An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems

Alexei Colin[1,2], Graham Harvey[1,2], Brandon Lucia[2], and Alanson P. Sample[1]

Disney Research, Pittsburgh[1]
Pittsburgh, USA
graham.n.harvey@disney.com
alanson.sample@disneyresearch.com

Electrical and Computer Engineering[2]
Carnegie Mellon University
Pittsburgh, USA
{acolin, blucia}@andrew.cmu.edu

Abstract

Energy-autonomous computing devices have the potential to extend the reach of computing to a scale beyond either wired or battery-powered systems. However, these devices pose a unique set of challenges to application developers who lack both hardware and software support tools. Energy harvesting devices experience power intermittence which causes the system to reset and power-cycle unpredictably, tens to hundreds of times per second. This can result in code execution errors that are not possible in continuously-powered systems and cannot be diagnosed with conventional debugging tools such as JTAG and/or oscilloscopes.

We propose the *Energy-interference-free Debugger*, a hardware and software platform for monitoring and debugging intermittent systems without adversely effecting their energy state. The Energy-interference-free Debugger re-creates a familiar debugging environment for intermittent software and augments it with debugging primitives for effective diagnosis of intermittence bugs. Our evaluation of the Energy-interference-free Debugger quantifies its energy-interference-freedom and shows its value in a set of debugging tasks in complex test programs and several real applications, including RFID code and a machine-learning-based activity recognition system.

1. Introduction

Energy-harvesting devices are embedded computing systems that eschew tethered power and batteries by *harvesting* energy from the environment [6, 12, 13, 16, 21, 25]. A power system collects energy into a storage element (i.e., a capacitor) until the buffered energy is sufficient to power the computing 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 also executes intermittently [18, 23]. A power failure may interrupt an execution with a reboot at any point. A reboot clears volatile state (e.g., register file, SRAM), retains non-volatile state (e.g., FRAM), and transfers control to some earlier point in the program (e.g., the start of `main()`). Recent work [18] defined and characterized the *intermittent execution model*, comprising periods of execution interspersed with reboots.

Intermittence makes software difficult to write and understand. A reboot can happen at any point in a program's code, and may occur tens or hundreds of times per second. Reboots complicate a program's possible behavior because reboots are *implicit discontinuities* in the program's control flow that are not expressed anywhere in the code [18, 23]. Even with checkpointing [11, 20, 24] and versioning [18],

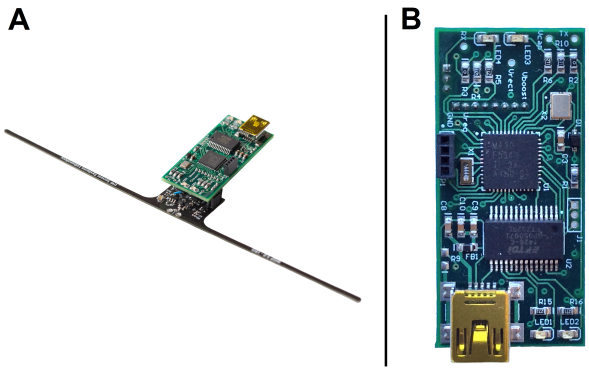

Figure 1: EDB, energy-interference-free system for monitoring and debugging energy-harvesting devices, attached to a WISP [25] (purple PCB) in Panel A and shown in detail in Panel B.

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA

DOI: http://dx.doi.org/10.1145/2872362.2872409

reboots cause control to flow unintuitively back to a previous point in the execution.

Intermittence can cause correct software to misbehave. Intermittence-induced jumps back to a prior point in an execution inhibit forward progress and may 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 [18, 23]. These failure modes represent a new class of *intermittence bugs*. To avoid intermittence-related malfunction, code must *correctly* leverage non-volatile memory. Writing intermittence-safe code for an energy-harvesting application or runtime system [2, 4, 11, 18, 20, 24] requires the programmer to understand, find, and fix intermittence bugs.

To diagnose intermittence bugs in their code, programmers need to monitor system behavior, observe failures, and examine internal program state. Unfortunately, this simple debugging methodology is unusable for intermittence bugs because existing tools power the target device, masking intermittent behavior. Programmers are left with 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 gain no insight into the system necessary for understanding the bug.

Our work addresses the lack of basic debugging support for intermittent systems with the *Energy-interference-free Debugger* (*EDB*), a complement of hardware and software for *energy-interference-free* monitoring and manipulation of intermittent devices, pictured in Figure 1. EDB can *passively monitor* a target device for its energy level, I/O events (e.g., I^2C, RFID), and program events. Monitoring with EDB, unlike with conventional debuggers, is energy-interference-free, because it is designed to be electrically isolated from the target device. EDB also provides a capability to *actively manipulate* the amount of energy stored on the device. Using this mechanism, EDB can compensate for the energy consumed by arbitrarily expensive tasks, effectively eliminating their impact on the energy state experienced by the program.

Many important intermittence debugging tasks are impossible without energy-interference-free monitoring and manipulation mechanisms. Passive monitoring allows concurrent tracing of energy, program events, and I/O under realistic scenarios. EDB's energy manipulation and compensation mechanism lets a programmer instrument application code with energy-hungry invariant checks (e.g., `asserts`) and trace statements (e.g., `printfs`) without impacting application behavior. The same mechanisms enable *interactive debugging* with breakpoints that can be conditioned on energy level and with access to the state of the target device.

To summarize our main contributions:

- We observe that *energy-interference-freedom* is essential for debugging intermittent, energy-harvesting systems.
- We design and build EDB, a platform for energy-interference-free system monitoring and manipulation.

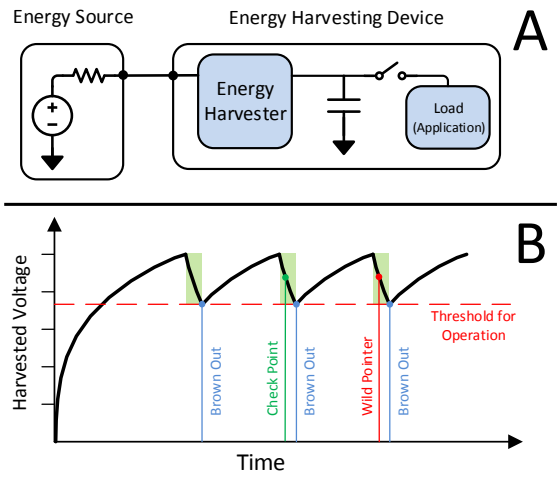

Figure 2: A simplified circuit diagram of an energy harvesting device (Panel A) and characteristic charge and discharge cycles which define its intermittent operation (Panel B).

- We develop debugging primitives for intermittent software, including energy-aware breakpoints, keep-alive assertions, and energy guards.
- We evaluate EDB and show that it is energy-interference-free and is instrumental in intermittence debugging.

2. Intermittence and Energy-interference

This section provides background on the challenges presented by intermittent energy-harvesting devices and illustrates that existing approaches to debugging fail to address these challenges. As a basis for our discussion, we assume an intermittent system that executes a C program that takes longer than a single charge-discharge execution cycle to complete. We assume our device has a mixture of volatile registers and memory, as well as some non-volatile memory. We further assume a checkpointing mechanism that periodically collects a checkpoint of volatile execution context (i.e., register file and stack) like prior work [11, 20, 24]. Note that this checkpointing assumption simplifies the exposition, but our ideas and prototype apply to a system without that support.

