5-2017

Sophisticated Batteryless Sensing

Josiah D. Hester
Clemson University, jhester@clemson.edu

SOPHISTICATED BATTERYLESS SENSING

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Josiah D. Hester
May 2017

ABSTRACT

Wireless embedded sensing systems have revolutionized scientific, industrial, and consumer applications. Sensors have become a fixture in our daily lives, as well as the scientific and industrial communities by allowing continuous monitoring of people, wildlife, plants, buildings, roads and highways, pipelines, and countless other objects. Recently a new vision for sensing has emerged—known as the Internet-of-Things (IoT)—where trillions of devices invisibly sense, coordinate, and communicate to support our life and well being. However, the sheer scale of the IoT has presented serious problems for current sensing technologies—mainly, the unsustainable maintenance, ecological, and economic costs of recycling or disposing of trillions of batteries. This energy storage bottleneck has prevented massive deployments of tiny sensing devices at the edge of the IoT.

This dissertation explores an alternative—leave the batteries behind, and harvest the energy required for sensing tasks from the environment the device is embedded in. These sensors can be made cheaper, smaller, and will last decades longer than their battery powered counterparts, making them a perfect fit for the requirements of the IoT. These sensors can be deployed where battery powered sensors cannot—embedded in concrete, shot into space, or even implanted in animals and people. However, these batteryless sensors may lose power at any point, with no warning, for unpredictable lengths of time. Programming, profiling, debugging, and building applications with these devices pose significant challenges. First, batteryless devices operate in unpredictable environments, where voltages vary and power failures can occur at any time—often devices are in failure for hours. Second, a device's behavior effects the amount of energy they can harvest—meaning small changes in tasks can drastically change harvester efficiency. Third, the programming interfaces of batteryless devices are ill-defined and non- intuitive; most developers have trouble anticipating the problems inherent with an intermittent power supply. Finally, the lack of community, and a standard usable hardware platform have reduced the resources and prototyping ability of the developer. In this dissertation we present solutions to these challenges in the form of a tool for repeatable and realistic experimentation called **Ekho**, a reconfigurable hardware platform named **Flicker**, and a language and runtime for timely execution of intermittent programs called **Mayfly**.

DEDICATION

E lei kau, e lei hoʻoilo i ke aloha.

Dedicated to Emily

ACKNOWLEDGMENTS

This dissertation was not produced in isolation, but is a product of advice, encouragement, patience and even prodding from many people over the years. Specifically I want to acknowledge:

My advisor, Jacob Sorber, who provided years of ideas, encouragement, and of course, funding. Jacob counseled and collaborated on seemingly endless paper submissions, was there to celebrate the victories, and to talk through the failures. This dissertation would not exist without Jacob.

Professors Dave Kotz, Dan Holcomb, and Brandon Lucia, who collaborated with me and each taught me significant things about academia, job hunting, and the scientific pursuit. Professors Steve Stevenson, Melanie Cooper, and Roy Pargas, who initially got me interested in a scientific path. I am incredibly grateful for all their advice and support throughout my academic career.

All current and former members of the PERSIST Lab, Kyle McGuigan, Joe Scott, Nicole Tobias, Steven Hearndon, the many undergraduates, and Kevin Storer, with whom I have worked on many publications, commiserated on failed submissions, and celebrated accepted papers.

My parents, brothers, and sister, for supporting me my entire life, providing endless entertainment, and being excited with me. Especially my parents, for providing me with the means to get a fantastic education, and supporting my decisions even when they did not align with their own. Dad, Mom, I will always be indebted to you, and thank you for your love and encouragement.

My wife, Emily, for her endless patience, sacrificial love, boatloads of common sense, and exquisite taste in men. Emily, thank you for helping me through this crazy process, keeping me sane, and continuing to push me to do better. You are my best, and closest friend, I love you. Now onto the next adventure together.

Ahuwale ka nane hūnā

TABLE OF CONTENTS

Table of Contents (Continued) **Page**

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

The future of sensing lies in supporting the vision of the Internet of Things (IoT); where trillions of heterogeneous devices collect, and share data across many contexts, and for many applications, from infrastructure monitoring to wearable computing. This vision echoes the original formulation of "smart dust" [65], which advocated for millimeter scale sensors, deployed in a dense volume, for the benefit of our everyday lives. These "smart dust" devices are the *fundamental unit* of the IoT, enabling the dense deployments essential for this vision. These devices are the "swarm at the edge of the cloud[73]", the leaf nodes of a global sensor network. These sensors must harvest energy from the environment to achieve the long lifetimes necessary (often measured in decades), and to reduce the size, and cost, of a larger energy store. Currently, many sensors combine energy harvesting with a battery or super capacitor, however, the inclusion of these expensive storage mechanisms *is fundamentally at odds with the IoT vision.*

The sensors that make up the current iteration of the IoT, usually use batteries as the primary energy source. This has hampered their ability to be deployed for long periods of time. Batteries only have a usable lifetime of a few years if used carefully. Additionally batteries are expensive, large, and pose environmental risks when disposed[72, 100, 124]. When batteries fail, their replacement requires human intervention; something not possible at the scales of the future IoT. The problems with battery powered sensing have inspired a range of smaller, cheaper computing devices, *without batteries*, that can be deployed maintenance-free for decades. These devices are powered by the abundant supply of environmental energy which they store in tiny capacitors that are much easier to recycle—and cheaper to produce—than batteries. This combination of energy harvesting and batteryless computing promises to enable a wide range of new applications in the future Internet of Things. This new generation of batteryless sensing devices can be used for passive RF-powered interactions (re-imagining contactless smart cards, and RFID tags), active data gathering to monitor the health of a structure, sensing to better understand the behaviors of wildlife populations, and many other applications that require fire-and-forget sensing. Now very long term studies of infrastructure, wildlife, and human factors becomes realistic. However, implementing complex applications on batteryless technology is currently very difficult.

Wireless sensing systems generally operate under the assumption that power is a limited, but *stable*, resource. For batteryless devices, this assumption does not hold. Batteryless devices rely completely on external energy sources, like energy harvested from an RFID Reader. The small capacitors that power these batteryless devices can't hold much energy—only enough for a few hundred milliseconds of work, losing memory and their location in time, when the supply is not sufficient to stay alive. These devices operate blind, forced to reconstruct progress from checkpoints left by a possibly long dead version of themselves. Long running tasks must be executed piecemeal across reboots using checkpointing[107] and careful scheduling[16]. Fragments of execution must be pieced together to deliver meaningful results. Even with careful checkpointing and scheduling, however, sensing outcomes are never guaranteed.

Building applications resilient to an unstable power supply is challenging because system designers lack the debugging tools, language and runtime support, and batteryless hardware platforms. Because of this, batteryless sensor deployments have been limited to simple programs at small scale, designed by experts. To move batteryless sensors beyond computational RFID and into the mainstream of traditional sensing, the developmental burden of batteryless applications must be eased.

1.1 Thesis Statement and Contributions

This dissertation seeks to firmly establish the following thesis: **Tiny, energy harvesting, batteryless devices can support long-running, sophisticated sensing applications.**

To test this thesis we present (1) a full stack approach and sensing platform that transparently manages the temporal, computational, and consistency difficulties caused by flaky power, (2) tools that enable exhaustive testing and confident deployment, and (3) language and runtime techniques that mask the complexity of intermittence from the programmer. The systems research done to support these applications readily translates to other areas of computer science and computer engineering, as well as enables broad impacts across many other fields of science and engineering.

1.1.1 Contributions

This work presents three broad contributions extending previous work; all with the goal of enabling batteryless, energy harvesting sensing on tiny devices meeting the constraints and requirements of the future Internet of Things.

- **Tools** which enable repeatable experimentation of batteryless devices (Chapter 3)

- **Hardware** platforms for managing energy storage and timekeeping between power failures, while providing a usable way to rapidly prototype batteryeless devices (Chapter 4)
- **Language** and runtime support for intermittent computing on batteryless devices (Chapter 5)

Innovating in these three areas enables application developers to quickly and confidently prototype, test, debug, and deploy perpetual sensing systems in many far-reaching application areas.

1.2 Dissertation Overview

The rest of this dissertation is organized as follows.

Chapter 2 motivates and explains the need for batteryless, energy harvesting systems to support the emerging concerns of the Internet-of-Things. This chapter discusses four main reasons from an economic and systems perspective, that prevent *battery powered sensors* from being deployed at very large scales.

Chapter 3 presents a system, **Ekho**, that establishes the scientific foundation for realistic and repeatable testing and comparison of batteryless sensors pre-deployment. This section introduces a powerful energy harvesting environment abstraction, I–V curves, that Ekho uses to establish this foundation.

Chapter 4 describes **Flicker** (and supporting technologies), a reconfigurable, modular, extensible, hardware platform for the batteryless Internet-of-Things. Flicker depends on two key technologies introduced and described in this chapter, Federated Energy, and Remanence Timekeepers.

Chapter 5 describes **Mayfly**, a language and runtime system for timely execution on batteryless sensing devices. Mayfly uses a declarative, graph based programming language to mask intermittent execution and improve usability and comprehension for developers of batteryless applications.

Finally, we conclude in Chapter 6.

CHAPTER 2
BATTERYLESS SENSING

Sensing platforms, from the very early days, have been powered by batteries. However, this must change, to enable the new applications required by the Internet of Things. Batteries are already a detriment to sensing devices, because of physical factors such as size, weight and cost, the human expense of maintenance, and the severe environmental impact. Aggravating the problem, improvements in battery technology have historically come slowly, never keeping pace with the rapid advances occurring in the sensing community. The problems of batteries will only compound with the scale of the IoT, as the number of connected devices go from millions, to billions, to trillions. Future ubiquitous sensing applications can not afford these costs. Below we describe the four main factors that prevent battery usage in the modern, and future sensing applications in the Internet of Things:

Size, Weight, Cost: Batteries are expensive, often one of the most expensive components of a sensor, especially when optimizing for energy density. Batteries are also heavy, and large, usually taking up the most area on a sensing device. These factors alone confine the usefulness of most sensing platforms by limiting the applications, and the deployed sensor density. *The cost of individual devices must be extremely low, and the size footprint small, to make the IoT vision feasible.*

Wear and Maintenance: Batteries wear out quickly in wireless sensor networks; even when carefully managed. All batteries will eventually age and die, no matter the control circuitry. Often the battery is severely damaged by overcharging and overdischarging, reducing the lifetime even further[13]. Replacement and maintenance, particularly for a large number of sensing devices, becomes prohibitively costly as this maintenance is generally done by humans. *It will be not possible to change the batteries of the trillions of tiny sensors that will make up the Internet of Things.*

Slow Battery Improvement: Moore's law does not apply to the chemical and manufacturing processes that underly the production (and research) around battery technology. Battery improvements have been slow after the most recent chemical substitution of lithium ion for nickel metal hydride, creating lighter and more dense batteries. Lithium (with three protons) is the lightest possible substance available to create batteries with.

Subsequently, battery technology improvements are on the order of 5% per year, impossible to keep up with the fast advances in computing power.

Environmental Harm: The most important issues with batteries powering the Internet of Things are the environmental concerns associated with processing and disposing of large numbers of dead batteries[72]. Fundamental technological obstacles must be overcome before large scale deployment of batteries is environmentally sustainable. This problem is already manifested in the area of consumer electronics recycling; the weight of dead, rechargeable lithium batteries in China is expected to surpass 500 thousand metric tons by 2020[135]. Additionally, dead batteries leech harmful chemicals into the soil, including chromium, lead, lithium, and thallium[66]. *Before the Internet of Things can truly succeed, the environmental burden of recycling trillions of batteries must be addressed, or subverted.*

2.1 Batteryless Devices and Applications

The physical factors, maintenance costs, and environmental problems surrounding batteries, have forced the creation of a new class of sensing device that is batteryless, operates purely on harvested energy, and computes intermittently. Motivated by new mobile applications with strict size and cost constraints, lifetime requirements measured in decades, as well as recent advances in low-power microcontrollers, developers of sensor platforms have decided to leave their batteries behind and attempt to sense on transient power. Batteryless devices usually have five major components; 1) an energy harvester (solar, kinetic, RF, microbial, and others), 2) a small capacitor for energy storage, 3) a microcontroller, 4) a communication channel (usually radio or backscatter) and 5) sensors and actuators.

Betteryless devices are perhaps most visible in the application realm that RFID occupies. Contactless smart cards and computational RFID tags (CRFIDs) [106, 114, 136, 133]— typically have limited computational power, rely on wireless transmissions from a reader both for energy and for timing information, and lose power frequently due to minimal energy storage.

These batteryless devices enable new sensing applications in wearable technology, especially mHealth[105], water infrastructure monitoring through microbial energy harvesting[30], environmental monitoring[89], greenhouse monitoring[53], and many other applications. Other applications can be imagined where sensors are embedded in clothing, inside animals, or put in space. Each of the applications described, were not possible with conventional sensing techniques; either because of extended lifetime requirements, deployment and maintenance impossibilities, or just economics. These applications change the way we look at sensing, however, each of these applications can't work with batteries.

Despite progress in batteryless platforms and technologies, the previously described applications and hardware are all engineered for one problem space or application. Currently, there does not exist a generalized platform for batteryless sensing. Fundamental challenges in energy management, timekeeping through power failures, testing and verification, and other areas must be overcome before a general platform can be engineered.

2.2 Challenges

Designing the software and hardware that composes batteryless devices is challenging. We detail the main challenges in the rest of this section.

Volatile Energy Environments: Unlike traditional battery powered sensors (which duty cycle to prolong lifetime), energy harvesting devices must work opportunistically. Having too much or too little energy is equally inefficient and wasteful—energy leakage is high when the devices reach their full capacity. Greedily using energy can restrict functionality, while under utilization of harvested energy is wasteful in terms of computation that could have been performed. Power supply fluctuations affect a device's runtime behavior in ways that are often difficult to predict or reproduce in the lab during testing. Matters are complicated further by the fact that the energy harvested by each device depends not only on environmental conditions, but also on the device's supply voltage at runtime. This means that the amount of energy harvested is partially determined by the behavior of the device.

Fragmented Execution: Since the processor frequently reboots because of an unstable power supply, execution becomes fragmented. Maintaining forward progress is difficult to code for, and difficult to visualize. Combining fragmented execution opportunities into cohesive programs is a difficult task for programmers.

Temporal Data: In traditional computing, a reliable sense of time can be provided using an internal clock. Time measurement errors, due to clock drift or power failures, can be corrected by synchronizing with a trusted peer or other networked time source. When the available energy of batteryless devices runs out, and the capacitor depletes, the microcontroller, volatile RAM, and all clocks are reset. All of the sensor platforms previous timestamps (and therefore sensor data) have no meaning, since the local clock will start back at zero. This makes management of sensor data that is time sensitive, and logging of execution, difficult.

Constrained Resources: Batteryless sensing platforms, out of necessity, are extremely constrained. The small amounts of energy that are gathered can only support slow processors(1-8MHz) with small memory space (8KB to 128KB). This makes it difficult to process data or perform complex algorithms even when one has a stable power supply.

Usability: All of the previous challenges combine to make the developer experience when working with batteryless devices abysmal. Determining what went wrong and where becomes very difficult when debugging can only be done with an oscilloscope. Lack of a definitive toolchain exacerbates these problems even further; potentially limiting the applications that are deployed, and making a high barrier to programming these devices.

CHAPTER 3

EKHO: REALISTIC AND REPEATABLE EXPERIMENTATION

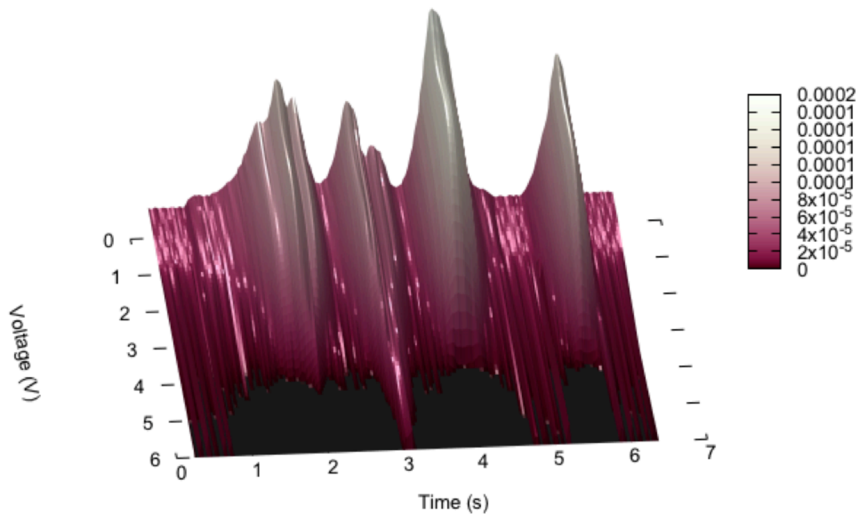


Figure 3.1: A solar IV surface generated from an IXYS solar cell exposed to a lightbox. An IV surface captures all possible harvesting scenarios for a harvester. Each possible harvesting current (I) for the supply voltage (V) over time are shown.

Ekho is a tool that records and emulates energy harvesting conditions for capacitor powered energy harvesting devices, and is generally applicable to a wide range of harvesting technologies. Ekho uses a novel method to explore and record an energy harvesting environment by modulating the load using a precisely controlled digital potentiometer. This energy harvesting environment (solar, RF, kinetic) is processed and stored to be later replayed through a custom analog front-end which serves as a current source. To evaluate Ekho, we have developed two prototypes; a desktop version, and a mobile recording version. Using these prototypes, we evaluate Ekho's ability to replicate energy harvesting conditions both accurately and consistently. In our evaluation we found Ekho provides a consistent environment for repeatable experimentation on batteryless sensing devices.

Energy is the greatest single limiting factor for many mobile sensing applications, and recent advances in energy harvesting are making it possible to deploy smaller sensing devices for long periods of time without the need for regular maintenance (to replace or recharge batteries). A wide range of environmental sources are readily available, and whether a device harvests solar [77, 67], radio frequency (RF) [107, 16], kinetic [86] or even energy from other devices [70], harvested energy sources are often highly variable, scarce, and difficult to predict.

Battery technology has been able to store surplus energy harvested during the high energy periods to be used during the low energy periods. However, rechargeable batteries wear out, charge slowly, require special protection and charging circuitry, pose environmental risks, and fundamentally limit the lifetime and deployability of today's mobile computing devices. These have inspired a range of capacitor-based energy storage solutions for sensor devices [114, 132, 133, 126, 47, 45] that harvest energy, charge quickly, and can store only enough energy for short bursts of operation.

However, hardware and software solutions for energy harvesting sensing devices with limited energy storage are difficult to design, debug, and especially to evaluate. Harvested energy varies and energy storage constraints continue to tighten in order to accommodate smaller mobile form factors. The consequence is that, in addition to the more traditional challenges faced by mobile devices (like uncertain network connectivity), it is often difficult for system designers to predict how their devices will behave at runtime. Reliably comparing different algorithms, approaches, or configurations is often impractical or extremely labor-intensive. These challenges are primarily the result of two key characteristics of energy harvesting systems: 1) *energy harvesting is erratic and unpredictable*, and 2) *the amount of energy harvested depends not only on environmental conditions, but also on the device's behavior at runtime.*

The combination of a behavior-dependent energy supply and a high degree of runtime volatility makes repeatable experimentation impractical, using traditional testing strategies. Two test runs with *the same hardware and software* may result in dramatically different results, due to differences in energy harvesting conditions. Two runs with *different software or hardware configurations* may produce dramatically different results under the same harvesting conditions, assuming that harvesting conditions can be replicated. Runtime conditions are often vastly different from in-lab conditions, and may be difficult to replicate during testing. In order to compare different algorithms or different hardware configurations, a system designer must currently either run a large number (hundreds or thousands) of tests under realistic runtime conditions and compare results stochastically (a labor-intensive and imprecise approach), or control energy harvesting conditions in simulation.

Simulators have been developed that predict the power consumption [35, 117, 107, 22, 97] and even the energy harvesting [47, 107] behaviors of sensor devices. Unfortunately, most ignore the impact of device behavior on energy harvesting. Indeed, simulators must depend entirely on an accurate model of the test device's hardware characteristics. As device hardware evolves, or when a designer wants to try out a different hardware component (e.g., a new sensor, actuator, or processor), the simulation software must be updated—often involving a significant amount of in-lab measurement and testing. Besides simulation, testing by deployment, then re-deployment on real hardware gives us an accurate picture of performance. However, with energy harvesting devices, this method of testing loses validity as energy environments constantly change.

This chapter explores another option, *emulation*. Instead of depending on software models of energy harvesting and consumption, an energy harvesting emulator records energy harvesting conditions and then accurately reproduces the recorded conditions (in the form of physical "harvested" power) to a real test device running in the lab. This approach provides system designers with a realistic and repeatable evaluation technique, without sacrificing flexibility—modifying the hardware and software on the test device does not require any changes to the emulator.

In this chapter, we describe the design, implementation, and evaluation of Ekho, a tool that records and emulates energy harvesting conditions for capacitor powered energy harvesting devices. Ekho is generally applicable to a wide range of harvesting technologies. Ekho uses a novel method to explore and record an energy harvesting environment by modulating the load using a precisely controlled digital potentiometer. This energy harvesting environment (solar, RF, kinetic) is processed and stored to be later replayed through a custom analog front-end which serves as a current source. To evaluate Ekho, we have developed two prototypes; a desktop version, and a mobile recording version. Using these prototypes, we evaluate Ekho's ability to replicate energy harvesting conditions both accurately and consistently. In our evaluation we found Ekho is consistent within 68.7 µA[1] from test run to test run, emulating recorded solar harvesting environments to mote-class devices running a variety of test programs. Ekho reproduces a recorded solar trace with a mean error of less than 77.4 µA from the recorded surface. We also found that Ekho was able to record RF energy harvesting environments and replay them with high fidelity, and low error rates for most transmit powers. Finally, we found that Ekho can reproduce kinetic energy environments with a mean error of 15.0 µA from the recorded surface. It should be noted that a faster processor was used for the kinetic results which significantly contributed to the accuracy gains over solar. Parts of this work were presented at SenSys 2016 [51].

[1]depending on capacitance

3.1 Harvesting Energy

Ambient energy, harvested from the environment, is key to the success of any sensing and pervasive computing application that requires small devices to operate maintenance-free over long periods of time. Energy in its many forms (solar, RF, mechanical, thermal, etc) can be converted into electrical energy that can be stored in batteries or capacitors and used to power the device's processor, sensors, and other components. Rechargeable batteries usually wear out after a few years due to a limited number of charge cycles, charge slowly and pose disposal and safety hazard. Capacitors, however, can last for decades of useful operation, charge quickly and are more environmentally friendly. **The work in this chapter focuses on battery-less energy harvesting devices, however, the techniques are generally applicable to battery powered sensing systems, just not as useful.**

Unfortunately, designing devices that effectively use this never-ending supply of free energy is challenging. Unlike traditional battery powered sensors (which duty cycle to prolong lifetime), energy harvesting devices must work opportunistically. Having too much or too little energy is equally inefficient and wasteful—energy leakage is high when the devices reach their full capacity. Greedily using energy can restrict functionality, while under utilization of harvested energy is wasteful in terms of computation that could have been performed.

Additionally, nearly all environmental energy sources vary widely and unpredictably at runtime, and as new applications require smaller form factors and lower energy storage capacities [21], power supply volatility increasingly influences and defines device behavior. Devices, like *computational RFIDs* (CRFID) [16, 113, 136], that replace batteries with small capacitors and store only enough energy for, at most, a few seconds of operation are especially susceptible, and may see their supply voltage increase threefold or fall to zero in seconds. Power supply fluctuations affect a device's runtime behavior in ways that are often difficult to predict or reproduce in the lab during testing.

Matters are complicated further by the fact that the energy harvested by each device depends not only on environmental conditions, but also on the device's supply voltage at runtime. The relationship between supply voltage and charge current can be characterized by an I–V curve, a function that describes how harvesting current (I) changes, with respect to the device's supply voltage (V). Different programs (loads) will occupy different areas of the I–V curve as shown by Figure 3.2.

Figure 3.3 shows six (6) example I–V curves, two produced by a solar panel under high and low light conditions, two produced by a Peltier generator—which converts thermal differentials into electrical

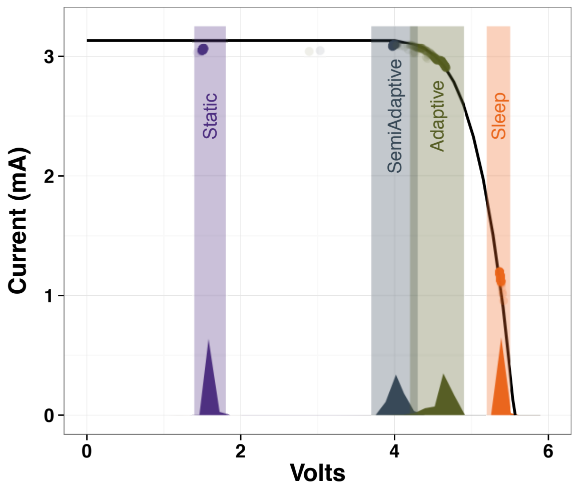


Figure 3.2: This figure shows how program behaviors influence energy harvesting performance. Four (4) programs' harvested currents, measured over a 20 s **period of time, are shown with respect to their supply voltage, while connected to a programmable solar environment (see Section 5.3) generating a single I–V curve (shown in black). Points represent an average of many samples (many points are not contained in shaded regions), and the histogram's along the bottom show the sample density at each point. Due to differences in behavior (power consumption), each example program occupies a different section of the I–V curve. These differences result in significant variations in harvested power.**

current—under 5 °C and 10 °C thermal differentials, and two produced by RF energy from a reader at +32.5 dBm and +27.75 dBm. In all three cases, environmental changes alter the harvester's I–V curve. In addition, each harvester produces its own distinct "family" of curves, with a common characteristic shape. These curves change over time, resulting in I–V surfaces, sequences of I–V curves belonging to a certain family.

At runtime, an energy harvester's I–V curves impact program behaviors and experimental outcomes. For example, two algorithms that draw different amounts of current will deplete their capacitors at differing rates, resulting in different supply voltages, and, consequently, different amounts of harvested power ($P = IV$).

Figure 3.4 illustrates this scenario by showing the amount of power harvested by two TI EZ430-RF2500 devices running two different programs under the same solar harvesting conditions. Both periodically read data from an on-board temperature sensor, however the Adaptive program modulates its wait time depending on the voltage so it can sense when energy is scarce, while the Static program senses and writes whenever it is able. Under the test conditions, Adaptive stayed near the high energy knee of the I–V curve (see Figure 3.2), maximizing on available energy by watching its supply voltage, while the Static program harvested significantly less energy by being greedy. The *maximum power point (MPP)* is also shown to demonstrate the amount of power that could potentially have been harvested by a device with the right supply voltage.

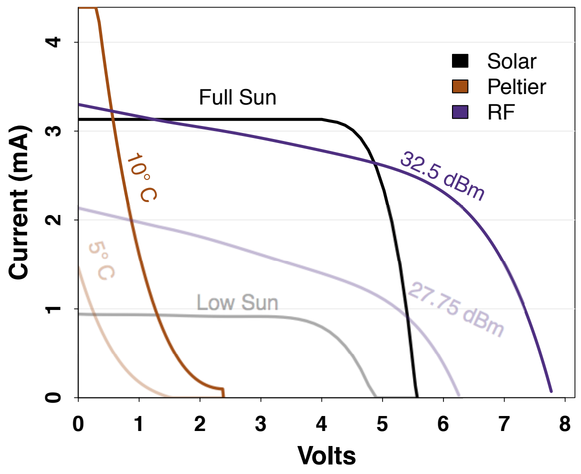


Figure 3.3: Six I–V curves are shown, produced by three different energy harvesters—a solar panel, a Peltier generator, and an RF Reader—each under two (2) different energy harvesting conditions. Each harvester produces its own "family" of curves, with a common characteristic shape. Each of the above I–V curves were captured with the recording feature of Ekho. Note that the Peltier curve has been scaled $17x$ **for purposes of illustration.**

This is illustrated further by Figure 3.5 and Figure 3.6. These I–V surfaces were recorded by Ekho from solar and kinetic energy environments, respectively. Different execution, behavior, or energy consumption causes different paths across the surface with different parts of the surface (those closest to the MPP) being more efficient than others.

