Sensing Platform (WISP)[2] is actively executing increases its total current draw by five times, from around 1 mA to more than 5 mA. Furthermore, in-code instrumentation is limited by scarcity of resources, such as nonvolatile memory to store the log, and an analog-to-digital converter (ADC) channel for measurements of the device's energy level.

Energy interference and the lack of visibility into intermittent executions make prior approaches to debugging inadequate for intermittently powered devices.

Energy-Interference-Free Debugging

EDB is an energy-interference-free debugging platform for intermittent devices that addresses the shortcomings of existing debugging approaches. In this section, we describe EDB's functionality and its implementation in codesigned hardware and software.

Figure 3 provides an overview of EDB. The capabilities of EDB's hardware and software (Figure 3a) support EDB's debugging primitives (Figure 3b). The hardware electrically isolates the debugger from the target. EDB has two modes of operation: passive mode and active mode. In passive mode, the target's energy level, program events, and I/O can be monitored unobtrusively. In active mode, the target's energy level and internal program state (such as memory) can be manipulated. We combine passive- and active-mode operation to implement energy-interference-free debugging primitives, including energy and event tracing, intermittence-aware breakpoints, energy guards for instrumentation, and interactive debugging.

Passive-Mode Operation

EDB's passive mode of operation lets developers stream debugging information to a host workstation continuously in real time, relying on the three rightmost components in Figure 3a. Important debugging streams that

are available through EDB are the device's energy level, I/O events on wired buses, decoded messages sent via RFID, and program events marked in application code. A major advantage of EDB is its ability to gather many debugging streams concurrently, allowing the developer to correlate streams (for example, relating changes in I/O or program behavior with energy changes). Correlating debugging streams is essential, but doing so is difficult or impossible using existing techniques. Another key advantage of EDB is that data is collected externally without burdening the target or perturbing its intermittent behavior.

Monitoring signals in the target's circuit requires electrical connections between the debugger and the target, and EDB ensures that these connections do not allow significant current exchange, which could interfere with the target's behavior. To measure the target energy level, EDB samples the analog voltage from the target's capacitor through an operational amplifier buffer. To monitor digital communication and program events without energy interference, EDB connects to wired buses—including Inter-Integrated Circuit (I^2C), Serial Peripheral Interface (SPI), RF front-end general-purpose I/Os (GPIOs), and watch point GPIOs—through a digital level-shifter. As an external monitor, EDB can collect and decode raw I/O events, even if the target violates the I/O protocol due to an intermittence bug.

Active-Mode Operation

EDB's active mode frees debugging tasks from the constraint of the target device's small energy store by compensating for energy consumed during debugging. In active mode, the programmer can perform debugging tasks that require more energy than a target could ever harvest and store—for example, complex invariant checks or user interactions. EDB has an energy compensation mechanism that measures and records the energy level on the target device before entering active mode. While the debugging task executes, EDB supplies power to the target. After performing the task, EDB restores the energy level to the level recorded earlier. Energy compensation permits costly, arbitrary instrumentation, while ensuring that the target has the behavior of an unaltered, intermittent execution.

EDB's energy compensation mechanism is implemented using two GPIO pins connected to the target capacitor, an ADC, and a software control loop. To prevent energy interference by these components during passive mode, the circuit includes a low-leakage keeper diode and sets the GPIO pins to high-impedance mode. To charge the target to a desired voltage level, EDB keeps the source pin high until EDB's ADC indicates that the target's capacitor voltage is at the desired level. To discharge, the drain pin is kept low to open a path to ground through a resistor, until the target's capacitor voltage reaches the desired level. Several of the debugging primitives presented in the next section are built using this energy-compensation mechanism.

EDB Primitives

Using the capabilities described so far, EDB creates a toolbox of energy-interference-free debugging primitives. EDB brings to intermittent platforms familiar debugging techniques that are currently confined to continuously powered platforms, such as assertions and `printf` tracing. New intermittence-aware primitives, such as energy guards, energy breakpoints, and watch points, are introduced to handle debugging tasks that arise only on intermittently powered platforms. Each primitive is accessible to the end user through two complimentary interfaces: the API linked into the application and the console commands on a workstation computer (see Figure 3c).