2.1 Challenges of Intermittence

Power intermittence complicates understanding and debugging a system, because the behavior of an intermittent system is closely linked to its power supply. This link is illustrated in Figure 2. A simplified energy-harvesting circuit diagram is shown in Figure 2A. An ambient energy transducer (e.g., solar, RF, vibration) connects to an energy storage element (a capacitor), and a load (a microcontroller). Unlike a battery, the ambient energy source has a high source resistance that limits its usable power, resulting in the characteristic "sawtooth" RC charging behavior shown in Figure 2B. The system charges its capacitor until there is enough energy and voltage to operate. Then, active operation begins and the

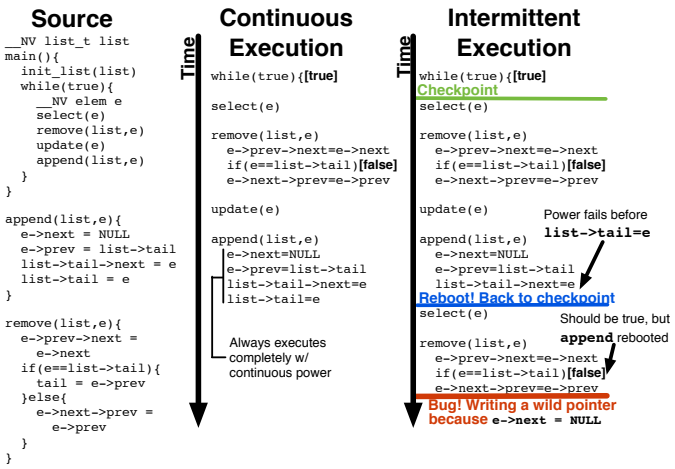

Figure 3: An intermittence bug. The linked-list stays correct with continuous power but is corrupted and leads to a wild pointer write with intermittent power.

capacitor starts to discharge (regions highlighted in green). The device continues actively until the voltage on the storage capacitor drops below a minimum operating threshold (dashed red line), at which point the system loses power and begins another charging cycle. This repeated charging and discharging of the device forces software into an *intermittent execution model* [18] where periods of powered execution are interspersed with reboots.

Figure 3 illustrates how intermittence induces bugs even with runtime support for checkpointing volatile state into non-volatile memory [11, 20, 24]. The code manipulates a linked-list in non-volatile memory using `append` and `remove` functions. A continuous execution completes the code sequentially, as expected. An intermittent execution, however, is not sequential. In the leftmost trace, a checkpoint happens to be collected at the top of the `while` loop and the processing continues until power fails at the indicated point (cf. Figure 2). After the reboot, execution resumes from the checkpoint. This sequence of events later leads to undefined behavior. The execution violates the pre-condition 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. This `NULL` pointer dereference cannot happen in a continuous execution and is an example of an *intermittence bug*.

2.2 Energy-interference during Debugging

Debugging intermittence bugs, like the one in Figure 3, using existing tools is virtually impossible. Conventional debuggers supply power to the device-under-test (DUT), which precludes observation of a realistically intermittent execution. Dedicated debugging equipment, like a JTAG [1] debugger, offers visibility into the device's state but is not useful because it provides continuous power and masks intermittence. JTAG power isolator devices [27] exist to decouple debug host power rails from DUT power rails, but these do not help with intermittence debugging, because the JTAG protocol fails if the DUT powers off. The inapplicability of JTAG precludes interactive debugging (e.g., like GDB or LLDB) for intermittent executions. Using a JTAG debugger for the code in Figure 3 would only ever result in the non-failing, continuous execution shown in the middle; the programmer would never see unexpected behavior.

One mostly energy-interference-free tool that can be used for debugging intermittent systems to a limited extent is a mixed-signal oscilloscope. A scope can collect an energy trace by probing DUT's power system and I/O lines. Unfortunately, a scope provides no insight into the internal state of the software running on the DUT. Scope-based debugging is not the interactive process familiar to most programmers. Moreover, oscilloscopes cost thousands of dollars, making them inaccessible to most developers. Using a scope to debug the code in Figure 3 would permit the problematic intermittent execution. However, the scope would not help relate changes in the device's energy state (which it can observe) to the software events that change device state and memory (which it cannot observe). That absent connective information is the key to understanding the failure in this code.

An alternative approach to diagnosing an intermittence bug is to directly write debugging instrumentation code into an application to trace certain program events. In embedded systems, a popular *ad hoc* approach is to toggle an LED at a point of interest. LED-based tracing does not work in energy-harvesting devices, because LEDs are power-hungry and their energy use changes the execution's behavior. As a case in point, it is prohibitively expensive to use an LED to indicate when a WISP energy-harvesting device [25] is executing code, rather than just charging. Powering an LED increases the WISP's current draw by five times, from around 1mA to over 5mA.

Another tracing strategy is to manually instrument code to log 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. Non-volatile tracing also consumes precious non-volatile storage space. To spare consuming non-volatile storage space, a programmer may write code 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 is expensive in time and energy and adds considerable complexity to code.

All of these instrumentation-based approaches change the point in the program at which energy is exhausted. As a result, the act of debugging alters the intermittent behavior of the application. Furthermore, the value of tracing depends

on the events which the programmer decides to trace. To understand the intermittence bug in Figure 3, the programmer needs to log particular events in the `append` and `remove` routines. The bug manifests as a wild pointer write and may appear to crash inexplicably, in code far from the either of those routines, giving little to suggest that `append` and `remove` contain the culpable code.

Energy-interference and lack of visibility into intermittent executions makes prior approaches to debugging inadequate for intermittence debugging.

3. EDB: Energy-interference-free Debugging

EDB is an energy-interference-free platform for intermittence debugging that addresses the shortcomings of existing approaches described in Section 2. This section describes the high-level capabilities and functionality of EDB, while Section 4 describes the co-designed hardware and software implementation that make EDB energy-interference-free.

Figure 4 illustrates EDB functionally. At the top are EDB's *capabilities* that together support the *debugging primitives* at the bottom. The functionality is organized into two parts. The first part is support for *passively monitoring* a device's energy level, program events and I/O, which we call EDB's "passive mode" of operation. The second part is a complementary "active mode" with support for *actively monitoring and manipulating* the target's energy level and internal state (e.g., registers and memory). We combine passive and active mode capabilities, to implement energy-interference-free debugging primitives, including energy and event tracing, intermittence-aware breakpoints, energy guards for instrumentation, and interactive debugging.

3.1 Passive Mode Operation

EDB's passive mode operation is built around the three rightmost components at the top of Figure 4. The developer gets the ability to acquire a set of streams and relay them to the host workstation continuously in real-time without active involvement from the target whether it is on or off. Streams instrumental for debugging are the energy level, I/O events on wired buses, messages exchanged over RFID protocol, and program events marked by *watchpoints* in application code. A key advantage of EDB is the ability to gather this data concurrently, letting the developer correlate changes in system behavior with changes in energy state. That correlation is important during development, but, as Section 2 describes, difficult or impossible using existing techniques.