Consequently, any attempt to predict how a low-power energy harvesting device will behave in the wild, must take into account the harvester's I–V characteristics and the resulting program variation. There are two common methods to predicting device behavior; (1) replaying a harvested power trace gathered from a device, and (2) using a programmable energy environment such as a light-box.

Replaying Power: One approach to making energy harvesting reproducible is to measure the harvested power as the device executes, and then replay the collected power trace. This approach has been used in other harvester-powered mobile systems [119], and our early efforts focused on replaying power traces.

Replaying a power trace is an attractive technique to predict device behavior under certain energy conditions since designing the hardware is simple and straightforward, and provides a reasonably accurate solution for devices with a stable supply voltage—like those with large batteries, which typically vary by less than half a volt when between 15% and 85% of a full charge. When the battery is nearly full or empty, simply replaying a power trace to simulate this will over or underestimate the energy that would be harvested in an actual deployment scenario.

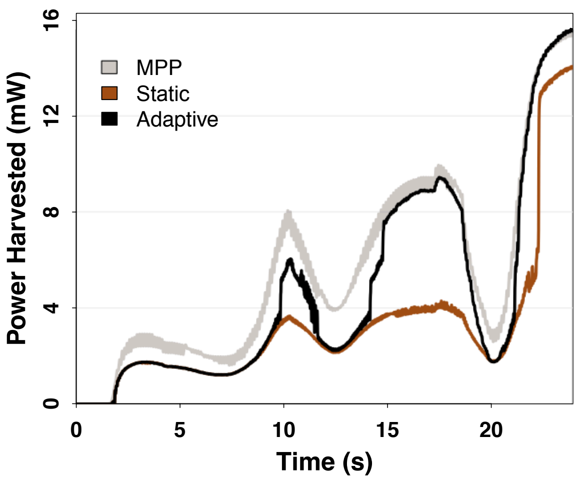


Figure 3.4: Harvested power is shown for two TI EZ430-RF2500 target boards running two different programs that both write to flash in different ways—the Static program writes as fast as possible if there is any power available, the Adaptive program adapts so it can write even when harvestable energy is scarce—under the same solar energy harvesting conditions. Differences in power consumption result in different amounts of harvested power.

While replaying power will work most of the time for devices with large batteries, devices that store their energy primarily in small batteries or capacitors have much less stable supply voltages that explore much more of the energy harvester's I–V curve. Figure 3.7 illustrates the I–V characteristics that are produced by recording the power harvested at a single point and replaying that power during experiments. Replaying constant power results in an effective I–V curve, defined by $I = \frac{P}{V}$, where P is the power being replayed. The figure shows three such I–V curves that could be inferred from the same solar I–V curve. In all cases, the "constant power" curves approximate the real energy harvesting characteristics in only a small part of their range. In a later section, we compare the results of emulating power with different training sets to Ekho, and the light-box mentioned below.

Programmable Energy Environments:

Programmable energy environments offer a somewhat more comprehensive effort at reproducing energy conditions than simple power replay. These environments make an effort to isolate an energy source, such as solar, RF, or vibrations, and create a repeatable environment to provide energy to a system [10]. Unfortunately, these devices are often difficult to construct and require careful calibration in order to provide accurate and reproducible results. For example, a reproducible vibrational energy environment requires special care to reduce the effect of ambient vibrations and noise events; experiments can be interrupted or affected by environmental events such as students walking down a hall, a chair moving, or even loud noises. These

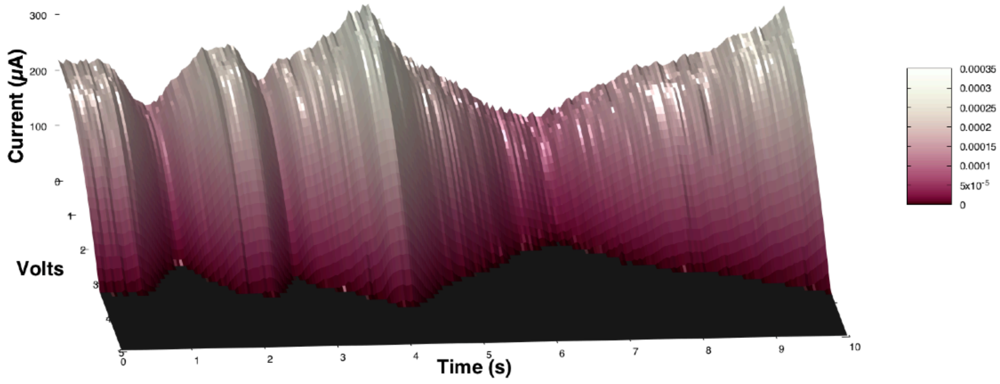


Figure 3.5: Shown is a solar I–V surface recorded by Ekho. A solar panel was attached to the harvester input of Ekho and a light was shown on the panel from different distances. The large hills are when the light was closest to the panel, with the valleys being when the light was farthest away.

devices also tend to possess many points of failure or errata introduction. However, these analog solutions to programmable energy environments are much better than naive approaches that simplistically replay power.

In developing Ekho, we made extensive use of three such environments, dubbed the "light-box", the "RF-box", and the "shake-table". The light-box consists of a vehicle headlamp whose output is controlled via microcontroller and offers the ability to provide a controlled amount of light directly to a solar panel with minimal influence from outside sources. The energy produced by the light-box can then be used to power a low energy system and produce reasonably repeatable results, however it is not perfect as Figure 3.8 shows. As the light-box heats up, it changes the energy conditions inside the box, causing program behaviors to change from one run to the next. The RF-box is lined with copper mesh, effectively creating a Faraday cage, that isolates the interior from wireless interference, which can cause variation in harvesting current. Inside the RF-box is a programmatically controlled antenna that can power small CRFID tags such as the UMich Moo [136]. By modulating the transmit power different harvesting conditions can be created. The shake-table uses a signal generator connected to a surface transducer to excite a piezoelectric ceramic, which provides power. The amplitude and frequency of vibration can be adjusted to create different energy harvesting conditions. Constructing other programmable energy environments requires similar steps to isolate the energy source.

3.2 Ekho System Overview

The Ekho emulator is designed to capture the physical characteristics of an energy harvesting environment and recreate those environmental conditions in order to enable repeatable and realistic in-lab

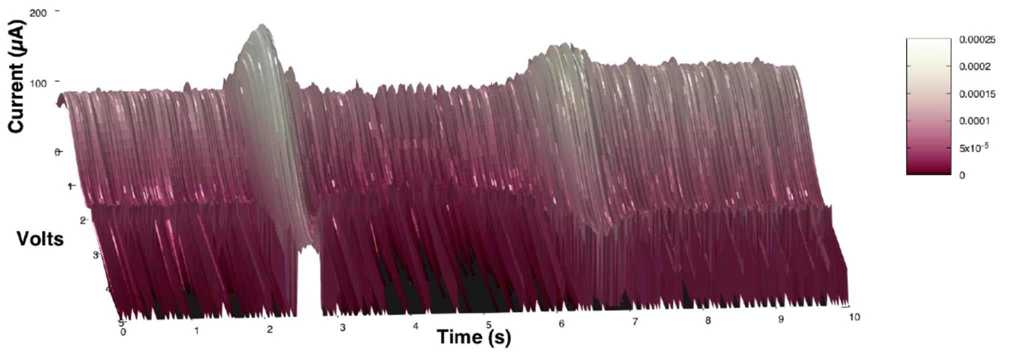


Figure 3.6: Shown is a kinetic I–V surface recorded by Ekho. This surface was generated by attaching a weighted MIDE V21BL piezo ceramic to a surface transducer generating a sine wave that ramped from 160 Hz to 390 Hz, then back to 160 Hz. The large dips are the result of the surface transducer generating the resonant frequency of the piezo, causing a burst of extra harvestable energy. Ekho allows system designers to capture electrical characteristics dependent on the energy environment (like resonant frequency) in a straightforward way.

testing. Ekho does not emulate program behaviors, but captures features of the energy environment that allow testing of different program behaviors in a realistic way. Rather than focus on supporting a specific harvesting technology, our design of Ekho is focused on providing a generally applicable tool that supports a wide range of energy sources, while providing users with flexibility, mobility, accuracy, and consistency.

Generality: In Ekho, energy harvesting conditions are represented as I–V curves—an abstraction that, as discussed in Section 5.1, can be used to characterize any common energy harvesting technology. Changes in harvesting conditions over time are represented by combining multiple I–V curves into I–V surfaces. This generality frees the experimenter from designing expensive custom hardware such as a light-box or Faraday cage[2] to test devices before deployment. Ekho uses a novel method to explore and record these I–V surfaces by quickly modulating the load using a precisely controlled digital potentiometer. This allows Ekho to rapidly explore any I–V surface, including RF, with minimal changes in experimental setup.

Flexibility: A key focus of our design is to allow application designers to effortlessly compare different software and hardware options. Ekho achieves this by mimicking the physics of an energy-harvester, providing realistic and repeatable power to real test devices. Using this approach, trying out a new sensor, energy harvester, scheduling algorithm, or even a new processor, requires no changes to the emulator, no profiling or modeling. The user makes the desired change to the experimental setup and continues using Ekho with no Ekho-specific configurations or alterations.

[2]A Faraday cage is a container made of conducting material that prevents the entry or escape of an electromagnetic field.

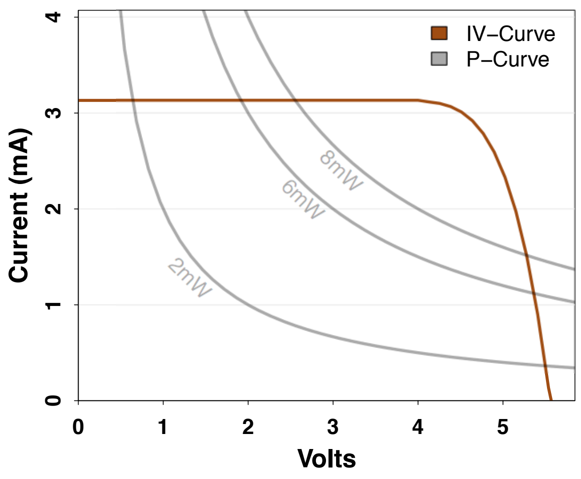


Figure 3.7: Shown is a single I–V curve, and the consequences of choosing constant power to represent it. Depending on the load, the generated P-curves from the power trace can cause unrealistic changes in the programs actual harvested energy. As shown, emulating constant power is a poor replacement for emulating the actual I–V curve.

Mobility: Since Ekho is a supporting tool for tiny energy harvesting devices deployed in many different conditions, a mobile form-factor is essential for the recording function. This feature allows Ekho to be deployed with existing systems, or pre-deployed to characterize the energy environment beforehand. We have designed a deployable version to demonstrate the mobility of Ekho.

Accuracy: An energy harvesting emulator is only as useful as it is able to accurately recreate energy harvesting conditions. At runtime, devices often experience a wide range of rapidly-changing harvesting conditions. Ekho is designed to accurately estimate I–V surfaces of varying shapes and magnitudes. This allows Ekho to recreate recorded conditions with sufficient accuracy to mimic energy fluctuations and patterns that devices confront in the myriad conditions of real-world deployment.

Consistency: Perhaps the most important goal for Ekho is consistency. No two recorded traces of energy harvesting conditions will be identical, and test engineers may often be willing to tolerate emulations that are similar, but not identical, to those recorded in the wild. In contrast, experiments that aim at comparing different algorithms or hardware choices require that test runs be consistent. Inconsistent emulation yields results that are not reproducible and difficult to interpret. Ekho offers favorable accuracy behaviorally and physically compared to other controlled energy harvesting environments but excels in reproducing energy conditions consistently.

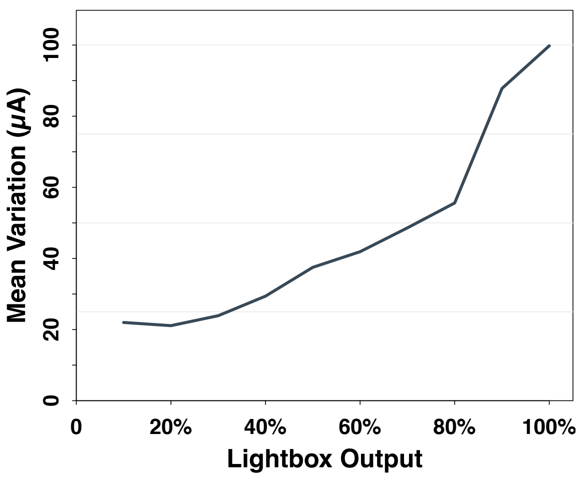


Figure 3.8: The light-box mean variation between runs increases with light intensity for static loads. The light-box, unlike Ekho is susceptible to environmental changes and care must be taken to control those. Temperature changes after long use are one such factor that affects repeatability.

3.2.1 System Architecture

In order to achieve these goals, we have designed a system architecture, shown in Figure 4.9, which consists of three interdependent modules: a *surface manager* that stores I–V surfaces and manages the high-level recording and emulation logic for the system; a low-latency *I–V curve controller* that sequentially emulates the I–V curves that correspond to each single point in time during an emulated surface; and an analog *front-end* module that facilitates controllable current emulation and provides signal conditioning that is needed for taking accurate current and voltage measurements, especially during periods when harvested energy is scarce.

Ekho's surface manager controls both the recording and emulation of energy harvesting conditions. This includes receiving current and voltage measurements from the I–V Controller during recording, estimating I–V curves from the received measurements, storing I–V surfaces in a digital format, and sending I–V curves one-by-one to the I–V curve controller during emulation. The storage and computational requirements for these activities fit comfortably within the capabilities of the current generation of laptop and desktop computers. The mobile version separates the recording surface management from the emulation, but otherwise performs the same function.

In order to accurately emulate I–V curves received from the surface manager, the I–V curve controller must be able to quickly gather current and voltage measurements and respond to those changes appropriately (within a few μs). This requirement is most easily satisfied by a processor with integrated

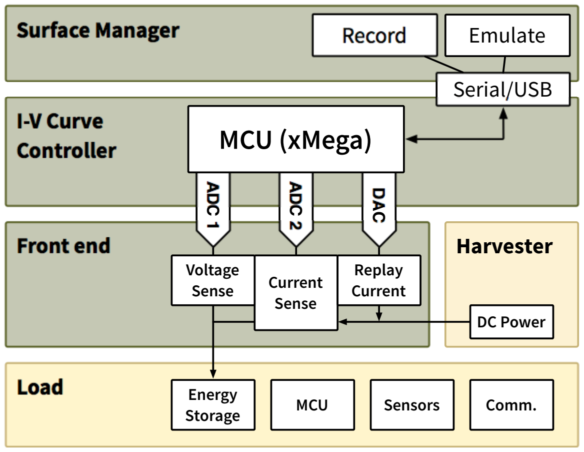


Figure 3.9: Ekho consists of three interdependent modules: a surface manager that stores I–V surfaces and manages the high-level recording and emulation logic for the system; a low-latency controller that sequentially emulates the I–V curves that correspond to each single point in time during an emulated surface; and a front-end module that facilitates controllable current emulation and provides signal conditioning that is needed for taking accurate current and voltage measurements.

analog-to-digital (ADC) and digital-to-analog (DAC) capabilities, a combination that is rarely found in today's high-speed processors, but which are provided by some higher-speed microcontrollers, like Atmel's AVR XMEGA line of controllers [26], and some ARM controllers, which we use in our prototypes, described in Section 5.3.

The I–V curve controller relies on the third module, an analog front-end, to provide the amplification and other signal conditioning needed for accurate I–V curve emulation and measurement. When capturing energy harvesting conditions, this circuit is placed between the harvester and test load. During emulation, the front-end takes on the role of energy harvester, providing the device under test with a current supply that mimics the energy source being emulated.

The following sections describe how these modules work together, in two different operating modes, to record and emulate harvesting conditions.

3.2.2 Recording I–V Surfaces

Ekho captures the energy harvesting conditions by measuring them directly. Electrical current is measured by the front-end as it flows from the energy harvester into the test device's storage capacitor. Current is measured by observing the amplified voltage drop across a low-tolerance sense resistor. The test device's supply voltage is also measured. These current-voltage (I–V) measurements are converted from analog voltages to digital values by the I–V curve controller as rapidly as possible and passed along to the surface manager for post-processing, where I–V curves are generated from the I–V pair point cloud.

This series of recorded I–V pairs represent a single path across the three-dimensional surface that represents the harvesting conditions during the trace; the surface manager's challenge is to estimate the entire surface from this single path. Each recorded I–V pair captures one point on the I–V curve that represents harvesting conditions at the time it was captured. When considered alone, each point could have been produced by an infinite number of different I–V curves; however, a series of I–V measurements can be used to infer the current I–V curve's shape, assuming 1) that the measurements are gathered quickly before the state of the I–V curve changes measurably, and 2) that the measurements adequately span the I–V curve's voltage range. Taking measurements rapidly (>1 million samples/second) is straightforward. Inducing enough supply voltage volatility to quickly and fully characterize the I–V curve at each point in time requires more care. A key contribution of Ekho is its novel method to induce supply voltage volatility.

3.2.2.1 Inducing Supply Voltage Volatility

At runtime, the power consumption of a typical test device (or test load), like a CRFID or mote-class sensor, does not often change rapidly enough or significantly enough to explore the entire I–V curve. This is illustrated in Figure 3.10, which shows two sets of 6,000 I–V pairs collected by Ekho over a period of 30 ms, under similar solar harvesting conditions, while using two different test loads: an off-the-shelf TI EZ430-RF2500 mote [58], and a custom *smart* test load that we have designed specifically for inducing voltage changes in order to assist with Ekho's recording mode.

The custom "smart" load is a digital potentiometer controlled by a microcontroller which rapidly alters its power consumption in order to induce large fluctuations in supply voltage for more accurate recording. The Arduino controls the potentiometer and makes it cycle through a predetermined number of resistance settings for a given time delay. These changes produce a wide range of load currents that explore different parts of the I–V curve. As long as the cycle frequency is high enough, and the upper and lower bound of the potentiometer's resistance settings can exercise the extreme ends of the curve, the shape of any instantaneous I–V curve can be gathered. In our experiments we have found that a 100 kΩ potentiometer provides a large enough range. For custom "smart" load "cycle frequency" (the number of times a second the smart load cycles through all its resistance settings), we found that 100 Hz can capture solar I–V curves, and 1000 Hz and above is sufficient to approximate an RF I–V surface

As shown in Figure 3.10, when using the custom load, the measurements are spread evenly across the I–V curve, the *smart* load effectively explores the entire I–V curve, while the mote measures only a small part

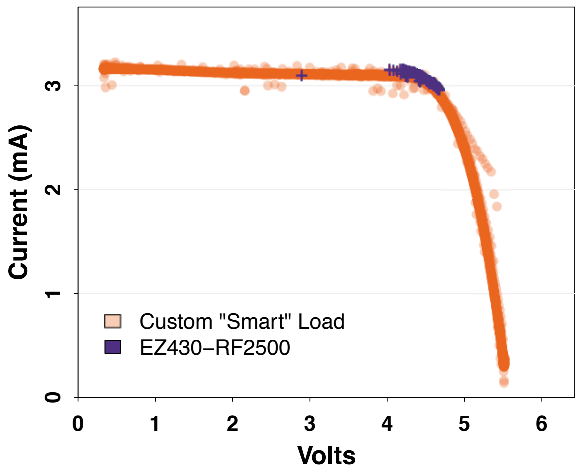


Figure 3.10: This figure shows recorded I–V measurements, as produced by both the Ekho smart load device and a typical mote-class sensor device. By intentionally increasing the power supply volatility, the smart load provides much better coverage of the I–V curve being recorded, which improves Ekho's recording accuracy.

of the curve.

3.2.2.2 Surface Construction

Once these measurements have been captured, the surface manager uses a curve-fitting algorithm to estimate the shape of the I–V curve that most closely fits each window of data, and the series of inferred I–V curves make up the I–V surface that is stored for later use during testing. A variety of curve-fitting algorithms exist, which could be used. For the sake of simplicity, we use the polynomial fitting algorithm provided by the GNU Scientific Library (GSL) [40], and have found it to work well in practice, both in terms of accuracy and efficiency. The size of each window is configurable (and is closely related to the custom load cycle frequency), and represents a tradeoff between temporal accuracy and I–V curve accuracy. If the window is too small, containing too few points with poor coverage, the estimated I–V curves may be inaccurate. If the window is too large, then short-term changes in the I–V surface could be effectively filtered out of the captured representation, decreasing the temporal accuracy of the surface; however this is harvester dependent. Trading temporal accuracy for a larger window (and therefore I–V curve accuracy) will not influence the final behaviors of most programs running on slow changing solar surfaces where curves switch at less than 100 Hz. However, for RF surfaces this can pose a significant problem as curves can change upwards of one thousand times a second.

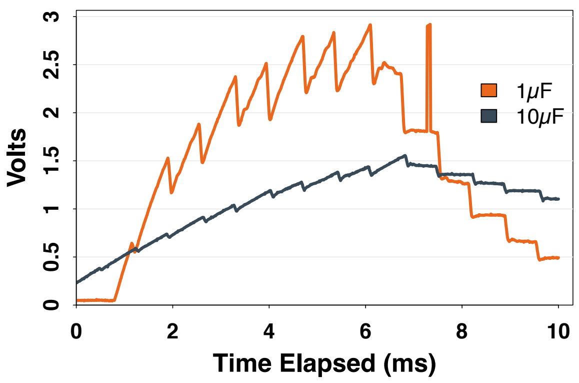
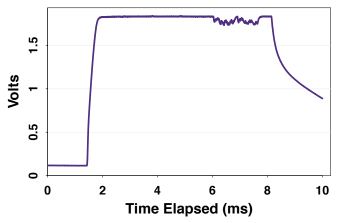


Figure 3.11: This figure shows the effect of capacitance while recording an RF I–V surface. As the capacitance increases the output is averaged, and important features are lost. Each peak is the custom "smart" load changing its resistance setting, these peaks are absorbed by the larger capacitance, which means that voltage volatility is lost. Because of this the I–V surface is not as fully explored. For energy environments, like solar, that evolve slowly, this may be acceptable, for volatile RF or kinetic harvesting environments, important surface information will be lost.

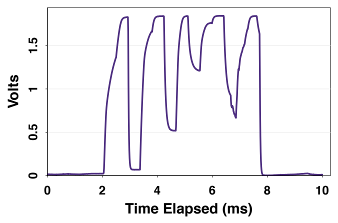
3.2.2.3 Complicating Factors

Care must be taken when recording an energy environment. The sensitivity of the capacitor powered energy harvesting device under consideration and the accuracy required for emulating will influence decisions made when recording. Choosing the capacitance and cycle frequency of the custom "smart" load is critical to an accurate I–V surface recording. Capacitance while recording has the effect of averaging out the surface over some time period as shown in Figure 3.11. This smoothing effect is desirable for I–V surfaces that change slowly (such as solar), as it reduces noise, and results in a cleaner representation of each individual I–V curve. However, applying extra capacitance to fast changing surfaces (such as RF) will average out peaks and valleys in the recorded surface. This averaging will change the final harvested power, and therefore the final program behavior. This smoothing capacitance is only necessary if I–V surfaces that are being recorded with Ekho are noisy, in most cases, a large smoothing capacitance is unnecessary.

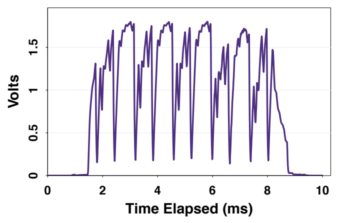
The cycle frequency of the smart load also plays a factor in determining the accuracy of the final constructed surface. If cycle frequency is set too low for a particular harvester type, the final surface will be missing important features. Alternatively, if it is set too high, Ekho may not be able to emulate it fast enough. Figure 3.12 shows differences in curve coverage and how they can affect the final surface. Ekho is configured to support a wide range of harvester types. Each of these harvester could require different combinations of cycle frequency and capacitance. In our experiments, we have found that a capacitance of 10 μF and a cycle frequency of 100 Hz is adequate for recording solar surfaces, while capacitances less than 0.1 μF and cycle



(a) Trace with a constant resistive load. Using just this to generate an I–V curve will miss important parts of the curve.



(b) Trace with smart load cycling at 1 kHz. The rapid voltage fluctuations explore more of the the I–V curve. However, the cycle frequency is not quite high enough to capture every part of the curve.



(c) Trace with smart load cycling at 10 kHz. The cycle frequency is high enough to capture details of the curve such that we can approximate the entire surface reasonably well.

Figure 3.12: RF energy harvesting voltage trace over 10ms, with three different custom "smart" load cycle frequencies.

frequencies of at least 1 kHz is required for recording RF and kinetic surfaces accurately.

3.2.3 Emulating I–V Surfaces

Ekho emulates stored I–V surfaces in three phases. First, the Surface Manager preprocesses each I–V curve in the surface for efficient transmission and emulation. Second, the curves are communicated at the appropriate time to the I–V Curve Controller. Third, the I–V curve is emulated by using the signal conditioning capabilities provided by the front-end.

In order for Ekho to emulate energy harvesting efficiently, each I–V curve needs to be represented compactly, in a form that reduces the computational workload of the I–V curve controller. To this end, each curve is discretized down to $2^n + 1$ points. A power of 2 is used for efficiency in looking up currents based on ADC-provided voltage measurements. The choice of n represents a tradeoff between smaller I–V curves which can be communicated more quickly, and larger curves which may represent the original curve most accurately. By default, Ekho uses 65-point curves ($n = 6$), which provides good results for most types of energy environments. Additionally, in order to reduce the computational load further, the surface manager precomputes the DAC value that is required to produce the desired current.

After the surface is preprocessed, the surface manager begins emulation, sending each I–V curve to the I–V curve controller at the time it is to be emulated. The new curve replaces the old curve in the microcontrollers' RAM, point-by-point, as it is received. The rate at which new curves need to be sent depends on the harvester being emulated (some harvesters' curves change faster than others). For especially fast surfaces, like RF, the I–V curve controller can store the entire surface in RAM to facilitate greater than 1 kHz curve updates. For the current prototype, this limits RF surface length to under two seconds, this limitation can be overcome using external memory to allow much longer RF traces to be emulated.

Throughout this process, the I–V curve controller emulates each curve by measuring the test device's supply voltage and playing the appropriate voltage to the front-end using its DAC, repeatedly. Finding the right DAC value requires two I–V curve lookups to find the two closest points on the curve and a linear interpolation between the two found DAC values. The voltages output by the DAC are amplified by the front-end (increasing the range up to nearly 8 V), and the amplified output is connected, through a low-tolerance 400 Ω resistor followed by the 10 Ω sense resistor (used for current sensing) to the test device's capacitor. This produces a predictable harvesting current ($I = \frac{V}{410\Omega}$).