Code and energy breakpoints. EDB implements three types of breakpoints. A *code breakpoint* is a conventional breakpoint that triggers at a certain code point. An *energy breakpoint* triggers when the target's energy level is at or below a specified threshold. A *combined breakpoint* triggers when a certain code point executes and the target device's energy level is at or below a specified threshold. Breakpoints conditioned on energy level can initiate an interactive debugging session precisely when code is likely to misbehave due to energy conditions—for example, just as the device is about to brownout.

Keep-alive assertions. EDB provides support for assertions on intermittent platforms. When an assertion fails, EDB immediately

tethers the target to a continuous power supply to prevent it from losing state by exhausting its energy supply. This keep-alive feature turns what would have to be a post-mortem reconstruction of events into an investigation on a live device. The ensuing interactive debugging session for a failed assert includes the entire live target address space and I/O buses to peripherals. In contrast to EDB's keep-alive assertions, traditional assertions are ineffective in intermittent executions. After a traditional assertion fails, the device would pause briefly, until energy was exhausted, then restart, losing the valuable debugging information in the live device's state.

Energy guards. Using its energy compensation mechanism, EDB can hide the energy cost of arbitrary code enclosed within an energy guard. Code within an energy guard executes on tethered power. Code before and after an energy-guarded region executes as though no energy was consumed by the energy-guarded region. Without energy cost, instrumentation code becomes nondisruptive and therefore useful on intermittent platforms. Two especially valuable forms of instrumentation that are impossible without EDB are complex data structure invariant checks and event tracing. EDB's energy guards allow code to check data invariants or report application events via I/O (such as `printf`), the high energy cost of which would normally deplete the target's energy supply and prevent forward progress.

Interactive debugging. An interactive debugging session with EDB can be initiated by a breakpoint, an assertion, or a user interrupt, and allows observation and manipulation of the target's memory state and energy level. Using charge–discharge commands, the developer can intermittently execute any part of a program starting from any energy level, assessing the behavior of each charge–discharge cycle. During passive-mode debugging, the EDB console delivers traces of energy state, watch points, I/O events, and `printf` output.

Evaluation

We built a prototype of EDB, including the circuit board in Figure 1 and software that implements EDB's functionality. A release of our prototype is available (http://intermittent.systems). The purpose of our evaluation is twofold. First, we characterize potential sources of energy interference and show that EDB is free of energy interference. Second, we use a series of case studies conducted on a real energy-harvesting system to show that EDB supports monitoring and debugging tasks that are difficult or impossible without EDB.

Our target device is a WISP[2] powered by radio waves from an RFID reader. The WISP has a 47 µF energy-storage capacitor and an active current of approximately 0.5 mA at 4 MHz. We evaluated EDB using several test applications, including the official WISP 5 RFID tag firmware and a machine-learning-based activity-recognition application used in prior work.[5,6]

Energy Interference

EDB's edge over existing debugging tools is its ability to remain isolated from an intermittently operating target in passive mode and its ability to create an illusion of an untouched target energy reservoir in active mode. Our first experiment concretely demonstrates the energy interference of a traditional debugging technique, when applied to an intermittently operating system. The measurements in Table 1 demonstrate the impact on program behavior of execution tracing using `printf` over UART without EDB. Without EDB, the energy cost of the print statement significantly changes the iteration success rate—that is, the fraction of iterations that complete without a power failure. Next, we show with data that EDB is effectively free of energy interference both in passive- and active-mode operation.

In passive mode, current flow between EDB and the target through the connections in Figure 1 can inadvertently charge or discharge the target's capacitor. We measured the maximum possible current flow over each connection by driving it with a source meter and found that the aggregate current cannot exceed 0.85 µA in the worst case, representing just 0.2 percent of the target microcontroller's typical active-mode current.

In active mode, energy compensation requires EDB to save and restore the voltage of the target's storage capacitor, and any discrepancy between the saved and restored voltage

Table 1. Cost of debug output and its impact on the activity-recognition application's behavior

Instrumentation method	Iteration success rate (%)	Iteration cost	
		Energy (%*)	Time (ms)
No print	87	3.0	1.1
UART `printf`	74	5.3	2.1
EDB `printf`	82	3.4	4.7

*Energy cost as percentage of 47 µF storage capacity.

represents energy interference. Using an oscilloscope, we measured the discrepancy between the target capacitor voltage saved and restored by EDB. Over 50 trials, the average voltage discrepancy was just 4 percent of the target's energy-storage capacity, with most error stemming from our limited-precision software control loop.