3.2 Active Mode Operation

The capability to manipulate the amount of energy stored on the target device underlies EDB's active mode of operation. Active mode frees debugging tasks from the constraint of the small energy store of the target device. EDB can *compensate* for the energy consumed by a debugging task that involves a costly operation on the target, such as interacting with the

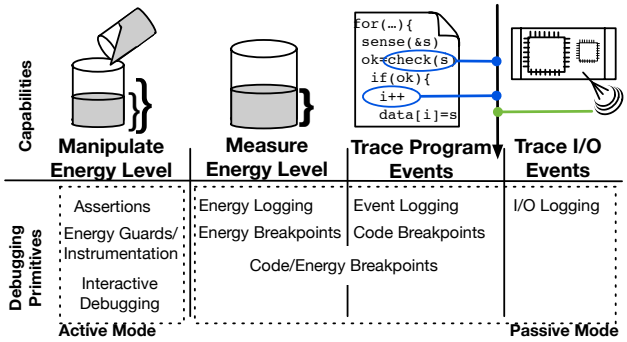

Figure 4: EDB's features and supported debugging tasks.

programmer, executing arbitrary debug code, or conveying state to the debugger. Before performing an active task the energy on the target device is measured and recorded. While the active task executes, the target is continuously powered. After performing the active task, energy on the target device is restored to the level measured before the active task. Continuously powering active tasks enables them to consume arbitrary amounts of energy. Energy compensation provides the illusion of an unaltered, intermittent execution to the application. Without this support, debugging tasks that require considerable involvement from the target are out of reach.

3.3 Energy-interference-free Debugging Primitives

Using the monitoring and manipulation 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. New intermittence-aware primitives are introduced to handle debugging tasks that arise only on intermittently-powered platforms.

3.3.1 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 help catch energy leaks due to unexpected code paths. They initiate an interactive debugging session precisely in problematic iterations when more energy was consumed than expected or when the device is about to brown-out.

3.3.2 Keep-alive Assertions

EDB provides support for using familiar *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 browning out. This *keep-alive* feature turns what would have to be a post-mortem reconstruction of events into an investigation on a live device. A

post-mortem analysis is limited to scarce clues in a tiny ad hoc "core dump" that a custom fault handler can manage to save into non-volatile state before the target runs out of energy and resets. The clues available in the interactive session that is automatically opened by EDB for a failing `assert` include the entire live target address space and I/O buses to peripherals.

3.3.3 Energy Guards

EDB can hide the energy cost of an arbitrary region of code if enclosed between a pair of *energy guards*. Code within energy guards executes on tethered power. Code on either side of an energy-guarded region experiences an illusion of continuity in the energy level across the energy-guarded region as if no energy was consumed. EDB implements energy guards using its energy compensation mechanism by recording the target energy level upon entering an energy guard and restoring it upon exiting the guard. Without energy cost, *instrumentation code* becomes non-disruptive and therefore useful on intermittent platforms. Two especially valuable forms of instrumentation impossible without EDB are *complex data structure invariant checks* and *external event tracing*. Extra code added to an application to check invariants on data structures or report when certain events have executed via I/O (e.g. `printf`, LED) can be costly enough to repeatedly deplete the target energy supply and prevent forward progress.

Besides instrumentation, EDB energy guards may also help incorporate non-intermittence-safe third-party code into intermittent applications. As long as third-party library calls are wrapped in energy guards, intermittence failures are guaranteed to not occur within the library. Functionality can now be developed separately from handling intermittence. Similarly, energy guards are useful for gradually porting code from a continuously powered platform. A programmer can start with an energy guard around the entire program and repeatedly exclude a module from the guarded region after verifying its correctness under intermittence, until the entire application is out of the guarded region and intermittence-safe.

3.3.4 Interactive Debugging

EDB supports interactive debugging of a target from a workstation. An interactive session provides full access to view and modify the target's memory, as in a conventional debugger (e.g., GDB, LLDB). A developer can enable code-energy breakpoints and can manually manipulate the target's energy level. An interactive session is entered automatically when a breakpoint is hit or an assertion fails or on demand by a console command. A unique benefit of EDB is its ability to trigger a manipulation of the target's energy state based on the target's program behavior and vice versa.

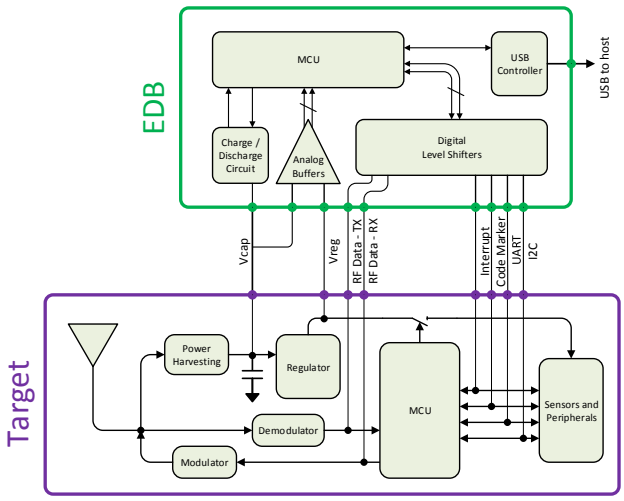

Figure 5: Block diagram of EDB connected to an RF energy-harvesting target. All signal lines are buffered to minimize energy interference. A charge/discharge circuit controls the voltage on the target's energy storage capacitor.

4. Hardware/Software Implementation

EDB's capabilities and debugging primitives are implemented in custom co-designed hardware and software. Figure 5 shows a block diagram of EDB (depicted in green) connected to an RF energy harvesting target (in purple). The labeled wires are physical connections between EDB and the target that carry both analog and digital signals and are exposed through header pins. Our prototype hardware board can connect to any energy-harvesting device with a microcontroller and a capacitor. To support a new device, the applicable physical connections from Figure 5 must be wired and target-side EDB software library of 1200 lines of C code must be ported to the new architecture.

4.1 Energy Level Monitoring

Energy-interference-free measurement of the target's energy level is essential to EDB's passive mode operation (Section 3.1). To measure a device's energy level, EDB uses two physical connections, *Vcap* and *Vreg*, to the target device's energy storage capacitor and its regulated power line, respectively. These signals pass through a dual high impedance, unity gain instrumentation amplifier to minimize leakage current from the target to EDB. These analog voltages are digitized by an analog to digital converter (ADC) and logged or used internally for debugging tasks. While it is *possible* for energy harvesting devices to measure their stored energy levels, doing so uses energy, perturbing the energy state being measured and altering software's intermittent behavior.

4.1.1 Energy Manipulation and Compensation

Energy manipulation and compensation are the basis for EDB's active mode of operation (Section 3.2). EDB has a custom circuit consisting of a low pass filter, keeper diode, and GPIO pins that can charge and discharge the target's

energy storage capacitor. This circuit is designed to prevent it from loading down or trickle charging the target while inactive (i.e., in a high impedance state).