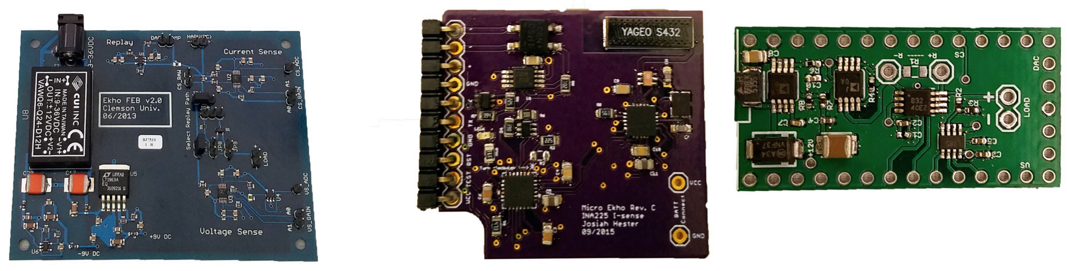
Note that the feedback loop executed by the I–V curve controller must be extremely fast. The action of emulating a current, in addition to the current draw of the test device, causes the supply voltage to increase or decrease, which necessitates a change in current. If the feedback loop is too slow, then a larger storage capacitor could be used on the device under test, if design constraints allow. Using a larger capacitor to store the harvested energy will cause the supply to change more slowly, giving Ekho more time to respond. However, in our evaluation with the most recent version of Ekho, we found that our emulation speed was more than sufficient to emulate with high accuracy, without changing the size of the storage capacitor of the device under test.

3.3 Implementation

In order to evaluate the efficacy and usefulness of our approach, we implemented four different prototypes, and the software, tools, and programmable energy environments to evaluate them, . This section gives implementation details of each.

3.3.1 Hardware

In order to evaluate the efficacy and usefulness of our approach, we have implemented four different prototypes; an analog front-end (shown in Figure 3.13a), a deployable, MSP430 based I–V surface recorded (sown in Figure 3.13b), a larger, more accurate ARM based I–V surface recorder (not shown), and an emulation only Teensy ARM breakout board (shown in Figure 3.13c). We use the evaluation setup shown in Figure 3.14 to conduct all our experiments. This desktop evaluation unit consists of a surface manager, an I–V curve controller, and a custom analog front-end. The mobile unit contains only the components necessary to record



(a) Analog Front-end (b) Micro Ekho (c) Emulator Only

Figure 3.13: Shown are our three reference Ekho implementations. Figure 3.13a shows the desktop analog front-end; this can be used for emulation and recording, and with any MCU and smart load. Figure 3.13b shows our mobile Ekho recording prototype. The mobile version is 9x smaller than the desktop version, and does not need any supporting hardware (such as the dedicated PC for the desktop version) to record I–V surfaces. With a small button battery, it can record for weeks. Figure 3.13c shows an emulation only Ekho device as a Teensy 3.2 ARM daughter board.

I–V surfaces; the analog front-end, the smart load, and a micro-controller as hybrid I–V curve controller.

3.3.1.1 Desktop Evaluation Unit

The desktop system employs a variety of different hardware components. The surface manager is implemented using a Windows 7 (64-bit) desktop. The analog front-end is implemented with a custom printed circuit board (PCB) that provides filtering and amplification for accurately measuring low-amplitude current and voltage signals. The analog front-end is powered by a 9V DC source. While Ekho is designed for low current harvesting scenarios, our current implementation can accept harvester input voltages up to 8 V and input current up to 0.5 A. This allows a broad range that can handle most low power devices. Our prototype uses two different devices to implement the I–V curve controller functionality—an Atmel ATXmega256A3B microcontroller [26] when in emulation mode, and an NI USB-6356 data acquisition device (DAQ) [56] when in record mode. The DAQ provides the needed high-speed data collection capabilities needed for recording, while the ATXmega provides a 32 MHz processor with integrated ADC and DAC for low-latency emulation. Total cost of the system, including ATXmega, Arduino, custom circuit boards, and parts (excluding the DAQ) is less than $700. The low-amplitude signals that Ekho must measure are highly susceptible to noise, induced from ambient electromagnetic radiation (from AC power lines and RF transmitters). A shielded enclosure and shielded cables are used throughout, in order to mitigate this problem. In later versions of desktop Ekho; specifically the version used for gathering the kinetic results, the I–V curve controller was implemented on a

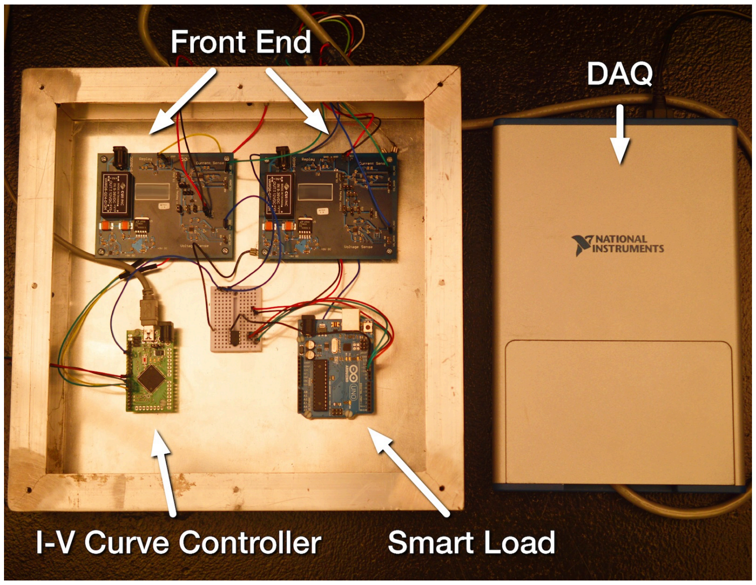


Figure 3.14: Our prototype desktop Ekho evaluation setup, including two custom analog front-end boards, the ATXmega256A3B-based I–V curve controller, and the "smart" load used to explore I–V surfaces during recording. Note that while only a single front-end board is needed for Ekho to function, we include two so that we can easily switch between experimental configurations. A shielded enclosure and shielded cabling are used to reduce induced measurement noise. An external NI USB-6356 data acquisition device (DAQ) (shown on the right) is used in our experiments to confirm Ekho's measurements. The DAQ can also be used to provide recording speeds that exceed the capabilities of the I–V curve controller when needed. For kinetic evaluation, the ATXmega256A3B was replaced with a "Teensy" ARM Cortex-M4 breakout board.

"Teensy" Cortex-M4 breakout board. The "Teensy" more memory and greater computational power when compared with the original ATXmega I–V controller. We also use the NI USB-6356 to collect voltage and current measurements during our experimental evaluation, as is described in the following section.

3.3.1.2 Smart Load

In order to support more accurate recording (as described in Section 3.2.2), we have developed a custom "smart" test load for our evaluation unit, that rapidly modulates its power consumption. This induces large fluctuations in supply voltage enabling more accurate recording. We have implemented this *smart load* using an Arduino Uno to control a digital potentiometer. The potentiometer [88] acts as a resistive load, with 128 settings ranging from 134 Ω to 100 kΩ of resistance. During the record phase the Arduino cycles through a predetermined number of these resistance settings randomly for a given time delay, producing a wide range of different load currents that explore different parts of the current I–V curve. Both prototypes use the smart load; however, the mobile version uses a 256 tap digital potentiometer with a wider resistance range of 120 Ω to 2 MΩ.

3.3.1.3 Desktop Emulation Unit

We anticipate that many designers will only be interested in the emulation function of Ekho, and will use previously recorded I–V surfaces to test with. Additionally, many of those interested in using the recording features, will do so with a separate unit, such as the Micro Ekho described in the next section. Therefore, a dedicated emulation module could be needed. We implemented an emulation only version of Ekho that functions as a Teensy 3.2 [99] daughter board, shown in Figure 3.13c. This unit can be powered over USB, but supports emulating I–V curves of up to 11.5V using the Analog Devices ADP1613 and AD623. In batches of 1000, including all components (except the Teensy 3.2), PCB fabrication, and assembly, we estimate the cost of the emulation unit to be only $11.47.

3.3.1.4 Mobile Recording

A key requirement of Ekho is that of mobility; it must be deployable to energy environments where sensors will be deployed. System designers can build in resiliency in their systems by recording I–V traces in future deployment environments, and then emulating those environments in the lab with Ekho on systems in development. To make the capture easier, we designed a mobile prototype that can be deployed and left in a future deployment area, powered by batteries, collecting I–V information to a microSD card. Once retrieved, the collected traces can be converted to I–V surfaces and emulated on hardware in development. Like the desktop version, the mobile version was designed to have as large an input voltage range as possible to accommodate the widest array of energy harvesters. We have developed two mobile recording prototypes to satisfy a variety of deployment scenarios. Figure 3.13b shows the smallest version; which is centered around an MSP430FR5728 as I–V curve controller. This mobile prototype, termed “Micro Ekho”, is meant for wearable, and wildlife tracking applications. In addition to the MSP430 microcontroller, Micro Ekho has a rectifier for AC signals, a CC1101 radio for communicating I–V information back to a basestation, and simplified “smart load” circuitry. The simplified “smart load” uses four N-channel MOSFETs with different resistance settings instead of a digital potentiometer. The MOSFETs are switched one at a time in rapid succession, exercising the attached energy harvester in the same way as the digital potentiometer, albeit with fewer stops. MOSFETs are a much cheaper and more power efficient solution, that allows recording higher voltage energy harvesters with a low voltage reference, suitable for deployment. In batches of 1000, including all components (except an enclosure and battery), PCB fabrication, and assembly, we estimate the cost of the

Micro Ekho unit to be only $14.14.

Our second mobile prototype, termed “Mini Ekho” (not picture) records I-V characteristics in the same way as the larger prototype. It is built around an off-the-shelf 96 MHz Cortex-M4 breakout board. The Cortex-M4 functions as the I–V curve controller; using it’s built in ADCs to sense voltage and current off the analog front-end. Mobile Ekho is mounted onto the processor breakout as a daughter board composed of the amplifiers and sense resistor that make up the analog front-end, the smart load (a 2 MΩ AD5242 256-tap digital potentiometer), the regulator, and a slot for a microSD card to store the raw surface data. Mobile Ekho can sense voltages up to 16 V and currents up to 0.5 A The entire system is powered by a Li-Po battery, regulated to 3.3V. The total cost of the mobile version of Ekho, including ”Teensy”, custom circuit boards, and parts is less than $45.

3.3.2 Software

In addition to the Ekho apparatus itself, we have also implemented the software necessary for recording, processing, and emulating energy environments for both the desktop, evaluation, and mobile versions. For recording on the desktop evaluation unit, we interfaced with the NI USB-6536 to record a physical energy environment. The NI USB-6536 is capable of sampling rates up to 1 MHz and we use usually use sampling rates between 200 kHz and 500 kHz in our experiments. The code used for processing recorded data, was implemented using a combination of python, C, and C++ to process and gather relevant data and generate I–V curves. We use the GNU scientific library [40] for polynomial fitting of I–V curves and surfaces from recorded I–V traces. We also used Numpy [3] and Scipy [62] to verify the quality of fit. We use R and Gnuplot for I–V surface visualization and data verification.

For recording on the mobile version, we implemented software that manages recording and storing of I–V point cloud data, that can then be converted into I–V surfaces using the above tools at a later time.

For emulating I–V environments, we used custom software, written in C, to handle timing on the PC that is responsible for appropriately timing traces and relaying data to the I–V controller as necessary. The microcontroller code is written in C and stores a curve in memory. It then constantly alternates polling an ADC for new voltage readings and a USART for new curve data. As voltage readings and curve data become available, it alters its DAC output and stored curve data appropriately. For emulating curves that change very fast, but are short in duration, the entire surface is kept in the memory of the I–V controller.

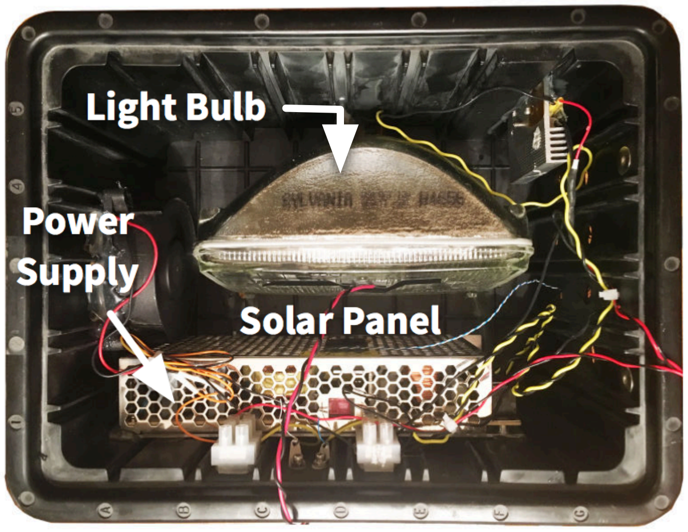


Figure 3.15: Our prototype light-box implementation provides a reproducible solar harvesting environment. We use it in our experiments to provide reproducible "ground truth" harvesting conditions. The implementation consists of an automotive headlight, a solar panel, an Arduino which serves as programmable dimmer-switch, and necessary power supply.

3.3.3 Programmable Energy Environments

The light-box used for much of the energy recording and emulating experiments is shown in Figure 3.15. Our light-box consists of a light source (an automotive headlight), which can provide different intensity settings depending on a PWM input. A solar panel is mounted inside the chassis which provides shielding from outside light sources. An Arduino Duemilanove-328 uses pulse-width modulation to drive a dimmer switch inside the light-box to control light intensity. This provides a relatively repeatable energy environment for comparison with Ekho.

To facilitate a repeatable and noise-free RF energy environment, we built a small Faraday cage out of brass screen and copper mesh seals, fully enclosed in a wooden box as shown in Figure 3.16. This RF-box effectively isolates the interior of the box from external radio, Wifi and other types of wireless interference. We mounted a programmable antenna connected to an Impinj Speedway Revolution UHF RFID Reader on the bottom of the cage to act as an energy source for RFID scale motes. By changing the transmit power of the antenna, many different I–V surfaces can be created. However, since each transmit power can generate thousands of different I–V curves, this is not always necessary.

We designed a repeatable kinetic energy harvesting environment we termed the "shake-table". It is

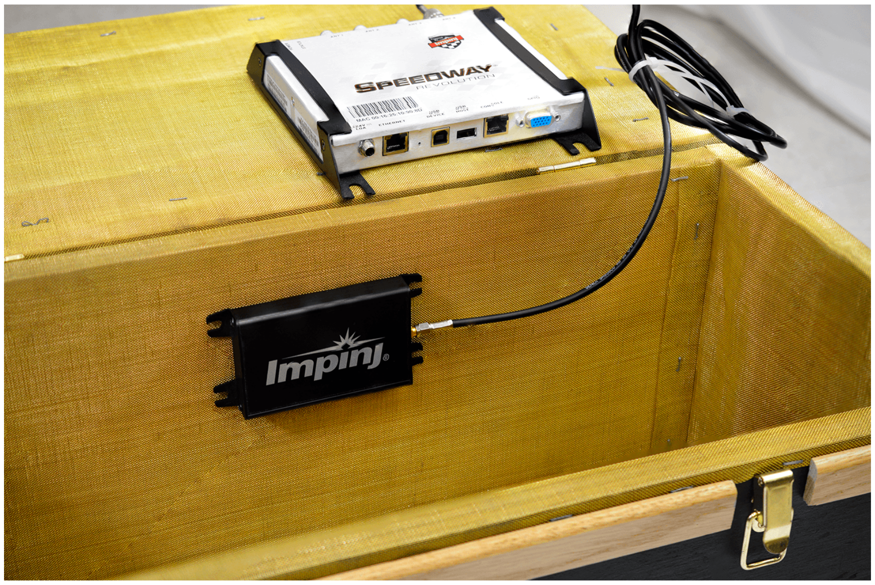


Figure 3.16: Our prototype RF-box provides a reproducible RF harvesting environment used in our experiments to provide "ground truth" harvesting conditions. The RF-box is composed of a wooden shell layered with brass screen and copper mesh seals. The copper mesh creates a Faraday cage, isolating the inside of the box from common RF interference. Inside the box is an antenna, driven by a programmatically controlled reader.

composed of a programmable signal generator with amplifier, and surface transducer, which excites a MIDE V21BL piezoelectric ceramic. As the ceramic is excited, it produces an AC voltage which is then rectified by an LTC3588 into a DC voltage, suitable for powering low power devices. We used the MIDE V21BL and LTC5388, but any combination of harvester and ceramic could be used. By modulating frequency and amplitude of vibration, different I–V surfaces can be generated.

3.3.4 Tools

When first attempting to profile and record I–V surfaces with Ekho, it is often necessary to view graphical representations of the I–V surface and I–V point clouds that inform the surface generation. We have developed graphical tools with C/C++ and OpenGL that render I–V point cloud data in real time to aid the designer in tuning Ekho to fully capture I–V surfaces. The ability to view the raw I–V point cloud, and the I–V surface it generates in real time, allow the designer to quickly make decisions on fit, scale, and usefulness of the surface under consideration. Besides providing 2D and 3D graphical representations of the I–V data, these tools also output helpful metrics (such as curves per second) that allow the designer to estimate space I–V curve controller memory requirements for recording and emulation.

Table 3.1: Replaying power to emulate Solar energy environment

	Replaying power with different training programs (flash writes)					
	Static training		**SemiAdaptive training**		**Adaptive training**	
Program	*mean*	*stddev*	*mean*	*stddev*	*mean*	*stddev*
Static	330.0	3	558.0	10	566.0	2
SemiAdaptive	214.0	10	414.0	5	469.2	24
Adaptive	29.6	4	16.0	3	5.1	1

All software drivers, tools, rendering programs, and hardware designs have been made available via our website[3].

3.4 Evaluation

In this section, we evaluate Ekho's ability to accurately capture energy harvesting conditions and consistently reproduce them in order to provide energy harvesting system designers with tighter experimental control, during testing. Specifically, we evaluate the consistency and accuracy of Ekho with respect to three programmable physical environments; the light-box, the RF-box and the shake-table. As a comparison, we also evaluate the previously discussed naive approach of replaying a recorded power trace (always replaying the same power, regardless of voltage). This comparison is conducted for a variety of different harvesting traces and loads (i.e. test programs). We also provide a more focused evaluation of Ekho's individual components (record and emulate) in order to explore the current limitations of Ekho and our prototype implementation.

In our experiments, Ekho was able to emulate solar I–V surfaces more consistently than our light-box, in terms of reproducing program behavior; in physical terms, Ekho is able to consistently produce solar I–V characteristics that vary by less than 68.7 µA (depending on capacitance) from test run to test run, emulating recorded solar I–V surfaces to mote-class devices running a variety of test programs. Ekho reproduces the solar I–V trace with a mean error of less than 77.4 µA from the recorded surface. Ekho reproduced kinetic surfaces with a mean error of 15.0 µA for medium impedance static loads, and slightly higher error for low, and high impedance static loads. Demonstrating the generality of Ekho; Ekho was able to emulate RF I–V surfaces significantly more consistently than the RF-box, for three (3) different transmit powers. In our experiments Ekho was able to reproduce RF energy harvesting conditions effectively such that program behaviors were accurate in comparison to the RF-box. In contrast, we also show how the naive approach of emulating constant power produces behavioral results that are inconsistent with the light-box, for battery-less, energy harvesting

[3]http://persist.cs.clemson.edu

devices, and inadequate for predicting the performance of these small devices in deployment.

3.4.1 Methodology

Our evaluation involves emulating a total of 10,647 solar I–V curves, generated from 27 different randomly generated light-box traces (ranging from 6 seconds to 5 minutes in length), for a total of 1,029,000 solar I–V curves tested. We also record or emulate a total of 599 kinetic I–V curves, generated from four different randomly generated shake-table traces (ranging from 500 milliseconds to 10 seconds in length), for a total of 2995 kinetic I–V curves tested. In our evaluation comparing the accuracy of emulating constant power versus emulating I–V, we emulate a total of 3408 constant power curves, generated from three program's harvested power traces. We emulate a total of 320 RF I–V curves, generated from three different recorded transmit power levels, for a total of 6400 RF I–V curves tested.

Emulation Platforms: Two computational platforms were used for emulating I–V surfaces during the evaluation of Ekho. The first, an XMega microcontroller, was used to emulate Solar and RF surfaces; the second, an off-the-shelf Teensy ARM Cortex-M4 breakout board, was used to emulate kinetic I–V surfaces. The ARM Cortex-M4 was overclocked to 96 MHz and used native USB for curve updates, allowing for much faster emulation (and better error results as shown in Table 3.6). The XMega microcontroller ran at 32 MHz and was limited by the speed of the onboard UART for curve updates. The XMega microcontroller was used for all solar and RF I–V surface emulation, while the Teensy ARM Cortex-M4 was used for all kinetic I–V surface emulation.

Test Devices: For test devices in our solar and constant power experiments, we use the EZ430-RF2500, a mote-class device produced by Texas Instruments, that consists of a MSP430F2274 ultra-low power microcontroller and a low power, 2.4 Ghz CC2500 radio; a 10 µF capacitor is used to store energy. For our RF experiments, we use the UMich Moo [136], an ultra-low power CRFID platform built around a MSP430F2618 microcontroller, and RF harvesting hardware. No batteries were used as power sources in any experiment, each mote device is powered exclusively from energy harvested and held in small capacitors. The static loads for kinetic testing were implemented using a digital potentiometer.

Programs: For solar experimentation; the EZ430-RF2500 devices run three different programs—Static, SemiAdaptive, and Adaptive—that provide different power consumption profiles and represent behaviors commonly seen in sensing applications. All three periodically read from the MSP430's internal temperature sensor and store the value to the sensor's internal flash memory. Between readings, all three programs put the the processor to sleep to conserve energy. They differ in how they manage energy. Static maintains a steady sampling rate regardless of energy availability. SemiAdaptive reduces its sampling rate when its voltage drops below a set threshold (2.3 V), in order to spend more time asleep and hopefully avoid a power failure. In addition to reducing its sampling rate during low energy conditions, Adaptive also increases its sampling rate when the its capacitor voltage exceeds a predetermined threshold (2.7 V), using its excess energy to collect more data. For RF experimentation, the Umich Moo devices run one program—Sense-and-CRC—that senses the internal temperature of the MSP430 using an on-board ADC five times, averages the readings, then performs a cyclic redundancy check (CRC) on the resulting data.

Harvesting Traces: We use the light-box, described previously, to provide a reproducible physical environment to serve as the ground truth for our solar experiments. We generate light-box traces, by randomly choosing a small number of light intensity settings distributed over a short amount of time, and interpolating those points using cubic splines, with exact boundary conditions. This is done multiple times to produce sets of different light-box traces. To test responsiveness of Ekho each of the solar traces changes much more rapidly than what would be seen in an outdoor deployment, with variations every 60ms. RF harvesting traces are generated for us by the inherent volatility of an RF reader, for our evaluation, we only modulate the transmit power. Despite this, RF traces are naturally more frantic and change more rapidly than any solar traces generated. Kinetic harvesting traces were generated by the shake-table by modulating the frequency and amplitude of the vibrations. Kinetic harvesting traces were especially sensitive to ambient vibrations and noise and were very difficult to reproduce consistently.

I–V surfaces: From the randomly generated light-box traces, the transmit power traces gathered in the RF-box, and the kinetic shake-table traces, I–V surfaces are generated using the previously mentioned *smart*-load. Parameters such as cycle frequency (number of times a second the smart load goes through all its resistance settings), and capacitance are chosen so as to give the best results for each surface type. Choosing different capacitance or period values can have a significant effect on the final granular accuracy of the recorded surface, as discussed in Section 3.2.2.3. Drawing on those observations, surfaces generated for RF emulation were gathered with smoothing capacitance <0.1 µF and very high cycle frequency, while the solar surfaces had smoothing capacitance 10 µF and much lower cycle frequency. When capturing kinetic surfaces, no smoothing capacitance was added, as the computational platform used for emulating kinetic I–V surfaces was significantly faster than the one used for solar and RF.

Constant power surfaces: Using the light-box traces mentioned above, we generate constant power surfaces from recorded power traces captured as different programs execute. Each power surface is generated from a single recorded trace chosen arbitrarily from a set of device runs. We create constant power surfaces for each of the three EZ430-RF2500 programs mentioned while running on a light-box trace.

Distance metrics: To evaluate the physical accuracy of Ekho, a metric was needed to compare two I–V curves. This is difficult for two reasons. First, an I–V curve relates two incommensurable units (Volts and Amperes), this renders as meaningless any euclidean distance from the curve. Second, there is not a one-to-one mapping between an observed (I,V) pair and an emulated (I,V) pair. The observed point could correspond to any number of points on the curve being emulated. In our development of Ekho, we have explored two metrics, current error (assuming the observed voltage is correct and measuring the difference in current) and voltage error (assuming the observed current is correct and measuring the difference in voltage). The current error is amplified (even for points very near the surface, as shown in Figure 3.17) when the voltage is high and the I-V curve is steep. Voltage errors are similarly amplified when the voltage is low, and current is high.

Using these emulation platforms, test devices, test programs, programmable environments, harvesting traces, surfaces, and metrics, we evaluate Ekho's ability to record and recreate energy harvesting traces produced by the light-box, RF-box, and shake-table. We measure the accuracy and consistency of Ekho, explore the the experimental characteristics that affect the system's performance, and demonstrate the generality of Ekho. We attempt to answer these questions:

1. *How consistent and repeatable is Ekho?*
2. *How accurately, in terms of behavior and physical conditions, can Ekho emulate energy environments?*
3. *Is Ekho able to record and emulate multiple types of energy harvesting environments effectively?*

Before we can explore these questions, we first need to understand exactly why simulating constant power is not sufficient for accurate testing of small, capacitor powered energy harvesting devices, we show our results in Section 3.4.2. We then explore solar harvesting consistency results of Ekho in Section 3.4.3, and show solar recording and emulation accuracy of Ekho in Section 3.4.4. We show consistency and behavioral accuracy results of Ekho with regards to RF energy harvesting in Section 3.4.5. We then study the accuracy of recording and emulation of kinetic energy sources using Ekho in Section 3.4.6. Lastly, we compared the recording performance of the desktop version of Ekho and the mobile version in Section 3.4.7.

3.4.2 Emulating P vs. I–V

To evaluate our claim that emulating power is not sufficient to simulate an actual energy harvesting environment in a deployment, we created constant power traces gathered from executing our three sample programs (Static, SemiAdaptive, and Adaptive). Each power trace was generated from one run of a program. We then compared the behavior (flash writes) by running each program on the light-box five (5) times, then on the Ekho generated I–V surface five (5) times, then on all three of the generated constant power-surfaces five (5) times each. We measure differences in program behavior by recording the number of successful writes to flash memory that were performed by each test run. Table 3.1 shows the results of emulating constant power using power traces recorded at program execution. By using constant power to emulate what is actually an I–V curve, behavior (here shown as flash writes) is dramatically different than deployment behavior as compared to the light-box behavioral results in Table 3.2. Table 3.1 shows that constant power-surfaces generally underestimated or overestimated available energy for all programs except (as would be expected) the program that the constant power-surface was generated from. In some cases the error was very apparent; when using Static as a training set, running SemiAdaptive gave half the amount of expected flash writes; when using SemiAdaptive as a training set for a constant power surface, and running Static on that surface, the flash writes were severely overestimated.

The choice of which run to use to generate the power trace can have a large impact on the emulated behavior. For example, if an outlier run (where lower or higher than average flash writes occurred) was arbitrarily chosen as a training set to generate a constant power surface, programs ran on the generated surface could significantly differ from the average run. This is apparent in the Adaptive column and row of Table 3.1, where runs on the Adaptive Training surface produced significantly less flash writes than the average Adaptive run on the light-box shown in Table 3.2. These results confirm what was previously shown in Figure 3.7, emulating power is not effective for devices with a volatile supply voltage.

3.4.3 Reproducing Program Behavior

The primary objective of Ekho is to make device behaviors consistently repeatable in spite of variations due to the energy harvesting. Our first experiment examines Ekho against this goal.

In this experiment, we recorded a randomly generated light-box trace using Ekho. We then use Ekho to emulate the recorded surface fifteen (15) times, five (5) times using each of our three test programs as the test device. As a point of comparison, we also run each of our three test programs five (5) times powered by the

Table 3.2: Program behaviors are shown for three test programs, when harvesting energy from the light-box and from the Ekho emulator.