Debugging Capabilities

We now illustrate the new capabilities that EDB brings to the development of intermittent software by applying EDB in case studies to debugging tasks that are difficult to resolve without EDB.

Detecting memory corruption early. We evaluated how well EDB's keep-alive assertions help diagnose memory corruption that is not reproducible in a conventional debugger.

- *Application.* The code in Figure 4a maintains a doubly linked list in nonvolatile memory. On each iteration of the main loop, a node is appended to the list if the list is empty; otherwise, a node is removed from the list. The node is initialized with a pointer to a buffer in volatile memory that is later overwritten.
- *Symptoms.* After running on harvested energy for some time, the GPIO pin indicating main loop progress stops toggling. After the main loop stops, normal behavior never resumes, even after a reboot; thus, the device must be decommissioned, reprogrammed, and redeployed.
- *Diagnosis.* To debug the list, we assert that the list's tail pointer must point to the list's last element, as shown in Figure 4a. A conventional assertion is unhelpful: after the assertion fails, the target drains its energy supply and the program restarts, losing the context of the failure. In contrast, EDB's intermittence-aware, keep-alive assert halts the program immediately when the list invariant is violated, powers the target, and opens an interactive debugging session.

Interactive inspection of target memory using EDB's commands reveals that the tail pointer points to the penultimate element, not the actual tail. The inconsistency arose when a power failure interrupted `append`. In the absence of the keep-alive assert, the program would proceed to read this inconsistent state, dereference a null pointer, and write to a wild pointer.

Instrumenting code with consistency checks. On intermittently powered platforms, the energy overhead of instrumentation code can render an application nonfunctional by preventing it from making any forward progress. In this case study, we demonstrate how an application can be instrumented with an invariant check of arbitrary energy cost using EDB's energy guards.

- *Application.* The code in Figure 4b generates the Fibonacci sequence numbers and appends each to a nonvolatile, doubly linked list. Each iteration of the main loop toggles a GPIO pin to track progress. The program begins with a consistency check that traverses the list and asserts that the pointers and the Fibonacci value in each node are consistent.

- *Symptoms.* Without the invariant check, the application silently produces an inconsistent list. With the invariant check, the main loop stops executing after the list grows large. The oscilloscope trace in Figure 4c shows an early charge cycle when the main loop executes (left) and a later one when it does not (right).
- *Diagnosis.* The main loop stops executing because once the list is too long, the consistency check consumes all of the target's available energy. Once reached, this hung state persists indefinitely. An EDB energy guard allows the inclusion of the consistency check without breaking the application's functionality (see Figure 4b). The effect of the energy guard on target energy state is shown in Figure 4d. The energy guard provides tethered power for the consistency check, and the main loop gets the same amount of energy in early charge–discharge cycles when the list is short (left) and in later ones when the list is long (right). On intermittent power, we observed invariant violations in several experimental trials.

Instrumentation and consistency checking are an essential part of building a reliable application. These techniques are inaccessible to today's intermittent systems because the cost of runtime checking and analysis is arbitrary and often high. EDB brings instrumentation and consistency checking to intermittent devices.

Tracing program events and RFID messages. Extracting intermediate results and events from the executing program using JTAG or UART is valuable, but it often interferes with a target's energy level and changes application behavior. Moreover, communication stacks on energy-harvesting devices are difficult to debug without simultaneous visibility into the device's sent and received packet stream and energy state.

In Table 1, we traced the activity recognition application using EDB's energy-interference-free `printf` and watch points. In this section, we trace messages in an RFID communication stack using EDB's I/O tracer. We