To charge the target to a desired voltage level, EDB activates a GPIO pin to raise the voltage on the energy storage capacitor. A basic iterative control loop in EDB's software ensures that the voltage converges to the desired level. Discharging works similarly: the target's energy storage capacitor discharges through a fixed resistive load and a software control loop ensures convergence to the desired level. In our prototype, the charging circuit assumes a capacitive storage element, but with software changes, the same design can support other storage media, such as thin-film batteries.

4.1.2 I/O Monitoring

EDB is designed to enable passive monitoring of arbitrary I/O and attached peripherals, such as sensors, communication buses, and radios. These digital signals (labeled RF Data Tx/Rx, UART, and I2C in Figure 5) connect to a digital buffer and level shifter. We use an extremely low-leakage buffer to prevent leakage current from the target to EDB, and we use the level shifter to match the buffer's voltage level to the target device's voltage level.

Note that while the target device has an on-board regulator, the *Vreg* line may drop below its specified, regulated value during a power failure on the target device. We address the *Vreg* drop with a simple tracking circuit consisting of an analog buffer to keep the level shifter at the target's voltage. This circuit is important because too large a mismatch (i.e., over +/-0.3V [30]) activates the voltage protection diodes in the target's MCU, which perturbs the target's power state.

Our prototype can monitor GPIO, UART, I2C, and RFID RX/TX data lines. A key benefit of EDB is that it monitors data communication lines externally. With external monitoring, messages (e.g., RFID messages) can be decoded even if the target does not correctly decode them due to power failures. EDB's I/O monitoring support aids developers in I/O calibration and debugging I/O related issues in software.

4.1.3 Program Event Monitoring

EDB can track program execution using the *Code Marker* connections in Figure 5. To monitor a code point, the programmer inserts a watchpoint with a unique identifier at that location. EDB's target-side software encodes this identifier onto the Code Marker lines when the program counter passes over the code point. On the debugger-side, transitions on the Code Marker lines are captured and decoded into watchpoint identifiers. EDB can simultaneously monitor $2^n - 1$ distinct watchpoints, where n is the number of GPIO lines allocated to the Code Marker function.

Monitoring program events using EDB is *practically* energy-interference-free. The main energy cost is the target device holding a GPIO pin high for one cycle to encode each traced code point as it executes. We measured the cost of this GPIO-based signaling to be negligible using the methodology described in Section 5. Without EDB, monitoring has a prohibitive cost in code, memory space, and energy. With EDB, events are not only logged without these costs, but also correlated with energy state into a multifaceted profile.

`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,watchpoints*}
`printf(fmt, ...)`	`read`|`write` *address* [*value*]

Table 1: Developer's interfaces into EDB.

4.2 Developer's Interfaces into EDB

EDB's debugging primitives are accessible to the end-user through two complimentary interfaces: the `libEDB` API and the host console commands listed in Table 1. The `libEDB` library statically links into the application and exports C macros for inserting assertions, breakpoints, watchpoints, energy guards, and energy-interference-free `printf` calls into the application code. Internally, the library implements the target-side half of the protocol for communicating with the debugger over a dedicated GPIO line and a UART link, which includes routines for reading from and writing to target address space.

The debug console is a command-line interface for interacting directly with EDB and indirectly with the target over a USB connection from a workstation. During interactive debugging in active mode, the console reports assert failures and breakpoints hits and provides commands to inspect target memory. During passive mode debugging, the console delivers traces of energy state, watchpoint hits, monitored I/O events, and the output of `printf` calls. EDB can emulate intermittence at the granularity of individual charge-discharge cycles using the `charge`/`discharge` commands.

4.3 Implementation Details and Release

We prototyped EDB as a printed circuit board (PCB) that connects to the target device via a board-to-board header. Our core design is also compatible with an implementation as an on-chip component within the target device architecture. EDB software includes 5600 lines of C code for firmware, 1200 lines of C code for `libEDB`, and 1200 lines of Python code for scripting API and host console.

5. Evaluation

The purpose of our evaluation is two-fold. First, we characterize potential sources of energy interference and show that EDB is energy-interference-free with detailed measurements. Second, we use a series of case studies conducted on a real energy-harvesting system to demonstrate that EDB supports monitoring and debugging tasks that are impossible with existing tools.

5.1 Experimental Setup

Our experimental setup consists of the EDB prototype board, a target energy-harvesting device and wireless en-

ergy source, and measurement instruments. The EDB board is always connected to a development workstation via USB and to the target device through a dedicated header. EDB is controlled from the workstation programmatically or manually through the console described in Section 4.2.

Our target device is a Wireless Identification and Sensing Platform (WISP) version 5 [25]. The WISP has a 47 μF energy storage capacitor, a turn-on threshold of 2.4 V, a brown-out threshold of 1.8 V, and an active current of approximately 0.5 mA at 4 MHz. The WISP is intermittently powered by RF radiation from an Impinj Speedway Revolution RFID reader. The reader is configured to continuously inventory tags at a transmit power of up to 30 dBm using the SLLURP toolset [3], and its antenna is placed at a distance of 1 m from the WISP. The amount of harvestable energy is inversely proportional to this distance. In our evaluation, we ran a collection of different software applications on the target device. We used a custom test program that manipulates non-volatile linked-list, and another that generates a persistent list of Fibonacci numbers. We also used two real applications, including the official WISP 5 RFID firmware, and a machine-learning-based activity recognition application from prior work [18].

To validate and characterize EDB— especially its energy-interference-freedom — we used some additional measurement equipment that is not normally necessary when using EDB. We collected data in the evaluation using a Tektronix MDO4104 oscilloscope and a Keithley 2450 SourceMeter. The oscilloscope channels were connected to the analog and digital lines between EDB and the WISP in order to record the capacitor voltage, moments when active debug mode starts and ends, and events that trace application progress.

5.2 Energy-interference

EDB's edge over existing debugging tools is its ability to remain isolated from intermittent power in passive mode and to create an illusion of an untouched energy reservoir in active mode. Next, we characterize these two modes of energy-interference and show with measurements that neither compromises EDB's energy-interference-freedom.

5.2.1 Current flow over electrical connections

Any current that flows between the target and the debugger through the connections in Figure 5 may inadvertently charge or discharge the capacitor on the target. As discussed in Section 4, EDB's circuits are designed to minimize the amount of current that can flow across any of these connections to or from the target's power supply. Imperfections in components, such as reverse leakage current in the diodes, inevitably cause some current to flow. We measured the maximum possible current flow over each connection and verified that in the absolute *worst-case*, when all lines are active, the effect it can have on the target power supply is negligible.

We used a source meter to apply a voltage to the driving endpoint of each connection and measure the resulting current. We measured each connection with digital logic endpoints in both *LOW* and *HIGH* states by applying either 0 V or 2.4 V, which is the maximum voltage that can arise on any of the connections. We measured analog endpoints under the worst-case condition of 2.4 V. The sum of the worst-case current flow in either direction across all connections is 0.85 μA or 0.2% of the typical active mode current consumption of the MCU in our target device. Table 2 characterizes the energy interference, showing a breakdown of worst-case current by connection, driving endpoint, and logic state.