	Program behavior (flash writes)			
	Ekho		**Light-box**	
Program	*mean*	*stddev*	*mean*	*stddev*
Static	326.0	4	291.6	8
SemiAdaptive	441.2	14	418.0	35
Adaptive	11.6	9	11.2	11

Table 3.3: Physical consistency over multiple runs is shown with the three test programs. The same light-box traces were recorded, and then emulated by Ekho with the same load to compare the physical consistency and repeatability of experimentation. Ekho performs with nearly the same physical consistency as the light-box.

	Physical error by program	
	Ekho	**Light-box**
Trace	*mean*	*mean*
Static	66.3 µA	50.6 µA
SemiAdaptive	72.6 µA	56.2 µA
Adaptive	67.3 µA	53.1 µA

light-box directly, using the same randomly generated trace. We measure differences in program behavior by recording the number of successful writes to flash memory that were performed by each test run. As we noted in Figure 3.8, the lightbox variation would increase as temperature increased after long and consistent use. To account for this, between each lightbox run, we waited one minutes to allow the lightbox to achieve thermal equilibrium. Ekho does not suffer from this inconsistency problem, and is not susceptible to temperature variations.

Table 3.2 shows the results of this experiment. For each program the average number of flash writes per test run, and the standard deviations are shown for both the Ekho and the light-box test runs. This table shows the result for a single light-box trace; however, we have found these results to be consistent across all of the randomly generated traces, we have tested. For all three programs, the behavior of the devices under Ekho emulation closely approximates the ground-truth behaviors. Behaviorally, Ekho was more consistent and had a smaller standard deviation in flash writes for each test program. In each case, Ekho emulation was comparable in physical consistency with the light-box output as shown in Table 3.3.

Table 3.4: Emulation error—the distance of an emulated point from the intended I–V surface—is shown for the three test programs, while emulating a Ekho-recorded randomly-generated light-box trace. Ekho has a slightly higher error rate on the high voltage, low current area of the I–V curve, this is because slight changes in voltage are accompanied by large changes in current; however, this area of the curve is generally avoided as it denotes an inefficient use of harvested energy.

	Emulation error	
Program	*mean*	*stddev*
Static	87.2 µA	46.7 µA
SemiAdaptive	86.9 µA	46.2 µA
Adaptive	88.9 µA	72.5 µA

3.4.4 Emulating Solar Energy Sources

Ekho is designed to make program behaviors deterministic, by accurately and consistently reproducing the physical energy harvesting environments that determine program behaviors. In order to determine how well Ekho reproduces the physical energy harvesting environment, we also measure the characteristics of the emulated energy harvesting conditions.

Table 3.4 shows the measured current error between the emulated surface and the intended I–V surface. For each program the mean emulation error (the distance in µA from the emulated surface to the intended surface for a voltage) is shown, as well as the standard deviation of this error. The choice of current error is arbitrary, as voltage error could also be used to the same result; both values are derived through the mechanisms discussed in Section 4.2.4.1. These results are for a single, representative light-box trace; other traces tested produced comparable results. For all programs, Ekho was able to reproduce I–V surfaces accurately enough that behavior remained consistent.

Emulation error is also influenced by the natural shape of the I–V curve as shown in Figure 3.17; on the high voltage part of the curve, past the knee the error rate increases as the slope of the curve increases, since minimal changes in voltage come with large current changes. While Ekho was not as consistent in emulation error in this area of the curve, this is not as important, as energy harvesting sensors start wasting energy (that could power useful computation) when they enter the steep high-voltage end of the curve that denotes a full capacitor with minimal processing (therefore minimal current draw). For example, a deep sleep program (refer to Figure 3.2). This is a departure from a traditional sensors application paradigm; which uses duty cycling to prolong battery life, and therefore extend the lifetime of the sensor.

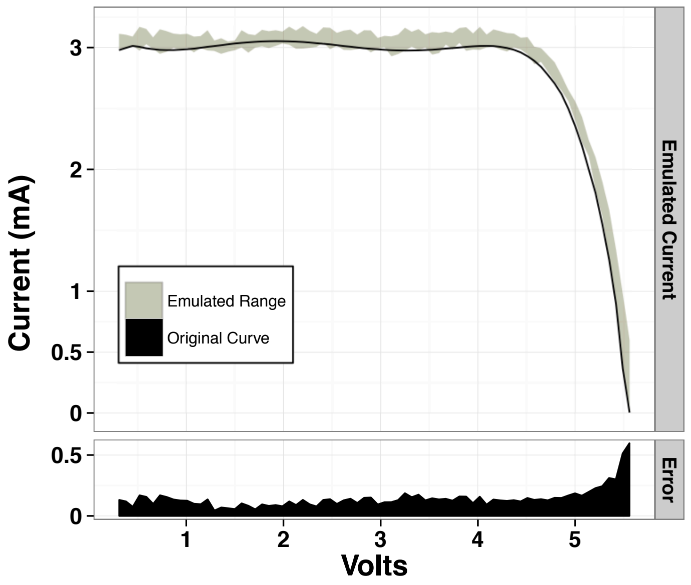


Figure 3.17: Shown in the top part of this figure are the target curve Ekho is emulating, and the actual range emulated for a number of test runs on a program. The bottom part of the figure shows the error amount for different parts of the emulated curve. As the slope increases, the current error increases past the knee of the I–V curve.

3.4.5 Emulating RF Energy Sources

In this experiment we profiled the program behavior of devices powered through the RF-box (detailed in Section 5.3). We then compared those results to the same device powered by Ekho instead. To profile RF-box behaviors, we placed the detached RF energy harvester from a Moo inside the RF-box over the reader antenna. We then took another Moo and connected it to the output of the harvester, outside of the Faraday cage so that the RF signals would not interfere with each other. This provided a repeatable, programmable RF energy environment that served as a ground truth for all our RF experiments with Ekho. We ran similar consistency experiments as were run on the light-box; the Moo under test was programmed to sense its internal ADC for temperature five times, average that data, then perform CRC on the data. It did this as many times as it could before brownout. In this experiment, we did not write to flash as in other programs due to the power and voltage requirements, which are difficult to achieve in ultra-low power CRFID platforms. Using the DAQ and Ekho, we monitored the number of CRC calculations performed and when they occurred, and the times that the Moo lost power. We conducted these experiments multiple times and found that the RF-box was reasonably consistent (as shown in Table 3.5) in regards to program behavior. However, we found that at low transmit power, the consistency of the RF-box deteriorated dramatically. We attribute this to timing issues

Table 3.5: Program behaviors are shown for the Sense-and-CRC test program running on the UMich Moo, when harvesting energy from the reader inside the Faraday cage and from the Ekho emulator (emulating the recorded RF I–V surface). The number of successful Sense-and-CRC readings were counted and compared. Additionally, the total harvested energy is shown for each transmit power. For all transmit powers, the Ekho-powered devices closely approximate the ground truth behaviors. Ekho reproduces these behaviors with significantly better consistency than the RF-box, especially for lower transmit power. Note that since the entire surface is stored in RAM of the I–V curve controller, the error rates are much lower than with solar emulation.

		RF program behavior (CRC)					
		Ekho			**RF-box**		
Transmit power	**Harvested energy**	*mean*	*stddev*	*error*	*mean*	*stddev*	*error*
+21.25 dBm	0.55 mJ	23.6	0.6	2.3%	21.0	8.3	39.4%
+27.75 dBm	2.57 mJ	208.7	0.7	0.3%	189.2	39.1	20.7%
+32.5 dBm	3.88 mJ	237.3	1.3	0.5%	266.2	12.5	4.7%

with synchronizing the RF-box, the DAQ, and the host computer. Using a dedicated (but expensive) signal generator could provide a more repeatable RF energy environment at low transmit power.

To evaluate Ekho's behavioral performance with volatile RF energy, we recorded and then constructed an I–V surface generated from each of three different transmit power levels using Ekho. To vary the I–V surface, we at first varied the transmit power over time; this proved unnecessary as an RF surface changes upwards of 1000 times a second for an arbitrary transmit power. We then used Ekho to emulate each recorded surface nine (9) times, for the Sense-and-CRC program running on the UMich Moo. We measure differences in program behavior by recording the number of CRC calculations that were performed by each test run.

Table 3.5 shows the results of this experiment. For each program the average number of CRC calculations per test run, the standard deviations, and the error rates are shown for both the Ekho and the RF-box test runs. This table shows the results for RF traces that were 120 ms in length. While this may seem a short timespan, because of the volatility of RF energy, eighty (80) different I–V curves were emulated in this window. In all cases, the behavior of the devices under Ekho emulation closely approximates the ground-truth behaviors of the RF-box. Behaviorally, Ekho was significantly more consistent and had a smaller standard deviation and error rate for each transmit power, but especially for the lower transmit powers.

3.4.6 Emulating Kinetic Energy Sources

In this experiment we captured a kinetic I–V surface using the shake-table (detailed in Section 5.3) and Ekho. Using the emulation feature of Ekho we ran three different constant loads over the surface, and recorded the emulation error (difference between the expected and emulated current) for each, similar to the

Table 3.6: **Emulation accuracy for kinetic I–V surfaces.**

	Emulation error		
Load	*energy*	*mean*	*stddev*
Low Impedance Load	709.7 µJ	23.7 µA	3.5 µA
Medium Impedance Load	713.8 µJ	15.0 µA	1.8 µA
High Impedance Load	685.1 µJ	38.7 µA	2.3 µA

experiment in Section 3.4.4 and Table 3.4. The results of this experiment are shown in Table 3.6. The total energy, mean distance from the expected curve, and standard deviation from the expected curve are shown. As shown, Ekho is able to reproduce kinetic I–V surfaces on medium impedance loads to within 15.0 µA. For high and low impedance loads for this kinetic I–V surface, emulation error is slightly higher. Physical emulation results for kinetic I–V surfaces are notably better in terms of accuracy than solar. This is not because of some inherent difference in the harvesters, but is owed to the updated, faster hardware used in emulating kinetic surfaces using the ARM Cortex M-4 as opposed to the XMega. If previous solar experimentation were replicated using the newer hardware, we expect emulation error would be reduced.

3.4.7 Mobile Ekho

For our evaluation, we compare the recording performance of the desktop version and the two mobile versions in terms of size, applicability, and energy requirements. Both mobile versions of Ekho were designed with three goals in mind, namely: 1) to be small enough to be deployable, 2) to be an all-in-one solution for recording different I–V surfaces, and 3) to have a low enough power budget to be deployed for a week or more.

Mini Ekho: Mini Ekho is nearly ten times smaller than the desktop version; the desktop version is 97.79 mm x 75.87 mm, while Mini Ekho is 21.92 mm x 36.32 mm. This small form factor enables easy deployment and testing.

The mini version satisfies the all-in-one requirement by replacing the PC and DAQ with a dedicated ARM microcontroller. The desktop unit requires extra hardware support (a DAQ for I–V measurement and a PC for surface management) that is built in to the ARM microcontroller. While some accuracy is sacrificed per sample as the DAQ uses a 16-bit ADC and the mini version has a 12-bit ADC, this is overcome by averaging multiple samples to approximate an I–V surface. While recording speed on the mini version is lower than the desktop version by a factor of two, this is a non-issue, as recording speed of the mini version is still over 1000 curves per second (depending on the smart load duty cycle); well beyond the maximum emulation speed of Ekho. Mobile Ekho also has a greater I–V surface range because of better component choice for the digital potentiometer; while desktop Ekho can sense currents as low as 10 A, mini Ekho can sense as low as 0.05 A, allowing a wider evaluation of potential energy environments.

The mini and desktop units are very different in terms of energy usage. Desktop Ekho must be plugged in and draws a minimum of 100 mA at 9 V. Mobile Ekho can be powered by a Li-Po battery, and typically draws 38 mA when recording and 138mA when writing a page to the SD card. Mobile Ekho can last for over two weeks in deployment in a solar environment powered by a 1750 mA h battery at 3.6 V, recording a single I–V curve a minute during the daylight hours.

Micro Ekho: Micro Ekho, is 34 mm x 34 mm, not including the coin cell battery and enclosure. Just as with Mini Ekho, Micro Ekho is an all-in-one solution, but tuned for ultra low power operation, allowing longer, connected deployments. Micro Ekho has reduced accuracy and precision, as it only has a 10-bit ADC, and can only acquire five different points on the I–V curve at a time. However, the long lifetime and long range radio allow it to be placed semi-permanently.

3.4.8 Complicating Factors

Different decisions when recording, and emulating I–V surfaces influence the final accuracy and consistency of Ekho. Because of the hardware limitations of the I–V controller (specifically the serial communication speed), Ekho using the XMega microcontroller emulation platform is only able to emulate I–V surfaces that switch curves at a maximum rate of 135 Hz. That speed can be increased by using the ARM as the emulation platform. Since RF surfaces can switch curves upwards of 1000 times a second, when emulating RF, those surfaces must be held entirely in RAM on the XMega microcontroller or ARM I–V controller. This allows a curve switching speed beyond 1 kHz (for 65 point I–V curve representations), but limits surface length to sub-second levels on the XMega microcontroller, and 1-2 seconds on the ARM Cortex M4. To simulate longer surfaces, either a larger RAM must be used, or other types of fast access external memory be made available (FRAM, or an SD card), or a faster BUS implemented in hardware. Another factor is response time—the I–V curve controller on the XMega microcontroller is able to respond to current fluctuations in 4 µs. Added capacitance in the circuit increases accuracy as it allows more time for the controller to respond to changes in current, but this also decreases the maximum effective curve switching rate. This can cause Ekho to skip curves when emulating, and thereby reduce accuracy of the whole surface. In emulation, the speed at which the controller can deliver curves limit the range of surfaces that can be emulated. Also, the number of points used to represent a curve in memory influences this. Choosing fewer points (for example 33 instead of the current 65) can increase emulation speed. Tradeoffs between emulation speed, and accuracy can be made in multiple places throughout the recording/emulation chain. Many of the same tradeoffs detailed in Section 3.2.2.3 apply to emulation as well.

3.5 Related Work

We proposed the conceptual framework for Ekho at HotPower 2011 [137] and introduced the idea of I–V curve-based emulation of energy harvesting conditions, but presented only a cursory evaluation of a prototype that lacked the ability to record and estimate I–V curves and could only emulate single static I–V curves with error rates as high as $170\mu A$. Our work builds on the ideas proposed in HotPower 2011 [137], and demonstrates that I–V surfaces can be accurately recorded and emulated, in order to provide both realistic and repeatable experimentation.

Emulation Techniques: Another closely related project, SunaPlayer [10] provided an analog hardware solution, specifically designed for larger solar harvesting applications, which used a high gain Darlington transistor to approximate the shape of a solar I–V curve. They also used a model of solar output based on temperature, humidity, and ambient light conditions to capture solar harvesting conditions. While this approach does allow for capture and replay of energy harvesting conditions, this approach is limited to a single energy harvesting technology (solar), while Ekho can record and emulate Solar and RF. Additionally, the latency of converging to a specific point on the I–V curve for emulation can reach as high as ten seconds, which is prohibitive for some applications. Ekho converges on an emulation point in a matter of microseconds.

Analog battery simulator B# [22, 97] is tangentially related to Ekho, in that it measures a current load, then computes a voltage from a battery simulator and mimics that voltage on a regulator. However, B# is only applicable to specific battery chemistries, and does not support recording of an actual I–V surface (it uses a battery simulator as a voltage lookup), nor does it have the ability to emulate volatile energy sources like RF and kinetic, or even solar. Ekho provides a more generalized approach that works with surface mount capacitor powered devices and does not rely on computationally expensive simulation models or profiling specific battery chemistries. Ekho is not a battery simulator, nor is it meant for devices that use batteries.

S# extends B# by emulating large wattage solar panels[76], with the intention of saving designers time by testing in the lab, before deployment, much like Ekho. As opposed to Ekho, S# ignores I–V characteristics and instead uses a simplified approach to emulation, by taking the input current and the amount of sunlight to determine the final power generated at the load. This approach works well because the devices S# is meant for have stable supply voltages. Ekho is specifically engineered to work with unstable supply voltages. Additionally, like B#, S# only works for solar cells, and a very specific range of solar cells (1-24V). Ekho can work with input currents and voltages much lower, fitting in the vision of enabling extremely power constrained, capacitor based, energy harvesting sensors. This also allows Ekho to emulate all the energy harvesting sources we highlighted in this dissertation.

EmPro [96] is an wireless sensor emulator that takes into account environmental variations, as well as sensor inputs. EmPro emulates power sources (only batteries, and solar panels), radio attenuation, and plays back sensor inputs. EmPro does not however, take into account how the load of a sensor can change the voltage, and therefor the amount of energy harvested. While EmPro provides a more holistic approach by attempting to emulate most of the concern of a wireless sensor node, EmPro is constrained to only two power sources, high wattage solar panels, and batteries. Ekho generalizes the energy harvesting problem that EmPro ignores, allowing for different hardware and software sensing solutions to be tested against multiple energy sources without changing designs.

Model Driven Simulation: Other related work include simulation tools for low-power sensors [117, 87, 35, 36, 8], some of which support RFID-scale sensing by considering I–V relationships when simulating harvesting conditions [47]. As described previously, these techniques are able to provide many of the same benefits as Ekho, but at a higher maintenance cost (models need to be updated to support new hardware). Recently, some systems has attempted to simulate intermittence and energy harvesting[2, 27]. When emulating using a model, a user must record every feature of the physical world that is relevant to the model. This often makes models very specialized, making them useless when new harvesting paradigms are developed. Ekho records the electrical properties of the harvester in question. It doesn't need different sensor readings for each harvester. Harvester models are complementary to Ekho, as the I–V surfaces Ekho (since they are in a digital format) could be used as input to other simulators like the ones mentioned. Even instruction level simulators like [35] could benefit from using Ekho I–V surfaces to inform their energy model, we discuss this further in Section 5.6.

Profiling and Efficiency Tools: Finally, a number of tools make it possible to measure and characterize the energy consumption of embedded devices and harvesters [60, 120, 127], simulate a battery under charge conditions[22, 97], and step debug in intermittent programs[24]. In addition to enabling energy aware software systems, we envision these technologies being extended in order to allow sensor devices to profile their own harvesting conditions in order to better predict their future energy budgets and operate more closely to their power harvesting potential.

3.6 Discussion

Ekho is designed to be the multipurpose tool for recording and emulating a wide range of energy harvesting environments, provided in a single integrated package. Based on the results described in the previous section, Ekho promises to make it possible to experiment with a wide range of low-power, energy harvesting devices, to an extent that has, to date, been infeasible. In spite of this promise, our current implementation is limited in a number of important ways. This section discusses these limitations and our efforts in the coming months on future work.

Ease of Use: Ekho is meant to simplify design, testing, and experimentation with tiny, batteryless sensors. This is achieved with a simple recording and emulation interface, and a suite of software tools. To record with Ekho, the developer only has to plug in the leads of the energy harvester of interest, into the receiving inputs of Ekho, and then either deploy the Ekho (most common for solar), or actuate the harvester in the lab (for example with an RFID reader). The generated I–V surfaces are then recorded and digitized to a desktop computer, or microSD card for later use. To emulate with Ekho, all the designer has to do is replace the energy harvester on the device that will be tested with Ekho. After connection, the developer can use the developer tools (detailed in Section 5.3) that allow visualization of recorded I–V surfaces as well as the ability to watch emulation in progress.

Hardware Constraints: Many of the limitations of our Ekho prototype stem from current hardware restrictions that can be overcome by readily available parts. The 7.4 ms curve update limitation when emulating energy environments is the result of speed constraints imposed by the USB-serial port on the XMega microcontroller. We have shown that we can overcome this limit using on-board RAM to store surfaces, which was necessary for RF emulation. We have also shown that managing the emulation loop with the Teensy ARM Cortex-M4 microcontroller, improves the error results, as curve updates happen sub-milli-second. Moving our emulation platform to be exclusively on the ARM Cortex-M4 will allow Ekho to emulate I–V surfaces that require more frequent I–V curve updates, and will also allow longer RF I–V surfaces. Replacing our current 12-bit ADC and DAC with faster 16-bit models will also improve accuracy and measurement ability.

Smart Load: The synthetic "smart load" used to explore the I–V surface could potentially be invasive or detrimental to the harvester under test. Some harvester sources (such as solar) can generate a significant amount of energy that if they have an excess of energy not used by the load. If this energy is not discharged after some amount of time, it will burn off in the form of heat, potentially damaging the harvester or changing the I–V surface[4]. The "smart load" may have the potential to cause this type of situation. In our experiments, we have not characterized or observed this effect for the small, low energy harvesters that Ekho is designed for, possibly because the "smart load" explores the surface very quickly, not allowing excess energy to be stored for long in the harvester. We expect this could become a problem when using Ekho to record much larger wattage systems (specifically solar panels). To address this for larger energy harvesting systems, designers could tune the load range of the "smart load" using information about the harvester under test. By constraining the range and avoiding the edges of the harvester's I–V curve, this heat effect could be avoided, using information that would already be in hand. However, we consider this an open question, and are interested in exploring, and characterizing this effect in small harvesters that are used in batteryless sensors.

Environmental Factors: Ekho does not necessarily take into account source-centric concerns that affect the performance of the energy harvester. For example, with solar energy, the irradiance and the temperature are not taken into account. For kinetic energy the accelerations (and possibly angular velocity) are ignored. For RF harvesting, the source attenuation and humidity of the air are not considered. Ekho records the electrical properties of the harvester in question. It doesn't need different sensor readings for each harvester. This makes Ekho more general than other approaches. However, since Ekho generates a digital representation of an energy harvesting surface, it is not far fetched to imagine that these other environmental conditions can be tagged as meta-data to any surface that is recorded. For example: an solar I–V surface could be recorded alongside a temperature and luminance sensor. This serves the purpose of mapping the environmental conditions to the I–V surface of a particular harvester. This metadata could be kept together along with location, time of day, and other information, gibing designers more information on how different energy harvesters work in different conditions. We envision creating a globally available repository of I–V surfaces tagged with this metadata, that will allow researchers to coordinate and share environments and results in a common format.

Automation: We also plan to explore ways to automatically tune some of Ekho's system parameters, like the window size used to infer I–V curves, the smoothing capacitance, and the custom "smart" load cycle frequency during record. Automatically detecting the window size by actively detecting the level of movement across an I–V curve and adjusting the window size to use the smallest possible complete data set, will make Ekho easier to use and improve recording precision. Automatically adjusting the period and capacitance of the smart

[4]However, this would likely not change the device behavior as this would only happen when the device has more than enough energy to perform its tasks.

load will also make the recording process more straightforward. We anticipate that simple switching circuitry could change the capacitance, per test run, however, as demonstrated with our newest version of Ekho, often this smoothing capacitance is unnecessary if the recording speed is fast enough. We anticipate not using any smoothing capacitance in future versions of Ekho.

Integration: Simulation approaches are often preferred when developing new protocols and applications for sensor networks. This allows testing, debugging, analysis, and system scalability investigations when time is limited and cost is a factor. Well-known sensor network simulators such as Cooja [37] for Contiki [31] and TOSSIM [74] for TinyOS [75] only have a power consumption model, i.e. Energest [32] and PowerTOSSIM [116], respectively, for better understanding the energy consumption of sensor network applications. The lack of support for emulating energy harvesting conditions limits the development of energy harvesting-based applications in these simulators. A tool that utilizes environmental data to compute the harvested power in Contiki is available [28]. However, unlike Ekho, this tool estimates the amount of energy harvested using traces of light values with some strong assumptions for harvesting efficiency, charging efficiency and no battery leakage. With Ekho, it is possible to feed the simulators with traces from gathered I–V surfaces to simulate sensor networks with more realistic energy harvesting setting.

We find this integration most interesting for instruction level device simulators, especially MSPSim [35]. One of the main limitations with Ekho (and emulation in general) is that the emulation can not be sped up, so it becomes difficult to exhaustively test many different harvesting conditions. By combining the speed of simulation with the accuracy of Ekho I–V traces, hundreds or thousands of energy harvesting conditions could be explored for a particular device or application, without spending a significant amount of time. This has been implemented in the MSP430 based SIREN simulator, which builds on Ekho [39].

3.7 Conclusions

In this chapter, we have described the design and evaluation of Ekho, an emulator that makes reproducible experimentation with capacitor powered energy harvesting devices possible, without the need for hardware and harvester models (required by simulators). Ekho is able to record energy harvesting conditions and accurately recreate those conditions in a laboratory setting consistently. We have also described the design and use of a mobile version of Ekho; allowing profiling of energy environments at locations where sensors are already in place, or will be in place.

Ekho is a general-purpose tool that supports a wide range of harvesting technologies. We have demonstrated, using a working prototype, that Ekho is capable of reproducing harvesting-dependent program behaviors by emulating solar energy harvesting conditions accurately to within 77.4 µA, and more consistently than our light-box, a programmable solar harvesting environment. We also found that Ekho can reproduce kinetic energy environments with a mean error of 15.0 µA from the recorded surface. Demonstrating the generality of Ekho; we have shown that Ekho is able to emulate RF I–V surfaces significantly more consistently than our RF-box, for a variety of loads and I–V surfaces. Ekho was able to reproduce RF energy harvesting conditions effectively such that program behaviors were accurate in comparison to the RF-box.

As embedded sensing devices continue to become smaller, with tighter energy constraints, energy harvesting will continue to become more important, and tools like Ekho will make possible the realistic and thorough testing that will be needed to deploy those devices with confidence.

CHAPTER 4

FLICKER: HARDWARE PLATFORM FOR THE INTERNET-OF-THINGS

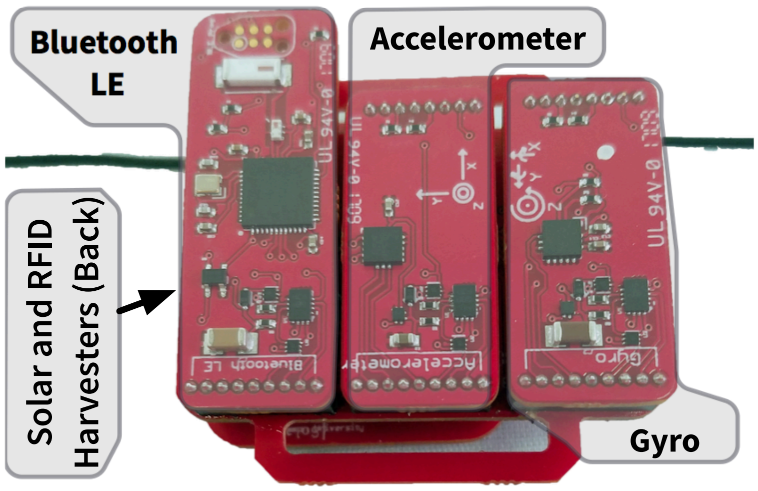


Figure 4.1: Flicker enables rapid prototyping of batteryless, intermittently powered sensing systems. This figure shows a Flicker device equipped to harvest RFID and solar energy, sense acceleration and rotational velocity, and communicate via a CC1101 Radio and backscatter.

Batteryless, energy-harvesting sensing systems are critical to the Internet-of-Things (IoT) vision and sustainable, long-lived, untethered systems. Unfortunately, developing new batteryless applications is challenging. Energy resources are scarce and highly variable, power failures are frequent, and successful applications typically require custom hardware and special expertise. In this paper, we present Flicker (shown in Figure 4.1, a platform for quickly prototyping batteryless embedded sensors. Flicker is an extensible, modular, plug and play architecture that supports RFID, solar, and kinetic energy harvesting; passive and active wireless communication; and a wide range of sensors through common peripheral and harvester interconnects. Flicker's software tools automatically detect new hardware configurations, and simplify software changes. We have evaluated the overhead and performance of our Flicker prototype and conducted a case study. We also evaluated the usability of Flicker in a user study with 19 participants, and found it had above average or excellent usability according to the well known System Usability Survey.