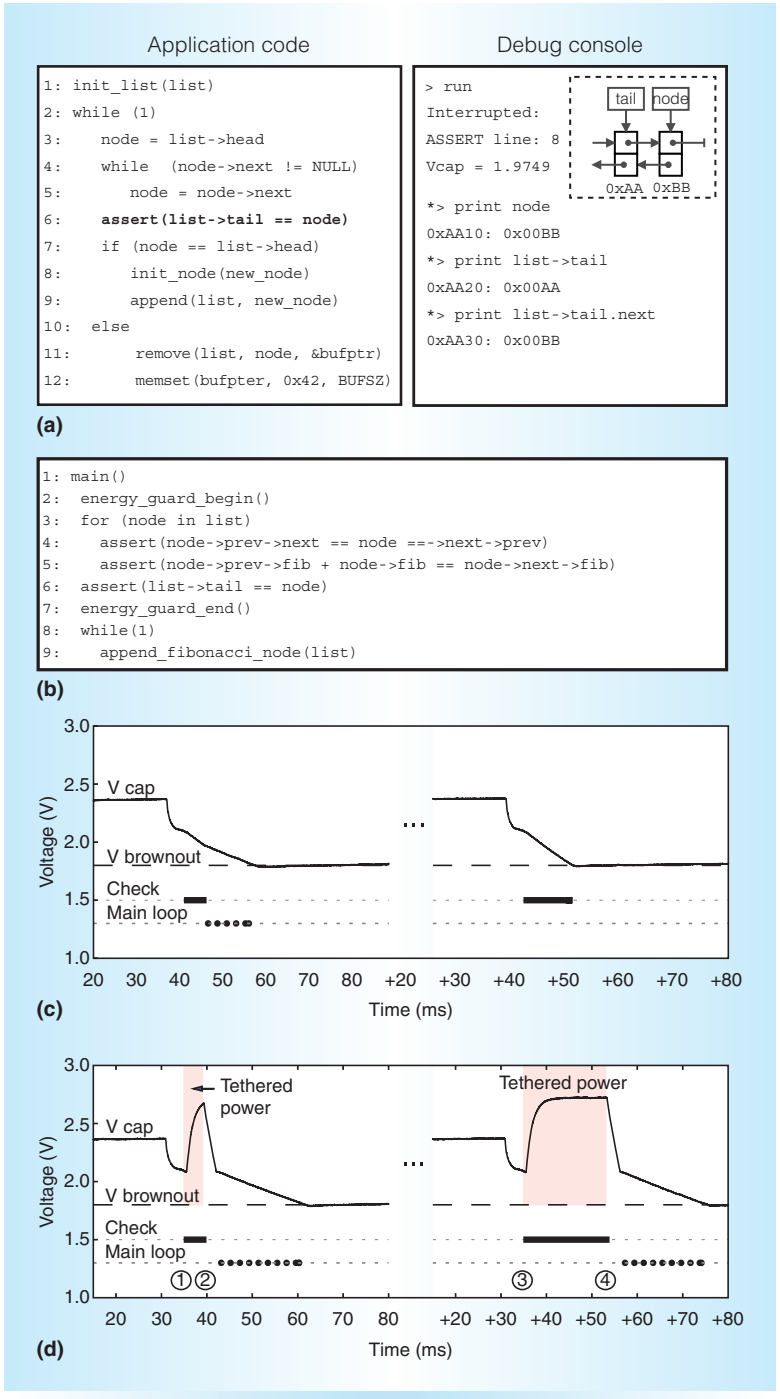


Figure 4. Debugging intermittence bugs with EDB. (a) An application with a memory-corrupting intermittence bug, diagnosed using EDB's intermittence-aware assert (left) and interactive console (right). (b) An application instrumented with a consistency check of arbitrary energy cost using EDB's energy guards. Oscilloscope trace of execution (c) without the energy guard and (d) with the energy guard. Without the energy guard, the check and main loop both execute at first, but only the check executes in later discharge cycles. With an energy guard, the check executes on tethered power from instant 1 to 2 and 3 to 4, and the main loop always executes.

used EDB to collect RFID message identifiers from the WISP RFID tag firmware, along with target energy readings. From the collected trace, we found that in our lab setup the application responded 86 percent of the time for an average of 13 replies per second. To produce such a mixed trace of I/O and energy using existing equipment, the target would have to be burdened with logging duties that exceed the computational resources, given the already high cost of message decoding and response.

Energy-harvesting technology extends the reach of embedded devices beyond traditional sensor network nodes by eliminating the constraints imposed by batteries and wires. However, developing software for energy-harvesting devices is more difficult than traditional embedded development, because of surprising behavior that arises when software executes intermittently. Debugging intermittently executing software is particularly challenging because of a new class of intermittence bugs that are immune to existing debugging approaches. Without effective debugging tools, energy-harvesting devices are accessible only to a small community of systems experts instead of a wide community of application-domain experts.