Debuger ↔ Target Connection		**DC Current (nA)**		
		Min	Avg	Max
Capacitor sense, manipulate		-2.5100	0.1445	0.8300
Regulator sense, level reference		-0.0300	-0.0029	0.0100
Debugger→Target comm.	high	-0.0200	-0.0004	0.0100
	low	-0.0300	-0.0200	-0.0100
Target→Debugger comm.	high	-0.0200	62.9349	108.2300
	low	-1.9200	-1.7982	-1.7100
Code marker (x2)	high	-0.0200	63.7853	111.5400
	low	-2.1600	-1.9770	-1.8300
UART RX	high	-0.0100	64.8042	111.2600
	low	-2.5500	-1.8909	-1.7200
UART TX	high	-0.0000	66.3433	139.8800
	low	-1.7900	-1.6705	-1.5600
RF RX	high	-0.0400	66.0402	115.0100
	low	-2.3000	-2.1271	-1.9900
RF TX	high	-0.0200	66.5382	117.9600
	low	-2.7300	-2.2726	-2.1600
I2C SCL	high	-0.0400	0.0358	0.0800
	low	-0.3200	-0.1780	-0.1500
I2C SDA	high	-0.0100	0.0367	0.0700
	low	-0.2800	-0.1754	-0.1400
Worst-Case Total Current				836.51 nA

Table 2: Measured worst-case current that can flow over electrical connections between the target device and EDB.

5.2.2 Accuracy of manipulating target energy level

To support debugging tasks presented in Section 3, EDB needs to save, change, and restore the amount of charge in the target's storage capacitor. A large discrepancy between the saved and restored level can undermine the illusion of an unaltered intermittent power supply that EDB presents to the target software. We quantified this energy level discrepancy, ΔE, by measuring the voltage on the capacitor before and after a save-restore operation and applying the expression for the energy stored in a capacitor: $\Delta E = \frac{1}{2}C(V_{\text{restored}}^2 - V_{\text{saved}}^2)$. This quantity was then expressed as a percentage of the maximum energy storable on the target: $\Delta\hat{E} = \Delta E/(\frac{1}{2}CV_{\text{max}}^2)$, where $V_{\text{max}} = 2.4$ V.

We used the charge/discharge commands introduced in Section 4.2 to run 50 trials of a save-restore operation. For each trial, we set an energy-breakpoint at 2.3 V, charged the target capacitor to 2.4 V, waited for the target execution to be interrupted by the breakpoint, and then resumed the target. Table 3 summarizes two independent sets of measurements of $\Delta V = V_{\text{restored}} - V_{\text{saved}}$, ΔE, and $\Delta\hat{E}$: one from our oscilloscope and one from EDB's ADC. The accuracy of EDB's save-restore mechanism, $\Delta\hat{E}$, in our prototype implementa-

	ΔV (mV)		ΔE (μJ)		$\Delta \hat{E}$ (%*)	
	O-scope	ADC	O-scope	ADC	O-scope	ADC
Mean	54	55	1.25	1.25	4.34	4.34
S.D.	16	7.8	0.37	0.18	1.30	0.62

* Energy cost is reported as percentage of 47μF storage capacity.

Table 3: Accuracy with which EDB saves and restores energy level quantified as the difference in capacitor charge before saving and after restoring and measured using either an external oscilloscope or the internal ADC in EDB.

tion of EDB is, on average, 4.34% of the target's $47\mu F$ energy storage capacitor. Our prototype's energy level discrepancy is small enough that it is unlikely to be problematic. We expect that further software optimization will leave a discrepancy closer to the accuracy limit imposed by EDB's ADC. A 12-bit ADC with effective resolution of approximately 1 mV imposes a theoretical lower bound on $\Delta \hat{E}$ of 0.08%.

5.3 Debugging Capabilities

We now illustrate the new capabilities that EDB brings to the development of intermittent software by applying it to debugging tasks that are particularly difficult to resolve using state-of-the-art tools. Energy-harvesting applications in the following case studies execute intermittently and keep state in non-volatile memory to make progress without relying on a runtime checkpointing system. A reboot causes execution to return to the program entry point (i.e., `main`).

5.3.1 Detecting memory corruption early

Memory corruption due to incorrect pointer arithmetic or a buffer overflow is a frequent yet difficult problem to debug. The root cause is obscured behind symptoms that are far from the offending memory write in time and in space. Memory corruption induced by intermittence is harder to diagnose still, because it is not reproducible in a conventional debugger as discussed in Section 2.2. This section studies an application that fails due to an intermittence-induced memory corruption and demonstrates how EDB's support for assertions exposes the root cause.

Application. The code listed in Figure 6 maintains a doubly-linked list data structure in non-volatile memory. On each iteration of the main loop, a node is appended to the linked list if the list is empty or removed from the list otherwise. The node is initialized with a pointer to a buffer in volatile memory. This pointer is retrieved when the node is removed from the list and data is written to the buffer it points to.[1] For illustrative purposes, at the beginning and end of the loop iteration, the code toggles a GPIO pin to indicate that the main loop is running.

Symptoms. After having run on harvested energy for some amount of time, the GPIO pin indicating main loop progress stops toggling. The real oscilloscope trace in the top of Figure 7 shows an early charge-discharge cycle when the main loop is still executing and a later one when it no longer does. After the main loop stops executing, the application never returns to normal, including after reboots on subsequent charge-discharge cycles. The only way to recover is to re-flash the device. Note that the failure problem never occurs when the device runs on continuous power.

Diagnosis. Since the broken final state persists across reboots, one approach is to attach a conventional debugger after the failure and attempt to determine why the main loop stopped running. This approach may help uncover the symptom, but not the root cause, because the information that happens to persist in memory may not be sufficient to follow the chain of events backwards in time. A better approach is to catch the problem at its source by `asserting` an invariant on the linked-list data structure whenever it is manipulated. However, conventional assertions fall short in this case, because they let the target drain the energy supply, reset, and continue past a failed assertion.

EDB's intermittence-aware `assert` mechanism is designed to tackle this class of bugs. We assert the invariant that the tail pointer points to the last element in the list as shown in Figure 6 and run the program on harvested energy with EDB attached. EDB's console reports the assertion failure, halts the program, starts continuously powering the target, and opens an interactive debug session. This sequence is captured in the bottom oscilloscope trace in Figure 6. The discharge cycle on the right is the one during which the assert fails at instant 1 and the capacitor voltage is seen rising to the level of the tethered power supply.

In the interactive debug session summarized on the right in Figure 6, we check the device's internal state using EDB's commands for inspecting target memory. The tail pointer points to the penultimate element instead of the last one, which is a consequence of an `append` interrupted by intermittence. Because of this inconsistency, the `else`-clause in the `remove` function would dereference a `NULL` pointer, read the buffer pointer from an invalid location, and cause `memset` to write to a wild pointer and corrupt non-volatile state beyond recovery. The `assert` and the interactive session uncovered the precise inconsistency in the data structure before any of these confounding consequences could take place.

5.3.2 Instrumenting code with consistency checks

To aid in debugging, applications often have separate *debug* and *release* build configurations. A debug build includes *instrumentation* code such as checks for consistency of data structures or array bounds. On continuously-powered platforms the convenience of the debug build comes at the cost of slower execution speed, higher memory usage, and higher energy consumption. However, on intermittently-powered platforms, the effect is more dire: the energy overhead of

[1] The role of the memory buffer in this example is to expose undefined behavior during access to the linked list, which takes place with or without the buffer, as an externally observable failure.