Software solutions for managing energy and keeping time on transient power fall short of enabling sophisticated applications on batteryless energy harvesting sensing devices. Developing a new batteryless application still requires specialized expertise, custom hardware development, and considerable hardware tuning. THe following key challenges must be addressed before most developers can create batteryless sensing applications.

Limited Hardware Options: The Intel WISP [115] and its descendants [136], have long been the platform of choice for computational RFID (CRFID) research, but the wide range of energy harvesters, sensors and other peripherals, and exciting new applications that are available to today's batteryless system designers is not well-supported by the WISP's RF-centric design. Application designers that need solar, kinetic, or thermal energy or can't depend on a close-range RFID reader are left to either hack the WISP [48] to suit their needs or create new hardware from scratch.

Limited Flexibility: The success of a batteryless sensing application often hinges on a variety of hardware adjustments that affect how energy is harvested and managed, how tasks are configured, and how data is gathered. Unfortunately, existing batteryless platforms are monolithic, tightly integrating energy harvesting, energy management, sensing, data processing and communication onto a single circuit board that is difficult to modify. Consequently, development is typically slow and developers may not consider promising design alternatives in fear of long delays.

Lacking Modern Amenities: Current platforms also lack recently-developed features, like hardware-assisted zero-power timekeeping [54] that allows devices to measure time across power failures. Timekeeping is incredibly important for sensing, security, data provenance, and data utility. Without a sense of time, batteryless active RFID cards could be brute force attacked for passwords, or endlessly tasked with authentication requests in a DoS attack. Data provenance and utility is usually tied to the time a data point was gathered. Federating energy storage for individual peripherals [53] not only simplifies software development and improves system availability, but improves energy harvesting efficiency. Timekeeping and federated energy storage are particularly important for batteryless applications that rely on ambient energy sources and can't rely on an RF reader to provide time.

Poor Usability or Community: Each of the previous factors contribute to the lack of usability in current hardware platforms, from the programming tools to the hardware inflexibility. Novice developers find it difficult to construct useful devices or verify that they work. The emergence of the maker movement has shown that hobbyists, as well as researchers and industry, are interested in building for the Internet-of-Things.

Without a usable platform, no community will emerge.

We have an answer to these deficiencies: Flicker, a flexible, modular hardware platform for energy harvesting, intermittently powered, batteryless sensing devices. With the help of common interconnects, Flicker developers assemble batteryless sensors from a set of interchangable modules — computational cores, energy harvesters, sensors, communication peripherals. We present twelve different modules, and describe how others can extend this set by developing their own compatible peripherals and harvesters. Flicker supports federated energy storage [53] and features a novel extension of the state of the art, allowing developers to programmatically set wakeup and task trigger points and change the relative priorities of individual sensors and other peripherals. Flicker also supports failure-resistant timing with an onboard capacitive timekeeper[54]. Flicker's software tools automatically detect new hardware configurations, and simplify software changes.

In this chapter, we first describe and evaluate the two foundational technologies of Flicker, zero power timekeeping (with TARDIS[103] and it's extension, CusTARD[54]) and Federated Energy[53]. Then we introduce the Flicker Hardware Platform with its novel extensions to Federated Energy, and its focus on rapid prototyping of batteryless Internet-of-Things devices. Finally, we close the chapter with a review of related work to hardware platforms.

4.1 Batteryless Timekeeping

One of the greatest challenges to batteryless computing is keeping track of time across power failures. When energy runs out, the microcontroller, volatile RAM, and all clocks are reset. This means that any previous timestamps have no meaning, since the local clock will start back at zero. Current embedded systems address the timekeeping issue in one of the following ways, none are sufficient:

Real Time Clock: A system can power a real-time clock (RTC). This is not practical on intermittently powered devices due to tight energy budgets and inconsistent power. Even a low-power RTC (*e.g.,* NXP PCF2123 [93] or Abracon AB08X5 [1] RTC chip) increases devices size, by using a battery, and violates the fundamental principles of batteryless sensing. This option increases the device's weight and cost, while reducing its usable lifetime—even rechargeable batteries wear out over time[100]. While in theory these devices could last decades on a small coin cell, in practice this is rarely the case. Constant recharging in deployment, hostile environmental conditions, poor charging circuitry design, and mismatched loads, have reduced the lifetime of batteries below expectations in many sensor deployments[124].

External Time: A system can keep time by accessing an external device (*e.g.,*an RFID tag reader) or by secure time synchronization [41, 122]. An external timekeeper can work well for some types of applications that are not time sensitive, or are deployed in easily accessible locations. However, relying an external timekeeper introduces security concerns and may either require significant infrastructure or severely limit range and mobility.

Approximation: A system can approximate or guess time by checkpointing state and integrating known constraints from the duty cycle or environment. However, this provides unreliable or inaccurate time and will not work for applications where constraints are fluid and the environment is unpredictable. Moreover, approximations are application dependent, and heavily rely on contextual information known before hand by the developer.

Remanence Based Timekeeping

None of the timekeeping techniques described can provide precise timekeeping that is applicable to general applications.

TARDIS[103] used the data remanence properties of SRAM to measure the time since a power loss on the UMich Moo. This method initializes a portion of SRAM (common in microcontrollers) to all ones on startup. After a power failure, and subsequent reboot after enough energy has been gathered, the TARDIS method counts the number of bits that have flipped in the pre-initialized portion of memory. Since SRAM decays at a consistent rate, elapsed time can be estimated by checking the ratio of flipped bits. This method can support measuring power failures up to a few seconds depending on capacitance.

We extended and generalized the TARDIS approach using dedicated capacitors and an ADC (or RTC) for precise timekeeping in[54]. By measuring the amount of voltage decay on a dedicated capacitor using an analog-to-digital-converter after a power failure, we can estimate time elapsed. We named this approach CusTARD, and call the collection of techniques Remanence based timekeepers. The execution model for using these timekeepers is shown in Figure 4.2. This hardware extension gives us greater control and measurement precision, and does not require large portions of volatile memory. This approach allows timestamps to have meaning across power failures, even when the microcontroller does not have enough energy to function. In contrast to the RTC method, keeping external time, or approximation techniques, remanence based timekeeping moves batteryless sensing towards a general solution to the problem of time in intermittently powered systems.

We detail this approach in a journal article currently in revision[54].

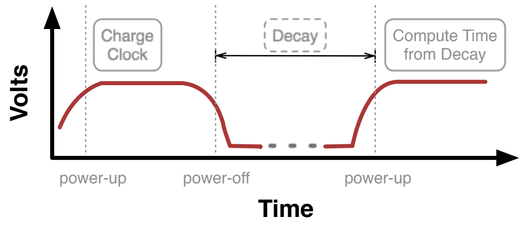


Figure 4.2: Remanence timekeepers estimate time from the amount of remanence decay. CusTARD measures how much the voltage on the capacitor has decayed, using an on-board ADC (compute time from decay). The lower the voltage, the longer the microcontroller has been off. Initially, the CusTARD capacitor is charged to a specific voltage (init timekeeper). This figure shows the voltage on the high side of the diode charging the MCU.

4.1.1 Timekeeping Initial Results

We found that using CusTARD dramatically improved our ability to keep time through power failures. Since the timekeeping accuracy and precision of the CusTARD techniques depended on capacitive decay, in our evaluation we examined the decay behavior of CusTARD and three factors that have major effects on this behavior, and therefore its timekeeping ability. We found that with a small capacitor (one micro-farad) we could keep accurate time for up to a minute when the sensor was off because of power failure. We also evaluated the overhead (in terms of software, and hardware) for implementing the CuSTARD approach.

Overhead: The overhead of using CusTARD comes from (1) energy required to charge the capacitor to the regulated supply voltage, (2) the time required to charge the capacitor to the supply voltage, and (3) the energy cost of using the ADC to read the voltage on the capacitor. Table 4.1 shows the cost in time and energy for charging the CusTARD capacitor. The charge time is the amount of time it takes to charge the CusTARD capacitor to 99% of the supply voltage after the diode, or $t = -\ln \frac{V-0.99V}{V} RC$, where V is the supply voltage minus the forward voltage drop of the diode, R is the charge resistor, and C is the capacitance. It should be noted that the charging of the capacitor is non-blocking; a single instruction sets the GPIO pin to high, starting the charging of the capacitor, immediately following this instruction, the processor can continue with regular function. The energy cost is determined by the amount of capacitance, the voltage on the capacitor, and the heat dissipation of the resistor: $E = CV^2$. With the MSP430FR5739 used in CusTARD experimentation, the maximum energy required for the ADC read with default ADC settings was 11.52 nJ.

We note that the TARDIS approach costs tens of microjoules and takes tens of milliseconds to operate. The calculated overheads of each approach show that CusTARD uses an order of magnitude less energy, and

Table 4.1: CusTARD energy and charge time costs.

Size	Energy Cost	Charge Time
0.01 µF	0.0013 µJ	0.0461 ms
0.1 µF	0.0128 µJ	0.4605 ms
1.0 µF	0.1276 µJ	4.6052 ms
10.0 µF	1.2755 µJ	46.0517 ms

takes less time than TARDIS for computation.

4.2 Federated Energy Storage

Energy is the greatest single limiting factor in the design and effective operation of mobile sensors and other untethered computing devices. A range of capacitor-based sensor devices [114, 132, 133, 126, 49, 45] that harvest energy, charge quickly, and can store only enough energy for short bursts of operation (a few seconds or even a few hundred milliseconds long). This new generation of tiny batteryless sensing devices will be cheaper, more durable, and more environmentally friendly than their battery powered predecessors. They will also have much tighter energy budgets and much more frequent power failures, two conditions that are poorly-supported by today's mobile hardware platforms, especially when it comes to storing and managing energy.

Traditional mobile devices store energy in a single common energy store (battery or capacitor) that is used to power all system components (*e.g.,* processors, sensors, radios and other peripherals) an approach that is simple to design and works well for devices with large batteries; however, when energy budgets are tight and failures frequent, each component's energy usage can significantly impact the availability of other components. For example, reading from a sensor may impact the device's ability to do computation. Power hungry operations, like transmitting a radio message, may cause the device to lose power entirely, becoming unavailable until the device can be recharged. Reasoning correctly about peripherals, consequent power consumption, and application priorities requires computational resources (for modeling and prediction) that transiently-powered devices cannot afford.

In this section, we propose a new federated approach to storing harvested energy that relaxes the coupling between a tiny, intermittently powered device's individual components (or subsystems). Our approach, called UFoP (United Federation of Peripherals), uses individual per-peripheral energy stores and low-power control circuitry to isolate and prioritize individual peripherals. Federating energy storage allows power-hungry

LIST OF LISTINGS 4.1: A conservative example program

```
when timer fires do { // Once every 1ms
  if can_sense_store_and_send? {
        while (!sample_buffer_full?) {
            collect_sample()
            sleep(1ms)
        }
        compute_and_store_mean()
        transmit_recent_means()
  }
    sleep()
}
```

operations to proceed without sacrificing the device's immediate ability to use other peripherals, gather new data, process incoming data, and respond to incoming stimuli. This approach also simplifies the task of programming energy-aware logic—effectively replacing complex modeling of analog circuit behaviors with simple binary decisions based on whether or not a peripheral is available.

Our initial implementation of the UFoP approach uses ultra-low-power comparators to control and prioritize the charging of individual energy stores, where the microcontroller gets the first priority. The main capacitor charges until it reaches 2.7 V, then the microcontroller turns on. When the input voltage reaches 3.1 V, the peripheral capacitors charge. In our experiments, we found that programs that use UFoP as the energy backbone had as much as 10% more computational availability, and as much as four times more radio availability than the centralized approach. Using UFoP, programs become more resilient, reducing low voltage events dramatically. Additionally, programs that use UFoP harvested more energy for all energy environments evaluated than programs using the traditional centralized energy storage approach.

While distributed energy storage has been employed in large-scale power systems [20], and hybrid storage systems have been used to improve the efficiency of battery charging [61, 101, 95], UFoP is, to the best of our knowledge, the first embedded system to employ federated energy storage, in a general way, to simplify management of individual system components.

4.2.1 Centralized Energy Storage

For half of a century, computing devices have used a centralized approach to power system components (processors, sensors, radios and other peripherals)—an approach that has reduced both device size and cost, and that, until now, has had no significant drawbacks. Whether power is supplied by a dedicated connection to wired infrastructure or by a battery that is regularly recharged, processors, memories, and peripherals all use a shared power supply and the nearly-universal assumption that power is unlimited and

LIST OF LISTINGS 4.2: A more optimistic program

```
when timer fires do { // Once every 1ms
  if can_sense? {
    collect_sample()
  }
  if sample_buffer_full? {
    compute_and_store_mean()
  }
  if can_send? && has_stored_means? {
    transmit_recent_means()
    }
    sleep()
}
```

stable. System designers have, at times, made efforts to reduce power consumption and extend battery life, but they rarely consider whether the program's next action may impair the device's ability to perform additional functions.

However, when designing applications for batteryless and other transiently-powered mobile devices, supplying power to all components using a single centralized capacitor or battery reduces the system's flexibility and complicates programmer decision-making. The following challenges must be carefully considered by a system designer before developing an application using centralized energy storage:

Capacitor tuning: Capacitor size is a critical factor that defines how an energy-harvesting batteryless device will operate. Smaller capacitors charge quickly, but may not be able to store enough energy for more power-hungry tasks. Larger capacitors can store more energy, supporting longer bursts of computation and more power-intensive operations. Larger capacitors also charge more slowly, incur longer power outages, and waste more energy (leakage). A system designer can maximize device availability and up-time by selecting a capacitor that is just large enough to support the operations that the application needs to perform. When an application performs both energy-efficient tasks (*i.e.,* sampling lightweight sensors or performing simple data processing) and energy-intensive tasks (*i.e.,* wireless data transmissions), a centralized energy store must be large enough to support all operations—both heavy- and light-weight operations.

Task coupling: A common storage capacitor also produces a tight coupling between program tasks and system peripherals. Sampling a sensor, for example, will consume energy and may either leave insufficient energy for subsequent tasks or may cause the supply voltage to drop low enough that the device loses power altogether, and subsequent tasks must wait until more energy becomes available.

In order to illustrate these challenges, let's consider two simple programs described in Listings 4.1 and 4.2. Both programs gather sensor readings until a buffer is full, compute the mean of the readings, and

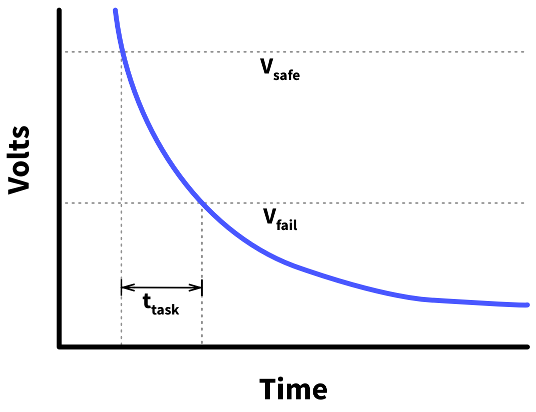


Figure 4.3: During task execution, capacitor discharge is sufficiently predictable to determine the voltage at which it is safe to begin executing it, in order to ensure that it will complete before a power failure.

wirelessly transmit a fixed number of recently computed means[1]. The first program (Listing 4.1) conservatively sleeps until it has harvested enough energy to an entire application cycle (collect, process, and transmit samples). The second (Listing 4.2) waits only until it has enough energy to complete the next task and then proceeds optimistically, assuming that enough energy will be harvested in the future to complete subsequent tasks.

Power failures are a constant concern for batteryless devices. When power failures do occur, computational state can be saved efficiently to nonvolatile memory (like FRAM) in case a power failure occurs between tasks or before a task completes. These checkpoints can be included explicitly by application developers or inserted automatically, using a system like Mementos [108]. Checkpoints allow some tasks to be resumed after a power failure; however, other tasks, like sampling a sensor or transmitting a radio packet, cannot be easily resumed due to timing and hardware constraints.

Consequently, the batteryless device that implements these example programs will need to determine, at runtime, when enough energy is stored on its capacitor to perform each desired task safely. A capacitor's stored energy can be estimated by measuring its voltage, and tasks can be safely run when that voltage exceeds a predetermined threshold (V_{safe}), which can be determined empirically or analytically based on the capacitance used, the time the task requires to complete (t_{task}), and the power needed while the task is executing. Figure 4.3 illustrates how this threshold is determined. If V_{safe} is chosen correctly, starting the task when the capacitor's voltage is higher than V_{safe} ensures that the task will always complete before the capacitor

[1]A small number of previous means are transmitted to provide some redundancy in case a packet is lost.

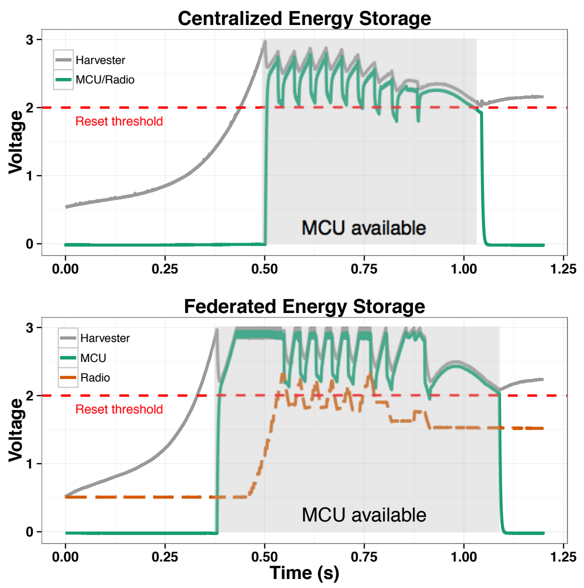


Figure 4.4: This figure shows a "sense-and-send" application using traditional centralized and our proposed federated energy storage approaches. Energy federation allows lightweight functionality (sensing and data processing in this example) to be available sooner.

discharges to V_{fail}—the point at which essential hardware components—MCU, memory, sensors, radios, or other peripherals—turn off or become unusable.

Some tasks within a single application may take longer or consume more power than others, so a separate threshold, V_{safe}, will be needed for each individual task. System designers may optimistically try to execute a task before the safe threshold is reached, in the hope that future harvested energy will be sufficient to finish the task successfully. When energy is abundant, this gamble may allow the device to produce results more quickly. When harvested energy is insufficient, sub-threshold task execution will result in wasted effort and may result in avoidable power failures.

Software federation of the energy supply as described above can lead to close coupling of components. The top plot in Figure 4.4 shows the capacitor and microcontroller voltages over time for a solar-powered batteryless sensor node that gathers sensor data and transmits it wirelessly to a base station (see Listing 4.1). In this scenario, a control circuit waits to turn on the microcontroller until the capacitor charges to a voltage (*i.e.,* around 2.7 V) sufficient to initialize the processor and accomplish some computation, the control circuit then turns off the microcontroller when the voltage drops below 2 V[2] (when processing becomes unreliable for most microcontrollers). The chosen capacitor is large enough (195 μF) to power both data collection and radio transmission on a single charge, and the application waits until the voltage charges up

[2]The control circuit's hysteresis is adjustable. The thresholds used are those we have empirically found to work well, in practice.

high enough to ensure that the transmission completes most of the time.

In this scenario, each radio transmission causes the MCU voltage to drop dangerously close to (and occasionally cross) the 2 V point where the MCU becomes unstable. The larger capacitor also charges slowly, and leaks, meaning that time and energy is lost.

4.2.2 Federating Energy

In light of the shortcomings of centralized energy storage, this work argues for a different approach which stores harvested energy in multiple independent small capacitors, one for core processing functionality, and another for each peripheral. We call this federated approach UFoP (United Federation of Peripherals).

By allowing the microcontroller, sensors, and radio to function independently, UFoP provides the following key benefits:

Useful work starts sooner: While some approaches have tried to mask volatility of the energy supply in software [19], the reality of capacitor based devices is that a smaller capacitor charges to an arbitrary voltage, faster than a larger one. A centralized approach may use software to mask volatility of the energy supply in order to simplify task management, but the energy store will still be unusable until it meets the necessary voltage for components to be turned on. By federating the energy storage, smaller capacitors charge more quickly, allowing lightweight tasks (some sensors and microcontroller operations) to be available while larger capacitors for radios and other power-hungry peripherals charge up. This can be seen in Figure 4.4 where the MCU becomes available hundreds of milliseconds before the centralized version.

Fewer power failures: Isolating each per-component capacitor prevents a power-hungry component from jeopardizing the whole system. For example, when the radio drains its dedicated capacitor's power to transmit messages, the device retains the ability (at least in the short term) to gather and process new data. UFoP prioritizes the charging of individual energy stores. In our current implementation, the microcontroller gets the first priority, while the priorities of sensors, radios, and other peripherals can be configured to suit individual applications. For example, the tight coupling present in the centralized approach shown in Figure 4.4 causes the supply voltage to dip below the reset threshold, endangering the device duty cycle. With UFoP, this problem occurs much less frequently.

Simpler application decisions: Each peripheral is available for use as soon as its capacitor is charged, and can be used independently of the charge state of other peripherals. When a peripheral is used and depletes its own energy, it becomes unavailable until it is recharged. This allows UFoP to "save up" energy for power-hungry tasks, like short bursts of radio communication, while allowing data collection and processing to continue. When using centralized storage, application decisions can be complicated—requiring designers to reason about the aggregate impact of multiple peripherals and tasks on a single capacitor. In contrast, a UFoP device can determine whether it can afford to use two peripherals (*e.g.,* a sensor and a radio), by simply checking their individual voltages.

Increased flexibility: UFoP devices provide more flexibility when combining peripherals with different energy requirements. For example, consider a sensor node that combines an MSP430 MCU as its computing core, a low-power sensor, and a more power-hungry radio. A larger capacitor will be needed in order to support the radio, which will charge much more slowly than a smaller capacitor that might be sufficient to support the other components for short bursts of operation. In a centralized energy architecture, radio transmissions would deplete the large common capacitor and may render the entire node unavailable while it recharges. In a UFoP device, the small capacitors dedicated to the core and sensor would charge more quickly, allowing the application to continue gathering and processing data, while waiting for the larger radio capacitor to charge. In both cases, data would be sent at roughly the same rate, but using UFoP the application would have more flexibility in deciding what to send.

Lower energy consumption: Not all system components require the same voltages to operate. UFoP allows individual component capacitors to be charged to the voltage required by that component, which often results in lower operating voltages and reduced per-component energy consumption.

Harvest more energy: When using UFoP, the energy harvester's voltage rises quickly (as the small microcontroller capacitor charges), but increases in harvester voltage is slow as power is diverted to charge peripherals. When UFoP's thresholds are set appropriately, the device spends more time in its most efficient voltage range and harvests more energy than the centralized equivalent.

4.2.2.1 UFoP Reference Design

Figure 4.5 shows a UFoP system that is integrated with two commonly used peripherals in sensing applications (*i.e.,* a sensor and a radio). The system consists of four main components: an energy harvester, charging controller, peripheral controller, and peripherals. The energy harvesting device harvests ambient energy and stores it in the first-stage capacitor. The charge controller is responsible for turning on and off the microcontroller and charging an array of peripheral capacitors. The peripheral controller turns on and off peripherals (sensor and radio), which are only available when their capacitors are charged.

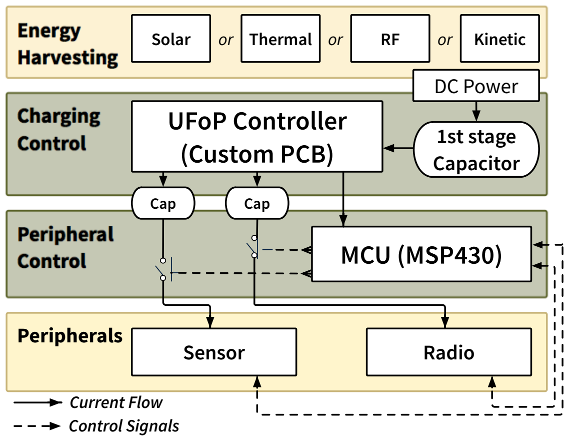


Figure 4.5: Overview of UFoP when integrated with a set of commonly used peripherals; a sensor, and a radio. A UFoP system is made up of four components: an energy harvester, charging controller, peripheral controller, and peripherals. The charge controller manages the charging of an array of capacitors (drawing from the first-stage capacitor) as well as turning on and off the MCU. The peripheral controller (an MSP430FR5739 in the current prototype) gates power to the peripherals, allowing peripherals to be completely off when not in use. Peripherals and the MCU communicate independently of the charging controller.

Energy Harvesting: The UFoP system is powered by ambient sources. The energy harvester converts free energy from the environment (*e.g.,* solar, thermal, radio frequency (RF), and kinetic energy) into electrical energy (DC), which the harvester supplies to the rest of the system. From the harvester, the current flows to charge the first-stage capacitor that powers the microcontroller.

Charging Control: UFoP uses low-power control circuitry as charge controller to control and prioritize the charging of an array of capacitors as well as turning on and off the microcontroller. UFoP is designed with the microcontroller as a non-negotiable first power priority. A sensor without a microcontroller could not process data, and a radio without a microcontroller would have no signal to transmit. From an implementation perspective, this is ideal since the microcontroller can then be used to control the power flow to the peripherals. As sensors and radios can provide functionality regardless of the operation of other peripherals, subsequent priority values can be assigned based on system requirements. In our reference design, the first-stage capacitor charges until it reaches 2.7 V, then the microcontroller turns on. When the input voltage reaches 3.1 V, the current starts to flow into the peripheral capacitors.

Peripheral Control: The peripheral controller in the UFoP system can be any ultra-low-power microcontroller, like the FRAM based MSP430 processors. In our reference design, we use the MSP430FR5739 as peripheral controller; which only draws 81.4 μA/MHz in active mode. When the first-stage capacitor capacitor charges up to 2.7 V, the microcontroller turns on. The microcontroller gates power to the peripherals, allowing peripherals to be completely off when not in use. This control communication is independent from the charging control scheme. The switches in the peripheral controller are designed to open (disconnect) when the microcontroller loses its power and turns off, *i.e.,* when the main capacitor's voltage falls below 1.8 V. This design satisfies the MSP430FR5739 supply voltage requirements, *i.e.,* 1.8 V to 3.6 V. We built a thin software layer to manage the ADC polling, timers, and interrupt wake ups, for the MSP430 line of microcontrollers. This layer can easily be ported to other platforms, as the components and software practices are common among embedded systems.

Peripherals: Two of the most commonly used peripherals in sensing applications are sensor and radio. The type of the peripherals and the tasks they perform determine the size of the capacitors used. When an application uses a lightweight sensor and a radio, the size of the capacitor for the sensor should be a lot smaller than that for the radio. If the application requires intensive data transmission, the radio's capacitor size must be large enough to support the tasks. In the UFoP system, the peripherals are only available when their dedicated capacitors are charged.

4.2.2.2 Application Development Simplification

Federating energy storage changes how sensing applications are built. From the hardware point of view, sizing several capacitors appropriately to peripherals is much easier than sizing one capacitor to multiple peripherals and a microcontroller that could have different supply voltage requirements. For example, the MSP430 microcontroller works within the 1.8 – 3.6 V range, the CC2500 radio (a very common 2.4 GHz radio) works from 2.0 V to 3.9 V, and a humidity sensor used in our greenhouse monitoring application works from 3 V to 5 V. This combination makes it difficult for application developers to size a single capacitor and determine duty cycle. In this example, the radio and microcontroller could potentially never come online while waiting for the voltage to be sufficient for the humidity sensor to function. UFoP allows application developers to simplify this by sizing and dedicating each capacitor at a specific voltage to each peripheral.