We identified energy interference as the fundamental shortcoming of available debugging tools. We designed EDB, the first energy-interference-free debugging system that supports debugging primitives for energy-harvesting devices, such as energy guards, keep-alive assertions, energy watch points, and energy breakpoints. Students in our lab and at a growing list of other academic institutions have successfully used EDB to debug and profile applications in scenarios similar to the case studies we evaluated.

EDB's low-cost, compact hardware design makes it suitable for incorporation into next-generation debugging tools and for field deployment with a target device. In the field, a future automatic diagnostic system could leverage EDB to catch rare bugs and automatically log memory states from the target device. In the lab, EDB can serve research projects that require data on energy consumption and program execution on an energy-harvesting platform, such as an intermittence-aware compiler analysis.

We created EDB because we found energy-harvesting devices to be among the least accessible platforms for research, requiring each researcher to reinvent ad hoc techniques for troubleshooting each device. EDB makes intermittently powered platforms accessible to a wider research audience and helps establish a new research area surrounding intermittent computing. MICRO

References

1. S. Gollakota et al., "The Emergence of RF-Powered Computing," *Computer*, vol. 47, no. 1, 2014, pp. 32–39.
2. A.P. Sample et al., "Design of an RFID-Based Battery-Free Programmable Sensing Platform," *IEEE Trans. Instrumentation and Measurement*, vol. 57, no. 11, 2008, pp. 2608–2615.
3. P. Mitcheson et al., "Energy Harvesting From Human and Machine Motion for Wireless Electronic Devices," *Proc. IEEE*, vol. 96, no. 9, 2008, pp. 1457–1486.
4. J.A. Paradiso and T. Starner, "Energy Scavenging for Mobile and Wireless Electronics," *IEEE Pervasive Computing*, vol. 4, no. 1, 2005 pp. 18–27.
5. B. Lucia and B. Ransford, "A Simpler, Safer Programming and Execution Model for Intermittent Systems," *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI), 2015, pp. 575–585.
6. A. Colin and B. Lucia, "Chain: Tasks and Channels for Reliable Intermittent Programs," *Proc. ACM SIGPLAN Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 514–530.
7. B. Ransford, J. Sorber, and K. Fu, "Mementos: System Support for Long-Running Computation on RFID-Scale Devices," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 159–170.
8. K. Ma et al., "Architecture Exploration for Ambient Energy Harvesting Nonvolatile Processors," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture* (HPCA), 2015, pp. 526–537.

9. D. Balsamo et al., "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, 2015, pp. 15–18.
10. M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An Energy-Aware Task Scheduler for Computational RFID," *Proc. 8th USENIX Conf. Networked Systems Design and Implementation* (NSDI), 2011, pp. 197–210.
11. V. Sundaram et al., "Diagnostic Tracing for Wireless Sensor Networks," *ACM Trans. Sensor Networks*, vol. 9, no. 4, 2013, pp. 38:1–38:41.
12. J. Yang et al., "Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks," *Proc. 5th Int'l Conf. Embedded Networked Sensor Systems* (SenSys 07), 2007, pp. 189–203.

Alexei Colin is a graduate student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests include reliability, programmability, and efficiency of software on energy-harvesting devices. Colin received an MSc in electrical and computer engineering from Carnegie Mellon University. He is a student member of ACM. Contact him at acolin@andrew.cmu.edu.

Graham Harvey is an associate show electronic engineer at Walt Disney Imagineering. His research interests include real-world applications of wireless technologies to enhance guest experiences in themed environments. Harvey received a BS in electrical and computer engineering from Carnegie Mellon University. He completed the work for this article while interning at Disney Research Pittsburgh. Contact him at graham.n.harvey@disney.com.

Alanson P. Sample is an associate lab director and principal research scientist at Disney Research Pittsburgh, where he leads the Wireless Systems group. His research interests include enabling new guest experiences and sensing and computing devices by applying novel approaches to electromagnetics, RF and analog circuits, and embedded systems. Sample received a PhD in electrical engineering from the University of Washington. He is a member of IEEE and ACM. Contact him at alanson.sample@disneyresearch.com.

Brandon Lucia is an assistant professor in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests include the boundaries between computer architecture, compilers, system software, and programming languages, applied to emerging, intermittently powered systems and efficient parallel systems. Lucia received a PhD in computer science and engineering from the University of Washington. He is a member of IEEE and ACM. Contact him at blucia@ece.cmu.edu or http://brandonlucia.com.