Application Code

```
1: init_list(list)
2: while (1)
3:  node = list->head
4:  while (node->next != NULL)
5:    node = node->next
6:  assert(list->tail == node)
7:  if (node == list->head)
8:    init_node(new_node)
9:    append(list, new_node)
10: else
11:   remove(list, node, &bufptr)
12:   memset(bufptr, 0x42, BUFSZ)
```

Debug Console

```
> run
Interrupted:
ASSERT line: 8
Vcap = 1.9749

*> print node
0xAA10: 0x00BB
*> print list->tail
0xAA20: 0x00AA
*> print list->tail.next
0xAA30: 0x00BB
```

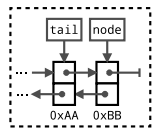

Figure 6: An intermittence bug that corrupts memory, diagnosed using EDB's intermittence-aware `assert` (left) and interactive console (right).

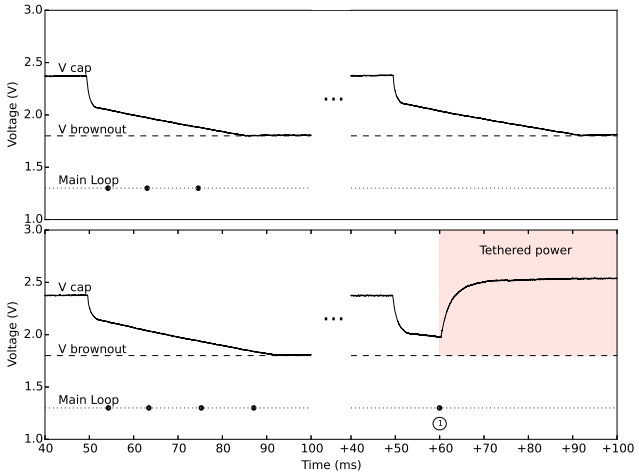

Figure 7: Oscilloscope trace of a memory-corrupting intermittence bug and EDB's intermittence-aware `assert` in action. Without the `assert` (top) the main loop runs at first (left) but mysteriously stops running in later discharge-cycles (right). With the `assert` (bottom), when it fails at instant 1, EDB halts the device and tethers it to a continuous power supply.

instrumentation can render an application non-functional by preventing it from making *any* forward progress. Yet, instrumented energy-harvesting applications must be run on harvested energy to diagnose intermittence-induced bugs, since these bugs are invisible while the device is continuously powered. In this case study we demonstrate how an application can be instrumented with debug code of arbitrary energy cost using EDB's energy guards.

Application. The code in Figure 8 generates the Fibonacci sequence and appends each number to a non-volatile, doubly-linked list. For illustrative purposes, each iteration of the main loop toggles a GPIO pin to track progress. In the debug build, `main` begins with an energy-hungry consistency check that traverses the list and asserts that the previous and next pointers and the Fibonacci value in each node are consistent. This invariant helps detect problems early before they precipitate into mysterious failures akin to the one in Section 5.3.1. With intermittent power, the invariant was violated in several experimental trials.

Symptoms. The application's release build produces an inconsistent list without any indication that there is a problem.

```
1: main()
2:  energy_guard_begin()
3:  for (node in list)
4:    assert(node->prev->next == node == node->next->prev)
5:    assert(node->prev->fib + node->fib == node->next->fib)
6:  assert(list->tail == node)
7:  energy_guard_end()
8:  while(1)
9:    append_fibonacci_node(list)
```

Figure 8: Application code instrumented with a consistency check of arbitrary energy cost using EDB's energy guards.

The debug build stops executing the main loop after having added approximately 555 items to the list. The top trace in Figure 9 shows an early charge cycle when the main loop executes and a later one when it no longer does.

Diagnosis. The energy cost of the consistency check is proportional to the length of the list. Once the list is long enough, the consistency check consumes all the energy available in one charge-discharge cycle and leaves none for the main loop. Once reached, this hung state persists indefinitely because the application cannot make progress in subsequent charge-discharge cycles.

EDB lets the developer keep the consistency check without breaking application functionality by wrapping the check with energy guards as shown in Figure 8. The effect this has on the target energy state is captured in the bottom oscilloscope trace in Figure 9. At instance 1, the target enters the energy guard, and EDB tethers it to a power supply. The capacitor starts charging, while the target continues executing the code within the energy guard. At instance 2, the target exits the energy guard, and EDB cuts the power supply and starts to discharge the capacitor to the level it had at instance 1. After the discharge completes, the target is allowed to continue. This sequence of events later takes place again between instances 3 and 4. With the energy guard around the consistency check, the main loop gets the same amount of energy in both early charge-discharge cycles when the list is short (left) and later ones when it is longer (right).

5.3.3 Tracing events and profiling energy cost

Intermediate results of calculations, frequency of events, and energy cost of operations are valuable clues for quick diagnosis of erroneous code. Directly extracting such information from an energy-harvesting device using existing tools changes the application's behavior. For example, the sample rate of a sensing application may increase by a factor of 100-1000x when powered continuously in the lab relative to when harvesting energy in a realistic deployment. This section demonstrates how EDB's energy-interference-free `printf` and watchpoints can peek under the hood of running code with minimal impact on application behavior.

Application. The activity recognition application outlined in Figure 10 reads an accelerometer sample, classifies the sample as "stationary" or "moving", and records statistics in non-volatile memory.

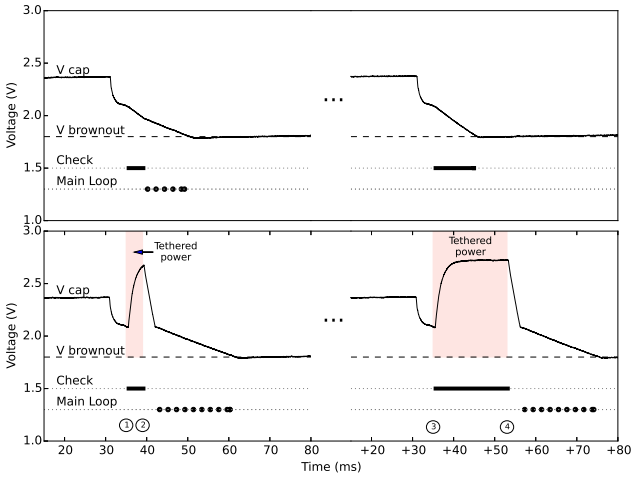

Figure 9: Oscilloscope trace of an application instrumented with a consistency check of high energy cost. Without an energy guard (top), the check and main loop both execute at first (left) but only the check executes in later discharge-cycles (right). With an energy guard (bottom), the check executes on tethered power from instant 1 to 2 and 3 to 4, and the main loop always executes.

Symptoms. There is no evidence that the recorded statistics are based on correct accelerometer readings and classification results. Moreover, the application cannot be tuned to the size of the storage capacitor without the energy profile of one classification operation.