From the software development point of view, UFoP allows a duty cycle when energy is scarce, and when it is abundant. That means UFoP lets the duty cycle scale up or down without hurting the average duty cycle performance. For example, in a classic "sense-and-send" application, the device can use a more powerful sensor when there is an abundance of energy, but performs only computations if there is very little energy. With UFoP, programmers have a more deterministic view of energy and task scheduling, allowing them to make better informed decisions during application development. In addition, UFoP simplifies applications by eliminating the need for complicated algorithms to predict when the peripherals become available.

4.2.3 Implementation

We have implemented a UFoP reference design on a custom printed circuit board. The prototype employs a variety of different hardware components. Two nano-power comparators are used per peripheral, that control the charging, and discharging of the peripheral capacitor. The current prototype supports two peripherals. An ICL7665 or MIC841 voltage monitor is used to monitor the first-stage capacitor, this monitor has a built in reference, and resistor defined hysteresis. The settable hysteresis allows a broad operating range for the microcontroller. Each of the comparators has a resistor divider defined trip point; the trip points and hysteresis are set in such a way that the microcontroller will always be on if the peripherals are charging.

Because UFoP is a hardware addition, some note must be taken of its size and cost. The current UFoP prototype measures 37.0 mm by 15.2 mm, with a low profile. The total cost of the prototype bill of materials, including all components, and PCB from a batch PCB supplier like OSH Park, amounts to less than \$20 per device, further development could easily lower this cost. If the current prototype was produced at scales of a 1000, the price drops to less than \$2 per device. Most of this cost would disappear in a custom silicon solution, which would also reduce the size and component count for a deployed sensor.

In addition to building the UFoP prototype, multiple systems were built or used in the evaluation. We used the energy environment emulator Ekho[52] to record and replay solar energy harvesting environments, as well as RF energy environments. Solar environments were generated by a programmatically controlled headlight focused on a solar panel. RF environments were generated from the harvester of a UMich Moo[136], which harvested RFID energy from the UHF band created by an Impinj Speedway RFID Reader. IV-surfaces were created by moving the reader back and forth across the front of the UMich Moo. The recorded IV-surfaces were later replayed in the lab using Ekho; which provided a realistic energy harvesting environment to test UFoP with. We plan to make all IV-surfaces we recorded freely available online at publication time. We also used an NI USB-6356[56] and Measurement Computing USB-201[25] data acquisition device (DAQ) for voltage and current measurements, as well as event counting. For all applications, we used MSP430FR5739 Launchpads as the main processing device.

We translated the programs described in Listings 4.1 and 4.2 into embedded C, running on the MSP430FR5739. Each program has two variants, one that is meant to run with UFoP functioning as the energy backbone, and one that runs with the traditional centralized energy approach. The federated and centralized variants are intentionally similar for both programs; the federated versions differ in that they dedicate an ADC per peripheral to monitor the energy storage levels of the radio and accelerometer supply voltages, in addition to the MCU energy storage levels. Both of these programs attempt to federate energy storage, one using UFoP, and one in software. We used these programs to evaluate the effectiveness of the federated energy approach in terms of availability, resiliency, and energy harvesting efficiency. We also developed federated and centralized versions of a greenhouse monitoring application, which is described in Section 4.2.5. These are similar in function to the above, but use a different set of peripherals. A basestation program was also created to listen for and log sensor readings during the greenhouse monitoring deployment.

All hardware designs, I–V surfaces, and software will be made freely available online at publication time.

4.2.4 Evaluation

In this section, we evaluate the performance of sensing applications that use our UFoP reference design as an energy backbone. Specifically, we examine how federated and centralized variants of sense-and-send behave in solar and RF energy environments. We compare these two approaches and measure them in terms of availability, resiliency, and energy harvesting performance.

In our experiments, we found that programs that use UFoP as the energy backbone had as much as 10% more MCU availability, and as much as four times more radio availability than the centralized competitor. Using UFoP, programs become more resilient, reducing failures by 4.5*x* in some cases. Additionally, programs that use UFoP harvested 0.7-10.2% more energy, for all energy environments evaluated, than programs using a centralized energy storage approach; meaning UFoP adds zero energy overhead.

4.2.4.1 Methodology

To evaluate UFoP, we consider multiple programs, in a variety of energy environments, with the same hardware and peripherals, but interchanging the centralized and federated approach to energy storage. We use the following experimental setup to evaluate how UFoP contributes to the performance, in terms of availability, resilience, and energy efficiency of a tiny, capacitor-powered sensing system.

Programs: We use the sense-and-send program variants described in Section 5.1 to provide points of comparison between UFoP and the centralized approach. The program described in Listing 4.2 we refer to as "optimistic" sense-and-send, the program described in Listing 4.1 we refer to as "conservative" sense-and-send. The program using UFoP monitors capacitor voltages, and gates energy flow to peripherals as needed. The

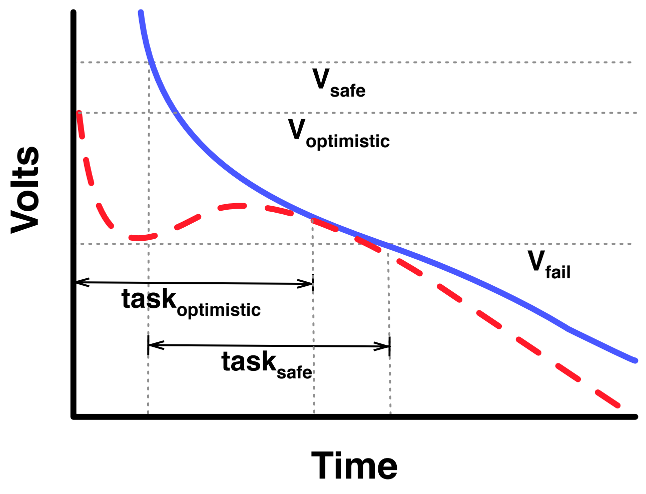


Figure 4.6: For our evaluation, two sets of thresholds were determined for each program variant. The first threshold, V_{safe} (the blue discharge curve), is set such that if no new energy is harvested, tasks started at this threshold are guaranteed to complete. The second threshold, $V_{optimistic}$ (represented by the red discharge curve), optimistically assumes new energy will be harvested to replenish the task capacitor during task execution.

centralized program attempts to federate energy storage in software, allocating from its single energy store when it becomes available at the required voltage.

Thresholds: Each program has certain voltage thresholds (V_{safe}) where the radio, sensor, and MCU turns on, as described previously in Figure 4.3. This threshold is the voltage on the capacitor that signifies there is enough energy to complete a task or set of tasks, such as sending a data packet over the radio. These thresholds are set in software for the radio and sensor, and hardware for the MCU. Thresholds are set differently for centralized, since the single capacitor must store enough energy to accomplish all tasks. We have two sets of voltage thresholds for each program, as shown in Figure 4.6. The first threshold, termed V_{safe} is the level that guarantees task completion. Setting the threshold above V_{safe} means a program will not get as many tasks done (but will never fail), while setting the threshold below V_{safe} will mean tasks are not guaranteed to complete. The second set of voltage thresholds is termed $V_{optimistic}$; the assumption is that new energy will be harvested to replace energy being used *during the task*, therefore these thresholds are set lower. Using the $V_{optimistic}$ set of thresholds does not guarantee task completion or zero power failures. These thresholds were gathered by manually executing programs with high and low thresholds over a solar I–V surface, effectively binary searching through all possible thresholds. V_{safe} and $V_{optimistic}$ thresholds were only gathered for the “optimistic” sense-and-send program. $V_{optimistic}$ thresholds were gathered for the “conservative” sense-and-send program.

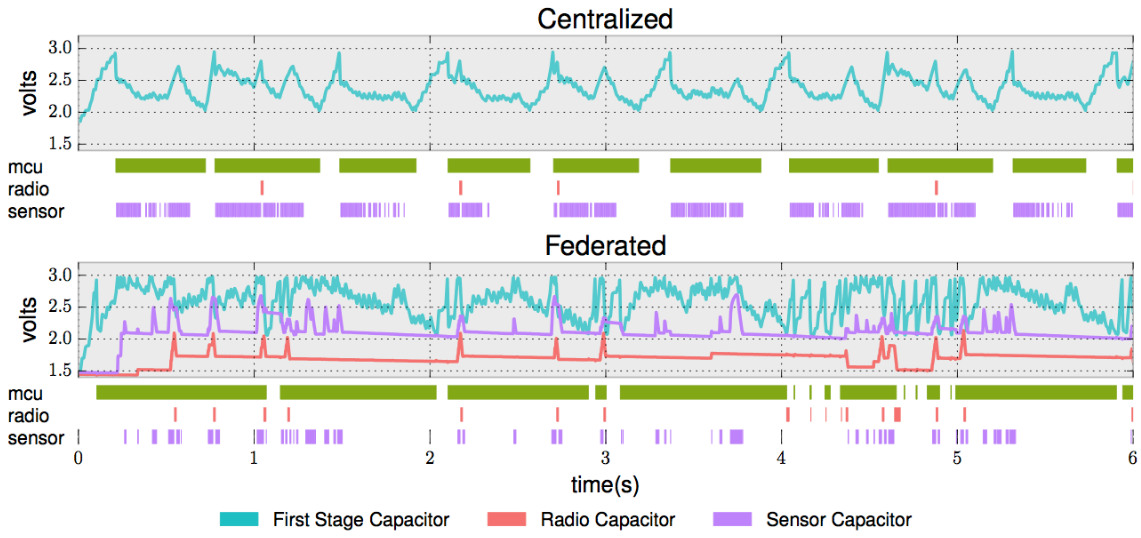


Figure 4.7: This figure compares the availability of the MCU, sensor, and radio when an MSP430FR5739 running the program described in Listing 4.2 executes across the I–V surface recorded from the RFID reader. The program was run multiple times using either the UFoP reference design or the centralized reference design, and the V_{safe} thresholds. Applications that use UFoP have significantly more MCU on-time, allowing more valuable computation, as well as significantly more sends over the radio. Because the radio can work with a lower supply voltage than the MCU, UFoP allows for more transmissions since the energy store is decoupled from the MCU. Sensor readings, while not as frequent as with centralized, are dispersed more evenly in time.

Test Devices: Each of the applications were run on Texas Instruments MSP430FR5739 processors. These devices have low sleep currents (*i.e.,* 5.9 µA), multiple ADCs, and FRAM memory for checkpointing and data storage.

Peripherals: Each test device manages its own set of peripherals. Peripherals used are an MMA7361 triple-axis accelerometer, and a 315 MHz RF Transmitter (CDT-88). These peripheral types were chosen because they are ubiquitous in sensing applications.

Voltage Monitor Reference Designs: To compare the federated and centralized approach, we designed a supply voltage monitor, based on the ICL7665 and MIC841 (also used by our UFoP reference design) to act as the energy storage backbone for all centralized program executions. The ICL7665 or MIC841 charges a large capacitor bank, and turns on and off the MCU with hysteresis. The centralized supply voltage monitor and the UFoP reference design have the same amount of capacitance (energy storage) and many of the same components, however, the UFoP reference design federates its energy storage in hardware, while the centralized reference design federates energy sotrage in software. In the evaluation, we refer to the centralized voltage monitor device as the "centralized reference design."

		Availability					
		UFoP			**Centralized**		
Program	**I–V Surface**	*MCU (%)*	*Radio (%)*	*Accel (%)*	*MCU (%)*	*Radio (%)*	*Accel (%)*
Optimistic	Solar	57.08	7.61	18.22	46.97	6.36	17.67
	RF High	81.85	4.55	10.45	76.89	1.18	25.29
	RF Low	86.58	1.23	9.43	75.69	0.64	20.93
Cautious	Solar	54.75	8.14	13.67	46.64	8.10	13.40
	RF High	68.11	7.99	9.59	80.69	3.77	12.00
	RF Low	74.23	4.67	5.70	74.97	2.66	8.09

Table 4.2: This table shows the percentage of time over the entire I–V surface that the peripherals and MCU were available, for both UFoP and centralized. For these results, the program described in Listing 4.2 (optimistic) and the program described in Listing 4.1 (conservative) were used, with the V_{safe} thresholds (as in, no radio transmission or power failures). For all three surfaces, using UFoP increases the availability of both the MCU (for computation) and the radio (for communication and data sending). UFoP does especially well on RF surfaces, where energy is scarce, since it does not have to wait to charge a much larger capacitor before it begins computation.

I–V surfaces: To control the energy environment, we use an Ekho[52] device to record and emulate I–V surfaces. Ekho provides a repeatable energy environment which replaces the energy harvester as input to the energy storage approach. Without Ekho, it is very difficult to control for the energy environment in experiments. We recorded three eight-second I–V surfaces for our evaluation. The first surface was recorded from a solar panel harvesting energy from a car headlamp that turns on and off four times, creating a sinusoidal solar surface. This means the device under test goes through the entire life cycle; charge, deplete, recharge. The second and third I–V surfaces were recorded from the RF energy harvester on the Umich Moo, while it harvested from an Impinj Speedway RFID reader. The reader antenna was waved across the Moo at two distances to produce two surfaces, referred to as "RF High Energy" and "RF Low Energy" in the rest of the evaluation.

Taking these programs, thresholds, test devices, peripherals, supply voltage monitors, and I–V surfaces, we can assemble an experimental setup that will allow us to make fair comparisons between federated and centralized energy storage. By running each program variant (federated or centralized) in each recorded energy environment, we can attempt to answer these questions about UFoP:

- Does federating energy storage provide more sensing and computational availability? (4.2.4.2)
- Does federating energy storage make applications more resilient? (4.2.4.3)
- What is the effect of federating energy on energy harvesting efficiency as compared to the centralized approach? (4.2.4.4)

- What is the overhead of the federated energy approach? (4.2.4.5)

4.2.4.2 Availability

The percentage of availability of the MCU for computation, and peripherals for sensing or sending over an entire duty cycle, is a critical metric of evaluating performance of tiny energy harvesting systems. In this section, we evaluate the availability of both our test programs, when running on our UFoP reference design and the centralized reference design. We execute each of these programs, with both energy storage reference designs, ten times each, on all three of our recorded I–V surfaces. All of the optimistic sense-and-send programs use the V_{safe} thresholds, while the conservative sense-and-send programs use slightly "optimistic" thresholds. Using the Measurement Computing USB-201[25] data acquisition device (DAQ), we recorded the voltage levels of all capacitors, the supply voltage of the MSP430FR5739 (the MCU), and the on and off times of the radio and accelerometer peripherals.

One set of program executions is shown in Figure 4.7. This figure shows the optimistic sense-and-send program executing across a surface generated by an RFID reader swiping over a UMich Moo. The three activity bars below each plot show when the MCU, the radio, and the accelerometer were in use. For this RF surface, UFoP provides more MCU on-time, and more radio transmissions. Because UFoP allows the application to use peripherals that do not have to exist at the MCU supply voltage, the UFoP program makes use of the lower voltage threshold of the radio to get extra work done in a low energy environment. Additionally, UFoP charges its capacitors faster, meaning that the MCU turns on sooner than the centralized version, this is shown in the bottom portion of Figure 4.7.

The results of all availability experiments are shown in Table 4.2. The percentage of time each component was being used is listed for both the centralized and federated approaches. For all cases, using UFoP shows improvement in availability. The most dramatic increase comes when using UFoP with low energy environments and peripherals that don't match the MCU supply voltage.

4.2.4.3 Resiliency

In this section, we evaluate the resiliency of our programs when using the UFoP reference design and the centralized reference design. Resiliency is the measure of how tolerant an application is to voltage threshold miscalculation, task incompletion, and power failures (of the MCU or peripherals). While most application programmers try to get the equivalent of the V_{safe} threshold described in Section 4.2.4.1 to ensure no failures, it is very easy to miscalculate the required energy budget for a specific task, especially when it

	Resiliency					
	UFoP			Centralized		
Program	*-Threshold*	*Low Voltage*	*Tx Fails*	*-Threshold*	*Low Voltage*	*Tx Fails*
Optimistic	-151 mV	6.2	3.5%	-130 mV	28.0	9.8%
Conservative	-93 mV	5.1	5.2%	-115 mV	34.6	16.3%

Table 4.3: This table shows the effect of over-estimating the harvestable energy. In the table, *-Threshold* is the voltage below the the V_{safe} threshold that was set when running programs for resiliency experiments. This threshold is used to determine when to turn on peripherals for the respective program listed on the left. These results are from execution over the solar I–V surface. The number of times the voltage on the microcontroller went below the minimum supply voltage and the percentage of radio transmission failures (because of MCU reset or peripheral reset) are shown. Poorly choosing voltage thresholds does not have as severe an effect when using UFoP, as with the traditional centralized approach.

comes from a single supply. Being overly optimistic about potential energy to be harvested can result in failed radio transmission, corrupted memory, and low voltage events. A low voltage event, where the MCU voltage drops below the minimum supply voltage, does not necessarily mean the microcontroller is reset, but once it happens, the MCU begins to draw more current, memory usually becomes unwritable, and at worst the Supply Voltage Supervisor will trigger a brown out. Therefore it is a state best to avoid if possible.

To evaluate resiliency we executed both of our programs, using either of our energy storage reference designs, ten times each, on our recorded solar I–V surface. We lowered the turn-on voltage of the radio from the V_{safe} threshold by a percentage, to see what effect this optimism over energy harvesting would have. Since we did not have V_{safe} thresholds for the conservative sense-and-send program, we chose thresholds that ensured zero radio transmission failures or resets over the solar surface. Since these thresholds were not matched like the V_{safe} thresholds, the conservative results are illustrative of the effect of over estimating your energy harvesting. Using the DAQ, we recorded the voltage levels of all capacitors, the supply voltage of the MSP430FR5739 (the MCU), and the on and off times of the radio, and accelerometer peripherals. Using this data, we gathered for each execution, the number of times a low voltage event occurred and the percentage of radio transmission that failed.

Table 4.3 shows the results of this experiment. For the optimistic sense-and-send program, the lowered thresholds result in a dramatic increase in low voltage events, as well as a nearly 10% increase in transmission failures for centralized programs. The optimistic sense-and-send program that ran on the UFoP reference design had a much smaller failure rate. Power failures and low voltage events are inevitable for capacitor based sensing. UFoP reduces the number of low voltage events and failures by prioritizing the MCU and separating the peripherals energy storage. Using UFoP, the consequences of miscalculation of the energy

	Energy Harvesting Comparison			
	UFoP		**Centralized**	
I–V surface	*mean* (mJ)	*stddev* (mJ)	*mean* (mJ)	*stddev* (mJ)
Solar	11.49	0.05	11.41	0.06
RF High Energy	11.00	0.10	9.98	0.11
RF Low Energy	9.15	0.12	8.66	0.10

Table 4.4: This table shows the amount of energy harvested by the optimistic sense-and-send program on each of the I–V surfaces, for centralized and UFoP energy storage. The peripheral turn-on voltage was set to the V_{safe} threshold, such that no transmissions would fail, and no resets would occur. When using UFoP, the application harvests more energy for all I–V surfaces tested.

harvesting potential of a future deployment environment becomes much less catastrophic.

4.2.4.4 Energy Harvesting Efficiency

Energy harvesters such as solar panels, piezoelectric ceramics, and thermal generators, do not supply a stable voltage to a sensor. The voltage on the harvester changes in response to the current draw of the sensor or vice-versa. This relationship between harvesting current and supply voltage can be described by an I–V curve. Energy harvesting over time can be described by a sequence of I–V curves; an I–V surface. Every I–V curve has a maximum power point (MPP) where the most energy can be harvested. Many systems attempt to track this point to increase energy harvesting efficiency. UFoP does not try to track the MPP, but because UFoP charges faster and keeps a more stable supply voltage, for some I–V surfaces, UFoP may harvest more energy by being closer to the MPP. This set of experiments seeks to compare the energy harvesting efficiency of the centralized and federated approach to energy storage, by observing the path that each approach traces across the same I–V surface. By keeping the same sensor combination, energy harvesting environment (I–V surface), and program, we can see the effect of the energy storage technique on efficiency.

The results of this experiment are shown in Table 4.4. For the energy harvesters used, the programs using the UFoP reference design generally harvested more energy than the centralized equivalent. UFoP will not always cause more energy to be harvested, however, if the voltage at the MPP of the particular I–V surface is close to the hardware set threshold voltage of UFoP, the stability of UFoP should provide more energy.

4.2.4.5 Overhead

Switching to a federated energy storage approach does come with overhead. This overhead comes from three places: 1) the addition of voltage monitoring hardware, which slightly increases the size, cost, and energy requirements, 2) the energy cost from polling voltage levels with the built-in ADC, and 3) software rewriting. We have not attempted to quantify the cost of software rewriting.

The biggest potential cost is the energy overhead; as UFoP is meant for energy harvesting systems, any energy spent on monitoring must be kept low. The active components that make up our reference design have a typical quiescent current draw of 2.7 µA. As a comparison, the centralized reference design has a quiescent draw of 1.5 µA. Despite the increase in overhead, the energy harvesting gains shown in Table 4.4 should offset the losses, and often give a net surplus of energy. Additionally, the larger capacitor used for the centralized approach has a larger leakage than the collection of smaller capacitors used in UFoP, causing some of the energy gains seen from using UFoP.

Most traditional centralized applications have some form of supply voltage monitoring through a dedicated ADC, interrupts on an input pin, or special hardware. UFoP has this same overhead, but multiplied by the number of peripheral capacitor voltages it has to monitor. The centralized reference design, over a one second period, expends 13.6 µJ polling, while the UFoP reference design expends 23.3 µJ over the same period. This extra energy cost can be reduced by polling fewer times, or using a low power or faster ADC. To reduce the polling overhead completely, an interrupt driven method can be used. With this approach, UFoP can trigger a wakeup pin on the microcontroller when a peripheral capacitor has reached a logic level threshold. This form of voltage monitoring requires no extra energy beyond the sleep current of the microcontroller. If greater accuracy is required, the program can do an ADC check immediately after the interrupt wakes the microcontroller.

4.2.5 Deployment

To evaluate UFoP in a real application scenario, we deployed a UFoP enabled greenhouse monitoring program for eighteen hours over two days, in a local greenhouse in late summer. We also deployed a centralized version on the same bed, as a comparison. Greenhouses waste a significant amount of water by overwatering plants. This happens because in large greenhouses, managers do not know the status of individual plant beds, and overwater to ensure plants do not die. This waste is significant for economic and sustainability reasons, as water is a finite and costly resource especially at large scales. Current commercial plant monitoring systems are little more than weather stations, these have dedicated power supplies, are too large or too expensive to be deployed densely, and can't move with the plants they monitor. Because of this, sensor data is usually very coarse, not localized, and often wrong (in the case of plants that were moved from bay to bay). Dense deployment of tiny, unobtrusive, energy harvesting, sensors has been suggested as one way to monitor large volumes of plants in a greenhouse. Equipping sensors with leaf wetness, temperature, and humidity sensors

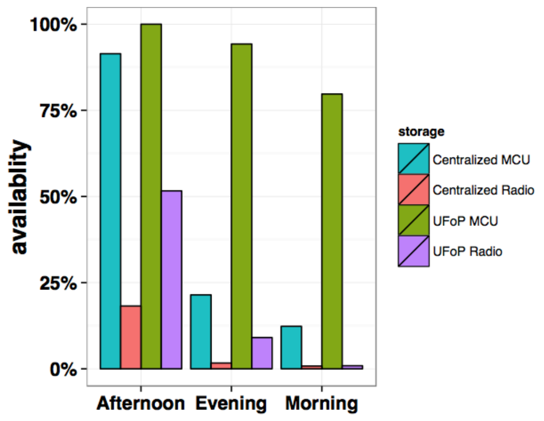


Figure 4.8: This figure shows the availability of the radio, and microcontroller, for both energy storage solutions, for the deployment. Three different time periods are shown; the afternoon, when the sun was brightest, the evening, when the sensors energy harvesting began to decrease dramatically, and the morning of the next day, when the energy harvesting begin to increase. Even though the centralized system and the UFoP system had the same amount of energy storage, the same harvester, and the same duty cycle, the UFoP sensor had more radio, and computational availability.

would provide all the information necessary for managers to make local decisions on water volume and plant health.

We developed an initial implementation of this greenhouse monitoring application with UFoP as an energy backbone. Two sensors derived from those discussed in Section 4.2.4.1 were built; one using our UFoP reference design, and one using the centralized reference design. Each sensor had two peripherals: a CC1101 transceiver for communication, and a resistive load that emulated a Decagon Devices LWS leaf wetness sensor, a standard sensor used in plant studies. The MSP430FR5739 was used as computational platform. Each sensor used a small solar panel (of the same model) for energy harvesting. To allow for data comparisons between the sensors, the panels were located as close as possible. Both sensors had the same amount of total energy storage in the form of SMD capacitors, and the same duty cycle. Both sensors periodically wakeup from a low power sleep mode, check the energy level(s) of the supply capacitor(s), and if high enough, sense and send a leaf wetness reading. A basestation was positioned inside the greenhouse to receive, and log, all sensor readings.

4.2.5.1 Choosing Capacitor Sizes

Choosing capacitors for both sensors required consideration of the duty cycle. Capacitors had to be large enough to support the peripherals, but not too large that they never charged to a high enough voltage. Before deployment, we profiled the distinct stages of the greenhouse monitoring duty cycle in terms of energy consumption, using an oscilloscope and current sensor. Each of these stages we matched to an adequately sized capacitor. The greenhouse monitoring program has three stages: "sense", "send", and "sleep". For each stage, we determined the minimum size of the capacitor, by looking at how much energy was used over time, for what voltage thresholds (we looked up voltage thresholds in device and peripheral datasheets). For example, the CC1101 transceiver when used during the "send" stage required 40ms at an average draw of 3mA to send a message, all with a supply voltage above 1.9V. A capacitor sized at 100 µF and charged to 3.2V stored enough energy at a high enough voltage to power the stage to completion.

The centralized version was equipped with the same amount of total energy storage as the UFoP enabled system, to keep the comparisons fair. Centralized thresholds were calculated in a similar fashion to UFoP. By summing the total energy required for all three stages, at the highest minimum voltage of all the components, the threshold voltage can be determined for the centralized variant.

While these capacitor size and voltage threshold calculations were done manually for this deployment, it is not hard to see how an automated system could size UFoP capacitors using simple peak detection techniques, developer or datasheet specified information about peripheral voltages, and energy harvesting information generated or gathered by an Ekho device.

4.2.5.2 Deployment Results and Discussion

An ARM microcontroller and light sensor was deployed with the sensors to unobtrusively record availability of the MCU, radio, and sensor over time. The amount of light on the solar panels was also recorded. Data was gathered for each sensor from 4pm, to 10am the following day. The UFoP equipped sensor outperformed the sensor with the centralized reference design in terms of MCU availability and radio availability as Figure 4.8 shows. The UFoP enabled sensor was able to harvest significantly more energy than the centralized version, especially during times when energy was scarce (evening and morning). In the morning, from 7-10am, the UFoP equipped sensors microcontroller was on for 79% of the time, while the centralized sensor's microcontroller was on for only 12% of the time. The amount of solar energy that was available to harvest, was not enough to charge the much larger capacitor on the centralized version, meaning that data was lost. In the evening, as the sun began to drop, the UFoP equipped system harvested enough energy for the radio to be broadcasting 9.6% of the time, while the centralized version was only able to broadcast 1.7% of the time. UFoP dramatically extended the amount of time the sensor was available compared to using a centralized energy approach.

This first implementation could be improved; in full sun both sensors were able to broadcast readings continuously, meaning that the solar panel used was too large. Greenhouse managers only need leaf wetness reports a few times an hour. By decreasing the size of the solar panel, sensors can be more densely deployed. Computational time was underutilized as well. UFoP allowed the sensor's MCU to be available even when there was very low sun, however, this computational time was not used. Future programs will use this time to calculate average leaf wetness readings, and calculate local statistics on plant status, freeing up computation on the basestation.

4.2.6 Discussion

In our experiments, UFoP was able to improve the availability, resiliency, and energy harvesting efficiency of tiny sensor devices with capacitor-based energy storage. Based on our experimental results, we believe that UFoP is a step forward for perpetual sensing, potentially making long-term deployments for mobile and other untethered computing devices possible. However, ambient energy is still scarce in deployment and energy storage capacity is terminally limited. Therefore, even though UFoP is able to relax the short-term coupling between peripherals, power management is always an important issue to consider. In this section, we discuss software approaches to federating energy, UFoP's design limitations, design alternatives, and applications.

Software Energy Federation: An alternative approach to using UFoP is to attempt to federate energy in software, by balancing the peripherals voltage and energy requirements with the volatile energy supply. This approach is used by all centralized programs in Section 4.2.5 and Section 5.4. Federating (or virtualizing, as in Virtual Battery [19]) energy in software has the advantage of being reconfigurable; the duty cycle (and energy partitions) can be changed dynamically. Additionally, no new hardware is required. However, the cost of software complexity in adding a virtual layer must be considered, especially in resource constrained systems that harvests energy. Additionally, a software approach suffers from all the shortcomings (task coupling, slow charging, reduced MCU availability,) associated with a single energy store. Hardware federation, for the same energy supply, will always provide more availability, and resiliency, than a software federated approach.

Limitations: The current UFoP reference design has some limitations. Peripherals are susceptible to starvation since the peripheral capacitors start charging when the input voltage in the main capacitor reaches 3.1 V. This can happen when an application requires the microcontroller to run too much computation in a very low energy environment and thus the peripheral capacitors never have the chance to charge. The centralized system has the same starvation problem. Another cause of peripheral starvation is if the microcontroller has a higher active current draw than the one we currently use (81.4 μA). To overcome this limitation, an application developer can program, through software, the microcontroller to sleep for a brief period immediately after turning on.

The current UFoP reference design trades off speed for low quiescent energy consumption in the choice of its active components. Some of the comparators switch very slowly, meaning that high current peripherals may be able to draw too much from the first stage capacitor if no new energy is being harvested. This can be solved by limiting capacitor charging with a resistor or op-amp, or trading off low quiescent current for a faster switching peripheral gate comparator.

Another limitation of the current prototype is we cannot use the peripherals in the order we need if we change the behavior of an application during runtime. This issue can be handled with dynamic priority, allowing capacitors to be charged and used by different peripherals according to the application. We leave this for future work.

Applications: UFoP enables low power sensing applications that use a variety of peripherals, with different energy needs and voltage requirements. UFoP is most useful for perpetual, energy harvesting systems that aggregate multiple types of sensor data. We envisage UFoP being used in applications ranging from greenhouse monitoring, low power wearable devices (humans and animals), and any computational RFID application including but not limited to, inventory management, building monitoring, activity monitoring, and infrastructure monitoring.

4.2.7 UFoP Conclusions

This section presents UFoP, the first system for capacitor-based sensors that employs a federated approach to the storage and management of harvested energy. UFoP stores harvested energy in multiple independent small capacitors, one dedicated to each peripheral. It employs a charge controller that charges the capacitors while maintaining the supply voltage of the microcontroller. With UFoP, power-hungry tasks from a radio or a heavyweight sensor will not cause low voltage events that can potentially reset the microcontroller. In our experiments, we found that programs that use UFoP as the energy backbone had as much as 10% more computational availability, and as much as four times more radio availability than the centralized approach. Using UFoP, programs become dramatically more resilient, reducing low voltage events and radio transmission failure. Additionally, programs that use UFoP harvested more energy for all energy environments evaluated than programs using the traditional centralized energy storage approach; meaning that UFoP functions with zero overhead in many cases.

4.3 Flicker Platform

The future of sensing likely depends on tiny, batteryless, energy harvesting devices because of the poor economics, sustainability, and scaling of batteries. Today, building and deploying untethered, batteryless sensors is challenging because 1) design-time decisions make prototyping slow and expensive, 2) flexibility of tasks and hardware is lacking, 3) few people have the expertise to build and deploy these sensors, and 4) few readily available hardware options exist, short of building custom hardware.

Modules from Sparkfun[3], or Arduino[4] shields can't just be wired to a batteryless sensor, deployed, and then expected to work. Arduino developers cannot replace the battery with a postage stamp sized solar panel and expect it to function just the same. The architectural, operating systems, and language support for intermittent sensors is not widely available or explored, except in custom configurations of hardware and software. The ultra low operating power requirements, variable energy environments, and difficult to use tools, discourage most developers from ever trying to work with batteryless sensors unless they can afford to design and assemble their own hardware from scratch.

4.3.1 Limitations of Static Federated Energy

Federating a batteryless sensor's energy storage improves the reliability, efficiency, and energy use of the whole sensor; however, the federated approach as implemented in UFoP [53] and descrobed in the previous section, is difficult to use, especially by non-experts, for three reasons described below.

Program-specific hardware designs are brittle. In order to use federated energy effectively, a developer needs to determine the right size for peripheral capacitors, the voltage at which each peripheral should start charging, and the voltage at which a peripheral should be deemed charged and ready to use. In UFoP, capacitor sizes and charging thresholds are static and set at design time, and software changes often require hardware changes—soldering or even circuit board revisions—to ensure good performance. Task priorities cannot be changed once deployed. The result is brittle systems with tight hardware and software dependencies and long, expensive development and debugging cycles. Even for those with hardware expertise, these systems are difficult to maintain and modify.

Static UFoP is inflexible at runtime. When using static UFoP, programmers cannot turn off peripherals when they are no longer in use or assigned to a task. These peripherals continue to charge, storing energy that

[3]Sparkfun is a popular company for DIY and maker electronics.
[4]Arduino is the definitive microcontroller ecosystem for embedded electronics makers.

may never be used and delaying more important tasks. Programmers cannot change the relative priorities of different peripherals adapting tasks at runtime—limitations that fundamentally bound application complexity and the ability to retask deployed sensors in the field.

Significant hardware complexity remains. Hardware complexities and unintended interactions can interfere with the operation of the sensor; in static UFoP this is seen in the approach to voltage regulation, logic levels, and the energy management interface. In order to maximize efficiency and availability, most batteryless sensors do not regulate supply voltages. For some peripherals (especially with RF components), voltage fluctuations will affect accuracy. Other components (like an MSP430 MCU), draw more power at higher voltages. Static UFoP does not propose a standard way to deal with these conflicting peripheral requirements. In its current form, UFoP does not manage logic levels for communication between the MCU and peripherals, and designers must carefully tune capacitors and harvesters to keep voltage level within logic bounds, concurrently. Additionally, UFoP's energy management interface uses a polling-based approach to manage energy levels, which wastes energy in frequent threshold checking.

Today, prototyping batteryless sensing devices is challenging enough to discourage all but the most determined developers. This paper addresses this problem by 1) improving the flexibility and efficiency of federated energy storage and 2) integrating these improvements into a novel and general platform, called Flicker, for flexible and rapid prototyping of the batteryless Internet of Things.

We have developed Flicker for IoT application designers who want to develop batteryless energy harvesting devices. Flicker's goals are (1) to provide multiple hardware options in terms of peripherals and harvesting technologies, (2) realize runtime and design time flexibility, (3) enable recent advances in timekeeping and energy management on a general purpose platform, and (4) provide a platform focused on entire stack (software and hardware) usability. Instead of a single, monolithic hardware platform, we present a system of hardware modules that can be used interchangeably and software tools that can automatically analyze hardware configurations, detect incompatibilities, and help developers more easily create new applications. Flicker rethinks Federated Energy to meet the requirements of a reconfigurable platform, adds failure tolerant timekeeping, and exposes multiple standard interfaces to peripherals. An overview of Flicker's design is shown in Figure 4.9.

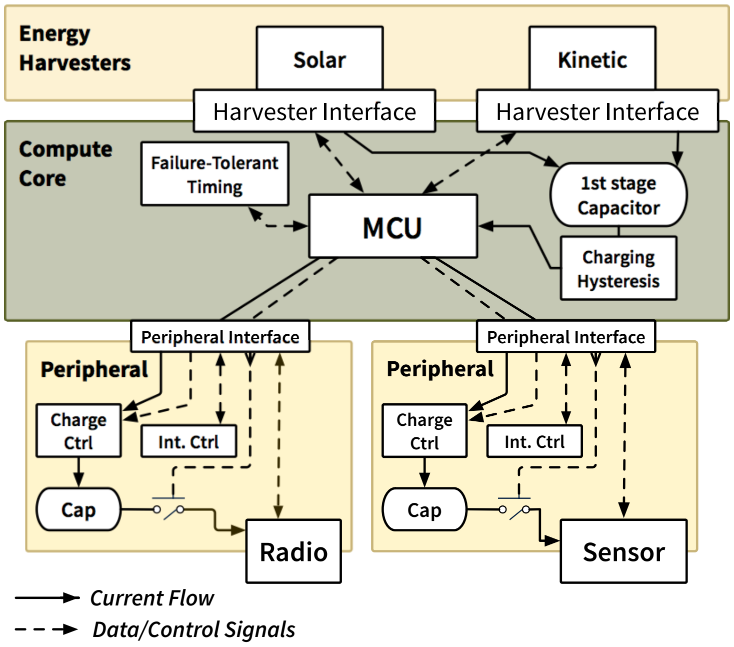


Figure 4.9: Flicker harware architecture. A multi source energy harvesting interface feeds into the first stage capacitor that powers the Compute Board. Peripherals are connected through a standard interface that maps power, and control signals.

4.3.2 Flicker Modules

Flicker modules come in three distinct varieties — compute cores, peripherals, and harvesters. A viable device configuration consists of a single compute core, one or more harvesters, and one or more peripherals.

Compute core modules include a microcontroller (MCU) (the device's main controller, programmed by the application developer and responsible for peripheral control and application logic), time-keeping functionality, ports for attaching harvesters and peripherals, and hardware support for managing federated energy stores. Any MCU can be used to create a core module, but we recommend low-power processors with on-chip FRAM, like the MSP430 FRAM series [57], that work well with a wide range of small harvesters and support efficient checkpointing. Core modules also control how peripherals are used and charged, and serve as the central hub, around which harvesters and peripherals are connected.

Peripherals connect to the core modules' peripheral interface ports, which provide power, control, and signaling for peripheral charging, in addition to analog signal lines, digital signal lines, and digital bus lines (SPI, I2C, and UART) for peripherals that communicate digitally. Peripherals can be radios, sensors, and actuators — in this paper, we focus primarily on radios and sensors, since most actuators are too power hungry for batteryless operation, but this is not a fundamental limitation. While peripheral behaviors and needs will vary significantly, each peripheral stores its own energy and contains circuitry that controls how that energy is stored and used.

Harvester modules harvest energy from a variety of environmental sources. A variety of energy sources — solar, kinetic, vibration, radio frequency (RF), and thermal — are available to application designers, but available harvesters provide the energy they harvest differently, as direct current (DC) or alternating current (AC) and at a variety of different voltages. In Flicker, harvesters are designed to provide energy in a form that can be used directly by the system. Specifically, harvester modules provide DC electricity at voltages that are high enough to support common modules and protect against reverse current flow, using a blocking diode or other similar mechanism. Reverse current protection is common in energy harvesting devices, and critical when multiple harvesters are used simultaneously (preventing one harvester from draining the energy harvested by another).

Some harvesters (namely RF energy harvested from a reader, and NFC) also combine data and energy. Flicker's harvester interface includes optional data lines, specifically to support these harvesters.

4.3.3 Reconfigurable Federated Energy

Reconfigurable Federated Energy is the crucial innovation allowing quick prototyping with a modularized platform. Converting what is usually a rigid hardware platform to one able to support a multitude of peripherals and energy harvesters. "Reconfigurable" means that programmers (or compilers) can assign (at runtime or compile time) the amount of energy to be harvested for each peripherals capacitor, the priority of charging of each peripheral, and the trip point where enough energy is stored to execute a task. This allows applications to tailor charging behaviors for different configurations of peripherals and harvesters at the prototyping stage, at compile time, or even at runtime. This makes it easy to efficiently support longer tasks that require more energy, without incurring long charging delays for shorter tasks that need less energy. This makes it easy to mix and match different tasks, and peripherals, and removes the hard coupling between peripherals and their code. Reconfiguring previously was a tedious, time-consuming chore that, using static Federated Energy [53], required hardware modifications. Many of these adjustments can now be performed easily in software.

The key challenge is in implementation — developing a Reconfigurable Federated Energy mechanism while keeping overhead (in terms of energy, processing requirements, and cost) low. We discussed the flaws in implementation in Section 5.1. To implement Reconfigurable Federated Energy we depart philosophically from the original in three ways: (1) we move from a *polling* to *interrupt* based "peripheral ready" signal generated by custom hardware, (2) we enable changing voltage thresholds (a direct proxy for energy stored in a capacitor) using digitally programmable resistor dividers, (3) we use dedicated voltage regulation depending on the peripheral. Flicker's support for reconfigurable federated energy is split between core and peripheral modules. Core modules include the first stage storage capacitor and the hysteresis control that support the microcontroller, as well as power and control lines for the Flicker peripheral interface.

Peripherals each have an individual storage capacitor, which stores the harvested energy that will be used for that peripheral and a programmable **charge controller**, which charges the capacitor only when its input voltage reaches a particular threshold set by the MCU (allowing assignment of priority, peripherals with higher priority start charging at lower voltage thresholds). In contrast to earlier federated energy systems that use static thresholds set in hardware [53], Flicker thresholds can be changed by an application over time as priorities and energy availability change, or baked in at compile time depending on the hardware modules used. Flicker also uses a programmable **interrupt controller**, that signals the MCU with an interrupt whenever the charge level exceeds a set threshold, which is also set in software at runtime. This approach replaces the more energy expensive technique used in [53] that polled the charge level on the ADC continuously.

4.3.4 Peripheral Ports

Flicker peripherals are designed to be flexible with little energy and computational overhead. As such, the peripheral interface requires common functionality to support energy harvesting, but does not constrain how peripheral-specific functions interact with the core. Instead, each peripheral port provides a wide range of connectivity options to each peripheral (such as GPIO, SPI, I2C, and reference voltages), with most peripherals only using a subset of the available pins. Ideally, enough pins would be provided to support any peripheral on any port. In reality, pins and other hardware resources are often limited. Some Flicker implementations may provide some limited hardware resources on a subset of its peripheral ports, and may not support some complicated peripherals that require an excessive number of control signals. Section 4.3.8 describes how we addressed this challenge in our implementation and how, in practice, we are able to support many common radios and sensors.

4.3.5 Failure-Tolerant Timing

Timekeeping is incredibly important for sensing, security, data provenance, and data utility, especially in the face of intermittent power with undefined (to the runtime) lengths of time between power failures. Failure-tolerant timing also allows for continuous user interface and response, and is a functional enabler for strengthening user privacy. Each core module (the compute core with the MCU) is responsible for providing a timekeeping mechanism that is robust *even in the face of power failures.* A variety of timing techniques can be used with Flicker, including remanence-based SRAM timing [103] which uses the decay characteristics of SRAM memory cells to determine the duration of power loss events. Timing failures with custom circuitry using capacitors with stable thermal properties [54] is more reliable and allows timing longer outages, but with precision reduced the longer the outage. Powering an ultra-low power real-time clock (RTC) off a small independent capacitor provides even more fine grained timing information, but at increased cost and space. The Flicker framework is compatible with any of these approaches. Our current implementation provides hardware support for all three to give broad application.

4.3.6 Auto-Detecting Configurations

Hardware changes often require software changes. In order to make batteryless prototyping fast and easy, Flicker harvesters and peripherals contain circuitry that allows the Flicker toolchain to automatically detect which modules are attached to which ports. For simplicity, Flicker uses resistor dividers to identify modules. Each module is identified by a single resistor value, which is measured by a special calibration firmware that also tests a variety of hardware functions. The autodetection process is not particularly energy efficient, and is designed to be done during calibration and testing, and not at runtime. We favor this approach over more sophisticated techniques (like using serial ID chips), to minimize cost and board size.

By autodetecting hardware configurations, many software updates can be made automatically, or by updating a simple port mapping.

4.3.7 Flicker Workflow

Flicker's modular design lends itself to the workflow shown in Figure 4.10. A developer initially selects a particular configuration she wants to try out from available peripherals and harvesters, based on her application goals and intuition. With the core module attached to a programmer, the discovery process detects which peripherals and harvesters are attached, produces a file that describes the configuration, and detects any

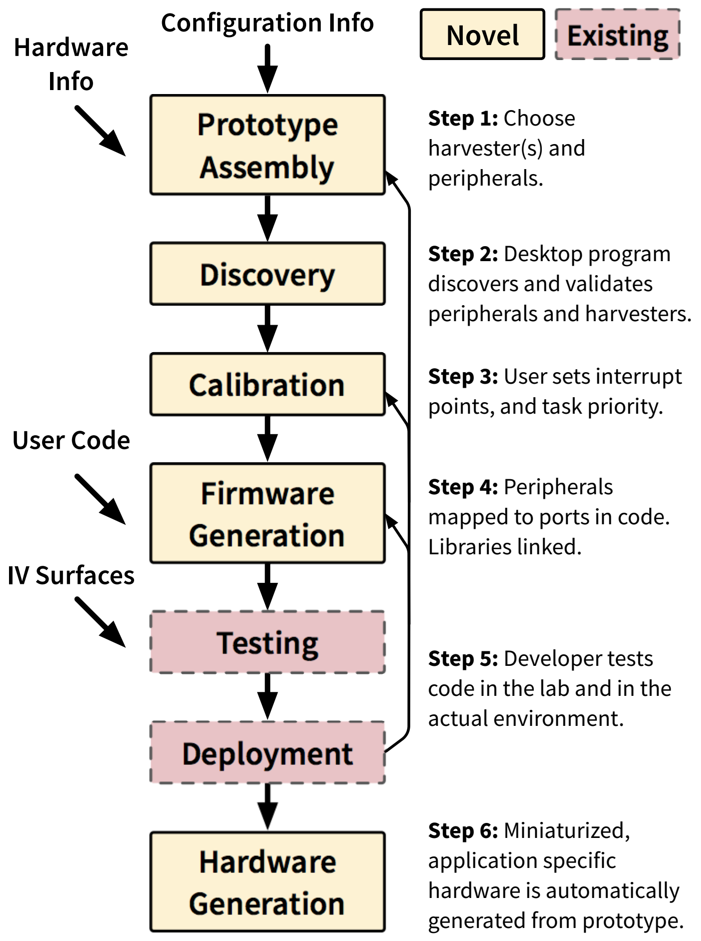

Configuration Info
Hardware Info
Novel
Existing
Prototype Assembly
Step 1: Choose harvester(s) and peripherals.
Discovery
Step 2: Desktop program discovers and validates peripherals and harvesters.
Calibration
Step 3: User sets interrupt points, and task priority.
User Code
Firmware Generation
Step 4: Peripherals mapped to ports in code. Libraries linked.
IV Surfaces
Testing
Step 5: Developer tests code in the lab and in the actual environment.
Deployment
Hardware Generation
Step 6: Miniaturized, application specific hardware is automatically generated from prototype.

Figure 4.10: Flicker Workflow.

incompatible connections (for example, a peripheral requiring I2C attached to a port that doesn't support I2C).

The configuration file contains a mapping between peripherals and ports, as well as the default threshold voltages for each peripheral's charge controller and interrupt controller. During calibration, thresholds are adjusted based on application priorities, developer intuition, and prior testing. The user's code is then combined with library code and threshold initialization code, compiled, linked, and installed on the device for testing and deployment.

Flicker's harvester interface is compatible with existing debugging tools like the Ekho [52] energy harvesting emulator and the EDB [24] debugger, for in-lab testing with I–V surfaces that are appropriate for the attached harvesters.

This process is usually iterative. Testing and deployment often indicate needed changes in the voltage

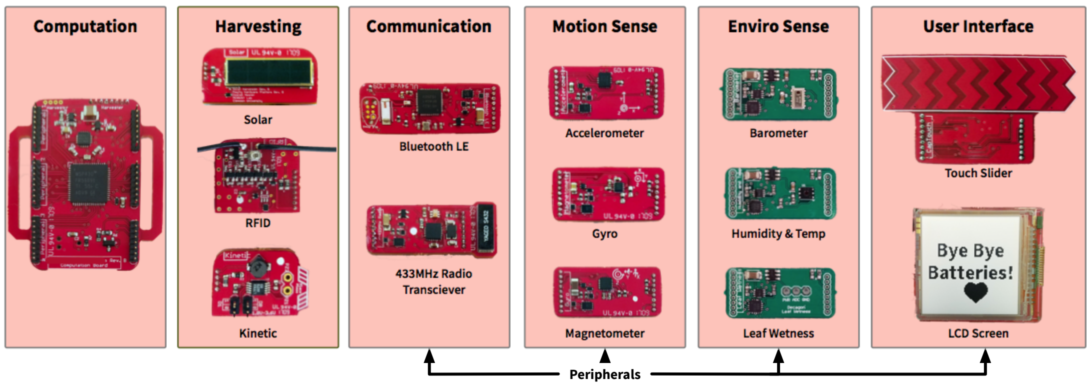


Figure 4.11: Flicker hardware modules, including a single compute core board, three harvesters, and eight peripherals for sensing and communication. Peripherals and harvesters attach to the core module, and support federated energy storage with charge and interrupt thresholds that can be dynamically adjusted in software.

thresholds or even the modules that are used. The developer makes adjustments to their configuration and application code as many times as it takes to produce a configuration that works well. With Flicker, these design iterations, which have traditionally taken days or weeks per iteration, can often be done in minutes.

When testing is complete, the developer may want to take the final step and generate her own custom hardware version of her configuration. Flicker's strength is flexibility and rapid prototyping, but many of its signal traces, connectors, and discovery hardware components increase device size and cost and are often not needed in a finalized device. In order to allow developers to further miniaturize their designs and adapt them to other form factors, the Flicker toolchain also generates a schematic and board layout for a configuration (without unneeded components) that the developer can modify as needed to fit her form-factor of choice.

4.3.8 Implementation

We implemented Flicker hardware (shown in Figure 4.11) and software, in order to 1) evaluate the efficacy of the Flicker approach and 2) provide a set of reference designs to the research community. We developed one core module board with connectors for three peripherals modules and two harvester modules. We also developed tools to ease the prototyping and design process, as well as created runtime libraries allowing for developer controlled adaptation for different energy harvesting scenarios in deployment. In this section, we describe the specifics of our Flicker hardware and software implementation. All hardware, software, and tools, as well as documentation and tutorials on using and extending Flicker are free and open source.

4.3.8.1 Hardware

The hardware modules, shown in Figure 4.11 enable a wide range of energy harvesting options, sensing activities and communication channels in order to support a broad range of applications. We plan to expand on this initial set of modules, and anticipate additional hardware contributions from the research community.

Compute Core: The compute core module is centered on a Texas Instruments ultra low power FRAM-enabled MSP430FR5989 microcontroller with 128K of FRAM, 2K of SRAM, and multiple communication and analog ports. This iteration of the core has three peripheral slots, each peripheral slot has an SPI and analog connection, while two have UART connections, and one has an I2C connection. Due to limited MCU resources, not every peripheral slot has I2C or UART. Any peripheral can be attached to any port, but some peripherals will not function correctly on all ports. Our prototype supports only a single I2C peripheral at a time — an artifact that is handled by the calibration stage; if a peripheral is attached to an incompatible port, the compilation process is stopped and the developer informed. The compute core also has a voltage reference, an Abracon AB0805 RTC supplied by a small 10 µF capacitor, and a remanence timekeeper. The latter able to time power outages upto 19 minutes, providing the developer a sense of time. A low profile Tag-Connect programming interface on the PCB allows firmware upload by the developer and calibration routines to be executed.

Universal Peripheral Interface: Each peripheral has a charge and priority controller, an interrupt controller, a storage capacitor, identification circuitry, and a power gating switch. The controllers are implemented with a dual digital potentiometer and network of comparators which gate harvested energy, based against a stable reference voltage supplied by the Compute Board. These potentiometers set the voltage on the capacitor that triggers an *interrupt* to the MCU, and the voltage on the first stage capacitor that triggers *charging* of the peripheral capacitor. Additional circuitry gates the power to the actual peripheral (radio or sensor), controlled by a pin from the MCU, and handles the identification voltage divider.

Each peripheral and harvester has an identification voltage divider (and therefore a set voltage out of the voltage divider) that can be read in discovery mode. The mapping of voltages to individual modules is given as input to the discover stage from a static file. Because of ADC voltage constraints, reference accuracy, and voltage divider noise, the maximum number of peripherals possible (without risk of mis-identification) in the Flicker ecosystem is 128.

Finally, the peripheral interface breaks out control pins, analog pins for direct sensing, and digital pins for communication with peripheral components. This common interface enables a broad array of sensing and communication peripherals as detailed below.

Environmental Sensing: Gathering information about a sensors outdoor environment is a common need for many sensing deployments, including greenhouse sensing, geo-spatial deployments, and others. Our current Flicker prototype has three peripherals for sensing information about the environment. A low power NXP MPL115A digital Barometer is used for pressure sensing, it communicates over SPI to the MCU. The barometer operates up to 5.5 V. The dynamic federated energy circuitry can take advantage of this voltage range, independent of the MCU. A Silicon Labs Si7021 Humidity and Temperature sensor provides relative humidity readings over an I2C interface. Finally an analog peripheral is connected to a Decagon Leaf Wetness sensor for gathering coarse information about the moisture and water needs of plants in a greenhouse. This peripheral can be used with any analog sensor by soldering power and signal lines to the headers.

Motion Sensing: Motion sensing has broad application for mobile sensor networks, wearables, infrastructure monitoring, and even manufacturing. We developed three peripherals for motion sensing using components commonly seen in the sensor networks and batteryless sensing communities. The Analog Devices ADXL362 is used for acceleration measurement, connected via the SPI bus. This IC only draws a few nanoamperes when sensing at low speeds, meaning that the size of the Accelerometers peripheral capacitor can be set to less than 1 µF for quick charging and increased availability. The Gyro peripheral is equipped with an STElectronics L3GD20H which enables gathering of angular velocity, which can be used in multiple applications in wearables that monitor the motion of the wrist (example: bite counting). The magnetometer peripheral uses the Honeywell HMC5883L, enabling a 3-axis compass functionality over a I2C interface.

Communication: Communication is a necessary part of nearly every sensing device, therefore it is essential for Flicker to support a broad range of communication modalities. We built a low frequency (433MHz) radio transceiver peripheral using the CC1101, a common radio in the WSN community. The Texas Instruments CC1101 uses a small chip antenna, and communicates via SPI to the microcontroller. Bluetooth Low Energy is a popular communication interface between phones, wearables, and sensors. We built a BLE peripheral that supports peripheral and central modes, using the the Nordic nRF51822 System-on-Chip and a small chip antenna. The nRF51822 SoC is programmable, and the BLE peripheral includes a low profile programming port for changing BLE behaviors. Both active radios voltage is regulated to minimize odd RF behaviors from an unstable supply. In addition to the active communication peripherals described, we have also implemented a passive communication scheme, using RFID backscatter based on the UMich Moo[136], allowing ultra low power two way communication with an RFID reader.

User Interface: We implemented user facing peripherals to allow testing and experimentation in the design fiction of intermittent computing. These peripherals make it simpler to implement wearable or embedded devices that interact with a user. We developed two peripherals, a 1.28 x 1.28 inch low power SHARP display, which draws 5 µW to hold a static image, and a capacitive touch sensor with eight buttons in a slider configuration. These two peripherals will enable a rich set of interactions and allow experimentation inside the new design space of intermittent displays, interaction, and computing.

Energy Harvesting: Our current Flicker Compute Core provides two slots for energy harvesting modules for multi source harvesting. Only one of the slot allows for energy sources that also are used as a communication medium, like RFID backscatter described in the previous section. Currently Flicker supports **three** energy harvesting modalities.