Diagnosis. Information can be extracted from the target device either over traditional debugger interface (e.g. JTAG) or I/O peripherals (e.g. UART or GPIO ports). To relay a data stream via a JTAG debugger, the target device must be on during the entire debugging session. Off-the-shelf USB-to-serial adapters are not electrically isolated from the target UART and permit energy to flow into or out of the device. Encoding information onto GPIO pins and decoding it using an oscilloscope costs pins and significant effort compared to a `printf` call that outputs text to a console on the host.

The measurements in Table 4 demonstrate the impact on application behavior of using UART. The energy cost of the print statement changes the iteration success rate, i.e. the fraction of iterations that successfully complete out of the total attempted. To trace application progress without disrupting its behavior, we instrumented the loop body with an EDB `printf` and three watchpoints as shown in Figure 10. The `printf` produces a stream of intermediate classification results for each iteration. The watchpoints produce a time and energy profile of a loop iteration as well as an independent calculation of the statistics that is useful for manual verification. The energy profile shown in Figure 11 was calculated from the difference between energy level snapshots taken by watchpoints 1 and 2, and watchpoints 1 and 3. Reference classification statistics can be calculated by counting occurrences of watchpoints 2 and 3.

```
 1: main()
 2:   while (1)
 3:     watchpoint(1)
 4:     sample = read_accelerometer()
 5:     class = classify(featurize(sample))
 6:     switch (class)
 7:       case STATIONARY: count[STATIONARY]++
 8:                        watchpoint(2)
 9:       case MOVING:     count[MOVING]++
10:                        watchpoint(3)
11:     total++
12:     printf("t %u s %u m %u\n", total
13:            count[STATIONARY], count[MOVING])
14:     stats[STATIONARY] = count[STATIONARY] / total
15:     stats[MOVING]     = count[MOVING]     / total
```

Figure 10: Tracing and profiling an activity recognition application using EDB's energy-interference-free `printf` and watchpoints.

	Iteration Success Rate	Iteration Cost		Print Cost	
		Energy (%*)	Time (ms)	Energy (%*)	Time (ms)
No print	87%	3.0	1.1	-	-
UART `printf`	74%	5.3	2.1	2.5	1.1
EDB `printf`	82%	3.4	4.7	0.11	3.1

* Energy cost is reported as percentage of 47μF storage capacity.

Table 4: Cost of debug output and its impact on the behavior of the activity recognition application.

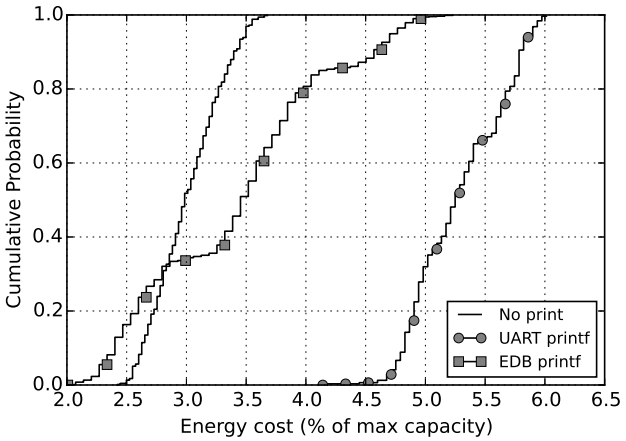

Figure 11: Energy profile of one loop iteration in the activity recognition application when instrumented with different output mechanisms.

5.3.4 Debugging and tuning RFID applications

Energy-harvesting applications that communicate over the RFID protocol are difficult to debug without simultaneous visibility into communication and energy state. This case study demonstrates how EDB can monitor RFID I/O messages and correlate them with available energy.

Application. The WISP RFID firmware [31] decodes RFID query commands from a reader in software and replies with a unique identifier.

Symptoms. The application and reader cannot be characterized and tuned without a measure of the target's performance in different RF environments, e.g. the number of responses per queries received. Correctness cannot be verified without

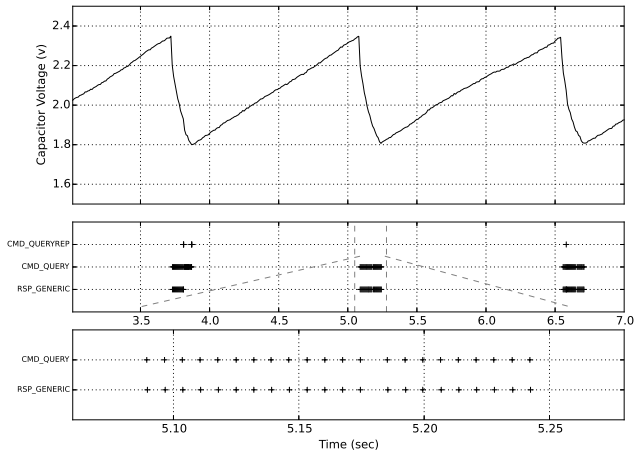

Figure 12: Incoming and outgoing RFID messages correlated with energy level recorded by EDB.

evidence that the application software successfully decodes and acts on each valid incoming query message.

Diagnosis. Both tasks require a trace of incoming messages that reached the target, i.e. bit patterns in the incoming demodulated waveform that *could* have been decoded into valid messages by software. An oscilloscope trace of the raw output from the RF demodulator does not reveal whether the waveform is decodable into a valid message. A decoder is necessary to separate messages that were corrupted in flight from valid messages that the target application failed to parse.

We use EDB to stream RFID message identifiers and target energy readings to the host. From data plotted in Figure 12 we find that in our lab setup the application responded 86% of the time for an average of 13 replies per second. The view focused on one discharge cycle confirms that the application successfully received consecutive incoming query messages and replied. 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 left after message decoding and response transmission.

6. Related Work

We discuss several areas of prior work that are especially related to EDB. Some recent work proposes models and emulators for energy-harvesting environments. Much effort is dedicated to improving the reliability of energy-harvesting systems. Earlier work explored debugging in continuously powered sensor nodes.

6.1 Modeling and emulation of energy sources

Ekho [9] is a device that records the amount of energy harvested by a harvesting circuit and reproduces the trace as power input into an application device. Ekho can reproduce problematic program behavior, but it cannot offer insight into this behavior. Complementary to Ekho's features, EDB offers debugging mechanisms for inspecting the program state and correlating program events to the energy level.

Application behavior on an intermittent energy source can be partially inferred from a simulation. Computational RFID Crash Test Simulator (CCTS) [8] can produce a voltage trace representative of a solar harvester with a specified capacitor size and load. CCTS is useful for exploring the design space for a new energy-harvesting application, but not for *in situ* debugging tasks that EDB is designed for.

6.2 Reliability in Intermittent Devices

Prior work pointed out that intermittent execution threatens forward progress [24] and memory consistency [18, 23]. There are three main approaches taken by prior work to combat these threats to reliability. The first approach is to *tolerate intermittent failures* by selectively capturing and restoring program state. The second is to *avoid intermittent failures* through aggressive duty cycling and scheduling. The last is to *eliminate intermittence* with non-volatile hardware.

EDB is related to these approaches because all of them aim to improve correctness of code and reliability of intermittent executions. EDB, however, is largely orthogonal to these approaches as they are mostly changes to the programming or execution model. Instead, the purpose of EDB is to give a programmer visibility into a system to *understand* why intermittence is causing problems, even in emerging programming and execution models.

Tolerating Intermittent Failures. Mementos [24] first equipped energy-harvesting computers to make progress through long-running workloads. Mementos checkpoints volatile state into the device's non-volatile state to preserve execution context and data across power failures. QuickRecall [11] took a similar approach with hardware support and Idetic [20] applied the idea to ASICs.

More recent work [23] observed that even with measures to ensure progress, intermittent execution can leave a system's memory in an inconsistent state. DINO [18] characterized the *intermittent execution model* and addressed these consistency issues with a task-based programming and execution model that selectively preserves both non-volatile and volatile memory across power failures.

Avoiding Intermittent Failures. Other prior work aims to avoid failures due to intermittence by scheduling and duty cycling. Eon [28] is one of the first efforts at avoiding intermittence failures in a solar-powered device. Eon associates computational tasks with their expected energy consumption, and a scarcity of energy causes the system to deschedule expensive tasks. Dewdrop [4] is a scheduler that brings an RF-harvesting device [25] into and out of deep sleep states that consume little energy. Dewdrop schedules tasks based on the likelihood that they will successfully execute, given the available energy. Hibernus [2] assumes hardware with a large capacitor that stores energy during execution. When power fails, Hibernus uses the stored energy to check-

point volatile state and puts the device to sleep, preserving volatile state and avoiding some failures.

Eliminating Intermittent Failures. Other prior work aims to eliminate intermittence with pervasive non-volatile hardware. Several recent papers [17, 19] discussed different approaches to using microarchitectural non-volatility to prevent power intermittence from leading to intermittent software execution. These efforts require invasive hardware changes and operate more slowly than conventional hardware. However, by making microarchitectural state persistent, these systems prevent progress and consistency issues due to intermittence.

6.3 Debugging in Embedded Wireless Sensor Nodes

Prior to recent interest in energy-harvesting systems [13, 25], there was considerable interest in battery-powered wireless sensor nodes [10, 12]. Sensor nodes necessitated programming [5] and operating system [7, 14] support, which in turn created a need for development and debugging support.

Clairvoyant [32] is the closest work from this era to EDB because, like EDB it provides interactive debugging capabilities. The system tries to minimize its effect on the program being debugged, in terms of memory use, network traffic, and system lifetime. However, because Clairvoyant targets powered nodes, it does not (need to) address energy-interference-freedom. Additionally, Clairvoyant does not discuss features supported by EDB like assertions, energy manipulation, or instrumentation.

Sympathy [22] provides support for debugging *networks* of sensor nodes. The scope of Sympathy is restricted to determining why data collection nodes stop sending data to "sink" nodes in the network. This work uses a series of metrics and an inference step to isolate failures and is largely orthogonal in purpose and mechanism to EDB.

TinyTracer [29] supports lightweight event tracing for sensor node programs written in nesC [5]. Its traces enable execution replay and manual failure analysis. Like TinyTracer, EDB provides lightweight event tracing, although EDB does not trace all events, only marked ones. In contrast, EDB provides fine-grained energy tracing, as well as energy-interference-free active mode tasks. Moreover, EDB's instrumentation support may be able to make TinyTracer energy-interference-free.

T-Check [15] and KleeNet [26] use model checking and symbolic execution (respectively) to *expose* failures in sensor node programs. Both are orthogonal to EDB, as they do not support monitoring or interactive debugging. Additionally, these systems do not address intermittence; however, if we assume they could be re-engineered to work on intermittent systems, they would be complementary to EDB: A developer could use EDB's debugging capabilities to understand and fix failures that they expose.

7. Conclusion

Intermittently executing, energy-harvesting devices present unique system reliability challenges, and our work in EDB presents the first debugging system that is designed to address those challenges. We identified *energy-interference-freedom* as a property that is essential to the utility of a debugging platform for power intermittent systems and built EDB to espouse that property from its circuits to its software. EDB supports passive monitoring of a target device's energy, software events, and I/O. Using its ability to manipulate a target device's energy, EDB also supports active debugging tasks with energy-interference-freedom, including assertions, instrumentation, tracing, and interactive debugging. We evaluated our prototype of EDB, including custom hardware, showing that it is energy-interference-free in both its passive and active tasks, and that it provides invaluable debugging information that is out of reach using existing tools and techniques. We see EDB and its energy-interference-freedom as a key part of future support for reliability in intermittent energy-harvesting devices.

8. Acknowledgements

We thank the anonymous reviewers for their time and valuable feedback. Thanks to Preeti Murthy for beta-testing our EDB prototype. This work was generously supported by Disney Research Pittsburgh and National Science Foundation grant CNS-1526342.

References

[1] IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture. *IEEE Std 1149.7-2009*, pages 1–985, Feb 2010.

[2] 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.

[3] Ben Ransford. SLLURP - Python Client for LLRP-based RFID Readers. `https://github.com/ransford/sllurp`. Visited August 10, 2015.

[4] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003.

[6] S. Gollakota, M. S. Reynolds, J. R. Smith, and D. J. Wetherall. The emergence of RF-powered computing. *Computer*, 47(1), 2014. doi: http://dx.doi.org/10.1109/MC.2013.404.

[7] L. Gu and J. A. Stankovic. T-kernel: Providing reliable OS support to wireless sensor networks. In *Proceedings of the*

4th International Conference on Embedded Networked Sensor Systems, SenSys '06, pages 1–14, New York, NY, USA, 2006.

[8] J. Gummeson, S. S. Clark, K. Fu, and D. Ganesan. On the limits of effective hybrid micro-energy harvesting on mobile CRFID sensors. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 195–208, New York, NY, USA, 2010.

[9] J. Hester, T. Scott, and J. Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 1–15, New York, NY, USA, 2014.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 93–104, New York, NY, USA, 2000.

[11] 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.

[12] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 271–278, New York, NY, USA, 1999.

[13] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 402–404, Feb 2012.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Sensor Networks. 2004.

[15] P. Li and J. Regehr. T-Check: Bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 174–185, New York, NY, USA, 2010.

[16] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013.

[17] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, J. Shu, and H. Yang. Ambient energy harvesting nonvolatile processors: From circuit to system. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 150:1–150:6, New York, NY, USA, 2015.

[18] 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.

[19] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 526–537, Feb 2015.

[20] 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.

[21] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1): 18–27, 2005. doi: http://dx.doi.org/10.1109/MPRV.2005.9.

[22] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 255–267, New York, NY, USA, 2005.

[23] 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.

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

[25] 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.

[26] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 186–196, New York, NY, USA, 2010.

[27] SEGGER. J-Link JTAG Isolator . `https://www.segger.com/jtag-isolator.html`, 2015.

[28] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007.

[29] V. Sundaram, P. Eugster, X. Zhang, and V. Addanki. Diagnostic tracing for wireless sensor networks. *ACM Trans. Sen. Netw.*, 9(4):38:1–38:41, July 2013. ISSN 1550-4859.

[30] TI Inc. Overview for MSP430FRxx FRAM. `http://ti.com/wolverine`, 2014. Visited July 28, 2014.

[31] WISP. WISP - Firmware Repository for WISP 5.0. `https://github.com/wisp/wisp5`. Visited August 10, 2015.

[32] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 189–203, New York, NY, USA, 2007.