The **solar** harvester module is equipped with a 22 mm by 7 mm Ixys solar cell that supplies up to 4.5 V. Charging is accomplished through a Schottky diode on the positive solar output to prevent reverse leakage from the first stage capacitor. The **kinetic** harvester uses a Linear Technology LTC3588 to harvest energy from piezoelectric materials. The IC is used as a low quiescent current rectifier and settable buck boost regulator. The regulated voltage output of the LTC3588 can be set by the developer before deployment using pin headers, as this voltage will depend on the piezoelectric used. The **RFID** harvester is based on the UMich Moo[136]. The module harvests energy from an ultra high frequency RFID reader such as the Impinj Speedway. It uses a charge pump built with off the shelf components, and allows for tuning pre-deployment using a variable capacitor accessible to the developer. Additionally, the harvester is equipped with circuitry that enables backscatter communication with the reader, which can be initiated by the MCU.

Mechanical Design: The mechanical design of the hardware factors into both the *flexibility* (in terms of deployment ability) and *usability* (in terms of ruggedness and comfort) of a platform. We chose to tradeoff size of the platform for ruggedness, by using larger pin connectors for modules instead of smaller but much more frail, board-to-board connectors. We also put cutouts on the compute board (increasing the size) that allow a watchband to be connected to the platform, enabling a quick wearable.

Cost and Size: The final cost and size of a fully assembled prototype varies with the peripherals chosen. An assembled prototype will have a maximum size of 61 mm by 36 mm if equipped to harvest RFID and solar energy, sense acceleration and pitch, and send sensor data with the CC1101 as shown in Figure 5.7. For the described prototype, we estimate the cost of components and printed circuit boards to be near $200 a piece in a small batch of ten. At scale, we anticipate the cost of prototyping with Flicker to be significantly reduced.

This cost in term of time and finance even at small batches is **an order of magnitude lower** than designing and assembling custom hardware when an application changes, or components are found to be defective.

4.3.8.2 Software

Flicker includes supporting software and firmware for managing each stage of the prototyping pipeline. These tools are meant to streamline developer effort and save time from design, to runtime, to deployment. We describe the implementation details of each piece below.

Discovery and Calibration: A combination of python scripts on the desktop and custom firmware on the Compute Core MSP430 handle the discovery and calibration phase of the firmware upload process. When discovery starts, firmware is uploaded to the MSP430 on the Compute Core. The discovery firmware sets the discovery pin of each peripheral and harvester high to power the identification voltage divider. The voltage of each connected module (harvester and peripheral) is read and then stored in a predetermined memory location on the MSP430. The MSP430 then goes into a wait mode. On the desktop side, once the wait mode starts, the python script interfaces with the MSP Debug Stack and programmer to download the memory locations on the MSP430 where the peripheral voltages are stored. These voltages are mapped to IDs, and then the python program outputs configuration information to the developer, and alerts on any incompatibility of peripherals.

After discovery and peripheral match error checking by the toolchain, the configuration file generated by the developer is used to set the voltage thresholds for the interrupts and charging. In this calibration phase the custom MSP430 writes the non-volatile registers on the digital potentiometers with the values defined in the configuration file (converted from voltage to digital).

Runtime Libraries: We developed runtime libraries for use with peripheral modules, many were adapted from open source code libraries (such as the Accelerometer). These libraries encompass minimum functionality of the components, allowing basic sensing and communication tasks. For the BLE peripheral we have implemented a simple forwarding mechanism over the SPI bus, allowing the MCU to treat the BLE as a radio modem. Currently backscatter is supported in hardware, however runtime libraries have not been ported from the UMich Moo MSP430 code base to the new FRAM MSP430 processor used by Flicker.

In addition to peripheral and harvester runtime libraries, we also have developed control and adaptation libraries for the timekeeper, peripheral energy and priority management function. Programmers can set voltage thresholds for both the interrupt level and the charging threshold using a simple API. They can also write these to non-volatile memory for long term application changes. We envision further runtime library development

Table 4.5: Flicker overhead

Parameter	**Value**
Timekeeper Charge Energy	39 µJ
Timekeeper Startup Time	1.1 s
Volatile Threshold Write Time	197.7 µs
Volatile Threshold Write Energy	69.3 nJ
Peripheral Voltage Range	1.7 to 5.5 V
Peripheral Current Range	0.0 to 40 mA
Compute Board Quiescent Current	5.77 µA
Peripheral Quiescent Current	4.47 µA

by ourselves and the community as more applications, peripherals, and harvesting modalities are created for the Flicker ecosystem.

Design Automation: Developers who have prototyped, tested, and even deployed their batteryless sensors using Flicker and need a more permanent, smaller, or easily scalable deployable solution, use the design automation tool to combine peripherals, harvesters, and the MCU to generate the final device. This is implemented with a combination of python and EAGLE CAD scripts that put all peripherals together in a single schematic at the peripheral interface points. We developed a python script that interfaces with EAGLE CAD (a very popular and free PCB design tool), takes the peripherals and harvesters listed by the developer, and the configuration file the developer generated for the calibration phase that defines the voltage thresholds, and creates a single schematic, with proper resistance settings baked in.

4.3.9 Evaluation

In this section we evaluate the overhead and performance of Flicker, and qualitatively evaluate how Flicker simplifies the process of prototyping batteryless, intermittent, energy harvesting sensing devices. Specifically, we quantify the usefulness of dynamically adjusted federated energy, evaluate the overhead of Flicker in terms of energy and user time, illustrate how Flicker simplifies prototyping with a real world use case, and finally discuss the usability perspectives of prototyping with Flicker. In our experiments and experience, we have found that using Flicker dramatically shortens the time to deploying a usable prototype, and enables use cases not possible with current hardware. All with low overhead.

4.3.10 Overhead

Because of Flicker's reliance on harvesting energy to power all operations, energy efficiency must be high. Table 4.5 shows the overhead of specific parts of Flicker. This table shows that Flicker trades off some energy-efficiency for flexibility. For example, the steady state quiescent current costs of the peripheral stems from the charge management and interrupt circuitry of dynamic UFoP. In a static UFoP implementation most of this cost would disappear. However, the overhead is manageable with the current set of energy harvesters. One source of overhead comes from managing the timekeeper, specifically the first time charging the small reservoir capacitor that maintains timekeeper state when the Compute Board is off. Longer off-timekeeping requires larger capacitors, and therefore more charge time and energy. We chose a 10 µF ceramic as a reasonable tradeoff between charge time, and energy cost. Another source of overhead comes from setting the voltage thresholds for charging, interrupts of peripherals. Writing the volatile thresholds must be done every time the MCU returns from a power failure, or if runtime adaptation happens in deployment. The quiescent current in steady-state of the processor, and control components like the timekeeper, and peripheral controllers, is also quite low as shown in the table.

4.3.11 Performance

In this section we evaluate and discuss different performance metrics pertaining to Flicker.

RF Harvester: The performance of the RF Harvester is comparable to the UMich Moo harvesting performance that it is based on. We connected the Flicker RF harvester to an Ekho device, and recorded the energy harvesting conditions for twenty seconds when placed within one centimeter of a small antenna, connected to an Impinj Speedway Revolution RFID reader outputting at high transmit power. We then captured the maximum power point (MPP), the maximum voltage, and the maximum current, of the energy harvesting environment. The Flicker RF Harvester's MPP was recorded at 1.4 mW, with a maximum voltage of 6.3 V, and maximum current of 1.0 mA

This power level is comparable to the Moo and WISP platforms, enabling a broad range of RFID powered applications. One major difference between the current harvester and the Moo, is that our harvester only has a two layer PCB instead of four, hurting overall RF performance. In additions, antenna tuning and careful design could increase harvesting ability. We expect future revisions to continue to improve performance, however, as is, the RF harvester is comparable to the Moo and useful for RFID powered applications.

Table 4.6: System Usability Survey (SUS) scores vs. Flicker

Interface Type	**SUS Score**
Cell Phone	66.55
Customer Premise Equipment	71.60
Graphical User Interface	75.24
Interactive Voice Response Systems	73.84
Web Pages and Applications	68.05
Flicker Platform	**84.9**

Timekeeping: Timekeeping using the ultra low power RTC is one of the critical parts of Flicker. Without an accurate clock, sensor data could be forwarded that is not relevant, and tasks may be executed that are superfluous. We chose a 10 µF ceramic capacitor as the reservoir capacitor, as this size takes 19 minutes and 40 seconds to discharge enough that the RTC fails, it's memory resets, and time is lost. If the capacitor discharges and the RTC resets, on the next power up, the RTC takes 1.1 seconds to wakeup and recharge. If the energy harvesting environment has very little available energy and cannot support the energy requirements of the clock, then Flicker could potentially not get past the initialization stage. However, this is easily overcome in software; once the Compute Board turns on, programs can sleep until the RTC sends the ready signal on it's I/O line, at which point the program can resume operation.

4.3.12 User Study

We evaluated the usability of Flicker on 19 participant drawn from a junior-level, university Computer Operating Systems course[5]. **We had the 19 students each participate in a half hour session building multiple devices with Flicker. In all, students built 76 devices and spent 9.5 hours working with the platform.** The participants rated the platform as having **excellent usability** according to results from the industry standard System Usability Survey [12] participants completed. The results of this survey are shown in Table 4.6.

4.3.12.1 Methodology:

Participants were recruited from an undergraduate, junior-level university Computer Operating Systems course consisting of Computer Science and Computer Engineering students. Each participant was asked to fill out an entry survey where students self rated their competency in computer engineering, computer science, and embedded systems, then described their previous experience with platforms like Arduino.

[5]This study was approved by our Institutional Review Board.

They were then given two pieces of documentation 1) a four page instructions handout on the Internet-of-Things and the promise of batteryless operation, including a brief overview of the motivation, applications, and difficulties in deploying and prototyping these devices, 2) a three page platform handout describing and displaying the Flicker platform, including the individual modules and their usage, as well as basic instructions on how to construct a device. Participants were then given Flicker hardware including the computation board, the Kinetic, RFID, and Solar harvesters, BLE and CC1101 radio, the motion sensors, and the leaf wetness sensor. Participants then built each of four devices (two greenhouse monitoring devices, a fitness wearable, and a earth science enabling micro satellite) described in the instruction handout by assembling devices using the Flicker modules and Flicker main compute board, described their justified their selection decisions to the study supervisor.

Finally the participants took an exit survey capturing their experience. The exit survey contained the System Usability Survey (SUS)[12], a Likert scale ten question survey administered to users for measuring the perceived ease of use (usability and learnability) of software, hardware, phones, wearables, and websites SUS is a well tested, standard method in industry and academia for evaluating systems, which provides a quantitative way to demonstrate the usability of Flicker. The exit survey also asked questions about prior knowledge, future interest with the platform, and enjoyment or distaste of the experience.

4.3.12.2 Sample:

Our 19 participants were either juniors (60%) or seniors, with 2 to 5 years of formal computing education, and 2 to 10 years of total programming experience. Participants self-rated programming abilities, and knowledge of systems and computer hardware as average or above average when compared to other students at their university. Participants nearly uniformly rated their knowledge of embedded platforms as slightly below average compared to other students and developers in industry. We note that our sample size of 19 was well above the stable size of five participants, and represents a core community (mid level computer engineering and computer science students) we would like to engage with the Flicker platform.

4.3.12.3 Results:

Tabulating and scoring the Flicker prototyping SUS surveys for each participant gave the mean SUS score of 84.9, with the median score at 87.5. SUS literature [7] states that a score of 70 is considered average, with higher scores meaning higher usability. Each participant scored Flicker above average, with most in the "excellent" usability category. Flicker SUS scores are shown in relation to other interface types in Table 4.6.

This table shows the interface types along with their average SUS scores, the average is derived from years of surveys, and a category had to have at least fifty surveys to be put in the table [7]. The SUS results show that the Flicker hardware platform has usability demonstrably well above average for prototyping batteryless Internet-of-Things devices.

In addition to the the SUS, participants were asked in the entry survey to discuss if they would use the Flicker platform in the future and in what contexts. Nearly 95% of participants agreed or strongly agreed that Flicker could be used to to create devices for many different applications. Nearly 90% or participants agreed or strongly agreed they would use Flicker on a new IoT project if it was available. The same percentage was interested in learning more about the platform and the context in the future. Every participant agreed or strongly agreed that Flicker devices could be deployed in a real environment for *short-term* use. A majority (58%) said the same about *long-term* use in real environments. A plurality (32%) agreed that Flicker devices could be deployed in a safety-critical application.

4.3.12.4 Caveats and Discussion:

Our user study investigated the usability of the core part of the Flicker platform, rapidly prototyping batteryless IoT devices by hand with one of the major expected user groups of these devices. However, the study did not look at the usability of different parts of the Flicker workflow and toolchain, including configuring the hardware peripherals, and writing software that runs on the device. We anticipate future work will attempt to fill this gap. While this study is not comprehensive, it demonstrates that the hardware devices themselves are usable, and enable a broad range of applications.

4.4 Related Work

This section outlines the related work conerning hardware and platform issues for sensors and the Internet-of-Things.

4.4.0.1 Timekeeping

The related work for remanence based timekeeping comes from both security applications, and advancements in low power technology.

RFID Security and Privacy: There are many applications for transiently-powered devices that require a method to throttle the responses from a tag. The inability of intermittently powered devices to control their response rates has made them susceptible to various attacks. For example, an RFID tag could be easily "killed" by exhausting all possible 32-bit "kill" keys. Such unsafe "kill" commands could be replaced with a "sleep" command [64]; however, lack of a persistent clock that enables wake up of the tag in time has made the use of the "sleep" command inconvenient. E-passports have been subject to brute-force attacks [4], where the key can be discovered in real time. The attack could be slowed down if the e-passport had a trustworthy notion of time. The minimalist model [63] offered for RFID tags assumes a scheme that enforces a low query-response rate. This model could be implemented using our approaches. Some RFID credit cards have used monotonically-increasing transaction counters as a proxy for time, with some cards ceasing to function after the counter rolls over [55].

Secure Timers: To acquire a trustworthy notion of time, multiple sources of time can be used to increase the confidence level an application has in a timer [111]. This method is not practical for RFID tags that use passive radio communication. The same issues prevent us from using the Lamport clock and other similar mechanisms that provide order in distributed systems [71]. This inability to acquire secure time precludes the use of many cryptographic protocols, including timed-release cryptography [84] [110].

Ultra-low Power Clocks and Timers: With the rise of pervasive computing come a need for low-power clocks and counters. Two example applications for low-power clocks are timestamping secure transactions and controlling when a device should wake from a sleep state. The lack of a rechargeable power source in some pervasive platforms requires ultra-low-power consumption. Low voltage and sub-threshold designs have been used to minimize power consumption of digital circuits since the 1970s [123]. Circuits in wristwatches combine analog components and small digital designs to operate at hundreds of nW [128]. A counter designed for smart cards uses adiabatic logic to operate at 14 kHz while consuming 11 nW of power [125]. A gate-leakage-based oscillator implements a temperature-invariant clock that operates at sub-Hz frequencies while consuming 1 pW at 300 mV [78]. These solutions, while very low-power, still require a constant supply voltage and hence a power source in the form of a battery or a persistently charged storage capacitor. However, embedded systems without reliable power and exotic low-power timers may still benefit from the ability to estimate time elapsed since power-down. Most closely related to CusTARD, is a TI-recommended technique [104] for the MSP430, that gives a hardware only, wakeup RC-timer. The technique charges a dedicated external capacitor from the microcontroller, then goes into a low-power sleep mode with clocks deactivated; the microcontroller is triggered to wake up when the capacitor voltage surpasses a threshold. CusTARD generalizes this technique to application clocks and intermittently powered, systems with unstable supply voltages. The addition of an

ADC allows for precise measurement and fine grained time. Additionally, this work quantifies the factors that impact measurement accuracy, and details the software and hardware methods for profiling CusTARD to reduce the effect of these factors, and improve overall accuracy. Moreover, we motivate this method as an enabling technology of the future Internet of Things.

4.4.0.2 Federating Energy

Currently in the literature there does not exist a federated approach that stores and manages harvested energy in hardware. However, in this section, we review state-of-the-art designs and implementations in energy harvesting and management for perpetual sensing systems.

Virtual battery [19] attempts to "virtualize" the available energy, allocating energy towards tasks. Virtual battery assumes a consistent voltage, power supply, and a known battery size. It does not consider how energy harvesting can change energy budgets. UFoP is designed to work with a volatile supply voltage, frequent power failures, unknown energy harvesting, and therefore unknown energy budget. The essential difference between these two systems is that virtual battery partitions available energy, while UFoP acts as a disruption tolerant energy manager, storing incoming harvested energy and notifying the application about changes in availability. To make a system like virtual battery work on intermittently powered devices, UFoP must first exist to hide the energy volatility. UFoP does simplify application development for intermittently powered devices in a similar way to virtual battery, in the sense that both systems inform the application what energy is available, however, UFoP manages this energy in real time and handles energy replacement. eShare [139] is another energy sharing system to balance energy supply and demand. The system's energy router consists of an array of ultra-capacitor with different levels of capacitances. Even though eShare can extend the network lifetime, this system can only be useful where power wiring is feasible.

Prometheus[61] uses two storage devices, *i.e.,* super-capacitor and lithium rechargeable battery. A solar panel charges the super-capacitor and when its voltage is higher than a threshold, it charges the battery. When the super-capacitor is exhausted, the battery is used. Heliomote [101] uses a solar panel and two AA type NiMH batteries. Energy harvesting occurs when the solar panel's output voltage is 0.7 V higher than that of the battery. When battery's voltage is higher than the solar panels, even though enough power may be available on the solar panel, a node can still draw current from the battery. Everlast [118] uses a solar panel and a super-capacitor. It charges a huge-sized super-capacitor (100 F) while tracking the Maximum Power Point (MPP) of its solar panel. AmbiMax [95] harvests energy from multiple ambient power sources (solar and wind) while performing Maximum Power Point Tracking (MPPT) on each of them. Each energy harvesting subsystem harvests energy and charges its own reservoir super-capacitors. The system is powered solely by the ambient sources when the reservoir capacitor array has a higher voltage at its terminal than a threshold. It draws power from the battery when the reservoir capacitor array's terminal voltage drops below the threshold.

Other related energy harvesting systems including TwinStar [142, 144, 143], EnHANTs [43, 44], SolarWISP [49] and the work of Yerva *et al.* [134]. TwinStar [142, 144, 143] is an add-on energy harvesting and management device, which uses ultra-capacitor as the only energy storage unit. It has a dual solar panel solution (a small boot solar panel and a big main panel to charge the ultra-capacitor). On top of the hardware, the controller maintains a high duty cycle when the voltage is high and a low duty cycle when the voltage is low. Energy-Harvesting Active Networked Tags (EnHANTs) [43, 44] can be attached to objects that are traditionally not networked, such as books. The prototypes harvest indoor light energy using custom organic solar cells, communicate and form multihop networks using ultra-low-power Ultra-Wideband Impulse Radio (UWB-IR) transceivers. In [49], the authors add solar panel (SolarWISP) to WISP (RF-powered tag) for hybrid energy harvesting. SolarWISP increases effective communication range threefold and quadruples read rate. In [134], the authors propose to add a new tier in the sensor network architecture by using energy-harvesting leaf nodes, which can communicate with battery-powered branch nodes and wall-powered trunk nodes.

Software strategies for adaptive scheduling based on the dynamic energy supply are studied in [140, 14, 68]. DEOS [140] is a dynamic energy-oriented scheduling method that dynamically adjusts the execution of tasks based on the tasks' energy consumption and the system's real-time available energy. Dewdrop [14] is a CRFID runtime that makes effective use of harvested energy. It adapts a tag's duty cycle to match the harvested power to the sensing and computation cost of tasks. In [68], the authors propose mathematical models to predict the ideal battery size and the rate of availability of harvestable energy with an assumption that the energy consumed by a node is always less than or equal to the harvested energy. Mementos [108] is a software system that transforms general-purpose programs into interruptible computations that are protected from frequent power losses by automatic, energy-aware state checkpointing. At compile time, Mementos inserts function calls that estimate available energy. At run time, Mementos predicts power losses and saves program state to nonvolatile memory.

4.4.0.3 Platforms

Flicker is the first reconfigurable hardware platform designed from the ground up for tiny, batteryless, energy harvesting, intermittently powered sensors. While many platforms have allowed some measure of reconfigurability, none support the constraints imposed by an intermittent power supply requiring careful and direct management of multiple small energy stores and timekeeping. In the sensing community there have been a wide range of platforms, reconfigurable or not, that Flicker builds on. We describe platforms, energy management techniques, and related work from the intermittent computing field and place Flicker in the literature.

Reconfigurable Platforms: In both the commercial and research communities, there are many reconfigurable platforms for sensing, and support of the Internet-of-Things. Epic[33] was one of the first sensing platforms that enabled a degree of freedom in assembling application specific sensor node. Epic required hardware expertise, and CAD tools to generate the final node, however, the core computation, wireless, storage, and programming interface were all standard. TandemStack[121] and other platforms[69] provide either a common interconnect, modular plug and play hardware, or FPGA based modules for faster prototyping and development of application specific sensing devices Recent platform advances have focused on reconfiguring part of the analog sensing component[112] instead of the full device, or focus on ultra low power interconnects for millimeter scale sensing[94]. Commercial platforms like Arduino support the concept of daughter boards, allowing for easy prototyping for the purpose of learning. Other platforms such as the Bosch IoT XDK[11], and EnOcean[34] claim to support Internet-of-Things applications, sometimes without batteries.

Each of these platforms falls short of providing a comprehensive, intentionally designed platform for batteryless, intermittently powered sensor prototyping in key ways. First, none of the platforms allow using different energy harvester, or multiple energy harvesters at the same time. This is critical because of the broad range of applications. Secondly, no platform enables reconfiguration of energy management, charging, and interrupt priority per peripheral. Without this, platform performance suffers from lower energy harvesting efficiency, decreased availability, higher failure rates[53]. Finally, each of these platforms ignores the critical need for timekeeping through power failures. Each of these shortcomings is addressed by Flicker.

Batteryless Platforms: Flicker is inspired by the United Federation of Peripherals (UFoP)[53]. Flicker generalizes the approach in UFoP by allowing changes to charge points and interrupt points, enabling retasking, and dynamic priority, as well as reconfigurability of the hardware platform.

Batteryless platforms have been built that harvest energy from an RFID reader[115, 136], indoor solar[85], thermoelectric energy[18], airflow[131], and power outlets[29]. These single application, single harvester platforms are not extensible or reconfigurable, do not consider timekeeping as a first class priority, and do not separate energy concerns to reduce failures. Energy-Harvesting Active Networked Tags (EnHANTs)[85] can be attached to objects that are traditionally not networked, such as books. The prototypes harvest indoor light energy using custom organic solar cells. Campbell et al.[17] proposed an architecture for energy harvesting in buildings based on event detection, but only allowed for a single energy harvester (indoor solar) and did not develop a prototyping platform.

Unlike previous approaches and platforms mentioned, Flicker provides an all in one plug and play solution to batteryless prototyping that includes energy management, timekeeping, and hardware design, in a modular, quickly reconfigurable, extensible platform. Flicker can be applied to a huge range of future and present applications, energy harvesting modalities, and sensing tasks.

Energy Harvesting: Previous work has characterized kinetic energy harvesting[46], developed techniques for harvesting RFID and Solar energy[48], and even harvesting solar, thermal, and vibration energy[6]. Flicker can harvest from multiple sources, but does so in a naive manner, giving priority to the harvester with most voltage potential. We view these related works as complementary to Flicker, and hope to integrate these novel multi source energy harvesting approaches into future versions of the platform.

Intermittent Computing: Other related work comes from efforts to simplify the programming, testing, and evaluation of batteryless, intermittently powered devices . Recently, tools for debugging (EDB)[24] batteryless programs, and emulating energy harvesting environments have been created (Ekho)[52]. These simplify development and allow rigorous in-lab testing. Other work has tried to simplify the programming and task management for intermittent computing[109, 81, 15]. We view this work as complementary to our own, in fact, each of these techniques could be immediately used with Flicker.

4.5 Discussion & Future Work

Flicker enables many use cases and new applications that were previously impossible without deep hardware knowledge, and large investments of time and money. However, the current implementation of Flicker has some limitations, as well as tradeoffs worth discussion. We detail these in this section.

Tuning Thresholds: Just as in previous work on Federating Energy storage for tiny batteryless sensors (FedEnergy)[53], figuring out the best voltage thresholds is difficult and imprecise. Energy environments change, and application requirements vary; for Flicker this can become more complex, as designers now have a choice of harvesters and peripherals. This ability to choose and set voltage thresholds for charging and peripheral wake up interrupts without a hardware revision gives maximum flexibility to the user, but choosing the best set of thresholds still requires careful consideration.

The added flexibility comes at a energy and space cost. Peripherals on Flicker are larger in size than peripherals using FedEnergy, because of the lower component costs of using static, hardware defined thresholds. This also means that static, FedEnergy will always be lower energy overhead than Flicker. However, the flexibility offered by Flicker is worth it, especially for developers not capable of redesigning hardware. Additionally, Flicker software tools can generate a static FedEnergy version from the Flicker version, for long term deployments and sensor building at scale.

Hardware Limitations: The current off the shelf non-volatile digital potentiometers have serious limitations when used with batteryless devices like Flicker. The high current draw of non-volatile EEPROM writes, the high start supply voltage requirement (2.7 V), the low resistance values, force some sensors and peripherals and energy harvesters out of reach. We envision that these high voltage requirements and write costs will be relaxed once non-volatile digital potentiometers start using FRAM instead of EEPROM.

Runtime Adaptation: Dynamic Federated Energy allows changing task charging and interrupt thresholds at runtime, enabling adaptation of tasks to the energy environment in-situ. However, as discussed, tuning is hard when not much is known about future energy harvesting conditions or even task schedules. Additionally, the high momentary cost of non-volatile EEPROM writes make runtime adaptation expensive unless done infrequently. If a simple metric could be developed for deciding when to adapt, and adaptation cost could be lowered by advances in low power memory, runtime adaptation could become incredibly useful.

Toolchain Integration: Flicker can be used with the two main tools for working with intermittently powered devices, Ekho, and the Energy Interference Free Debugger (EDB), without any hardware changes. Ekho emulates energy harvesting conditions, so an Ekho device can be plugged directly into one of the harvesting ports of the Compute Board, just as with any other device. Two Ekho devices are required to emulate multi source harvesting. With EDB, there are some constraints. Since EDB needs access to a few I/O pins, as well as the capacitor voltage for full functionality (breakpoints, watchpoints, and energy guards), an entire peripheral slot must be dedicated to using EDB. In the future, we hope to create a dedicated port for using EDB so that all three peripheral ports can be used when debugging.

4.5.1 Future Work

Flicker gives developers a powerful prototyping and design tool for batteryless, energy harvesting, intermittently powered sensors. While powerful and complete, there are many areas we envision where Flicker could be improved and augmented. We describe some potential avenues for future research in this section.

Hardware improvements: Now that the baseline hardware, software, and firmware are implemented in the Flicker ecosystem, other harvesting modalities, sensors, and application can be imagined. In the short-term, we plan to support user interface components like buttons, vibration motors, and ultra low power displays. We are currently implementing NFC and thermal energy harvesters to support more environments and applications. In order to support long-term deployments with Flicker hardware, we plan to support mass storage peripherals using microSD cards or large FRAM memory ICs. Flicker support for multi source harvesting wastes energy opportunities as it prioritizes the energy harvester with the highest voltage potential. A more sophisticated charge circuit to more efficiently support any configuration of harvesters is needed.

An interesting area of research would be investigating changing energy storage capacity on a peripheral level. This would allow more dynamic tasks if a single peripheral supports a large dynamic range of tasks, from a total energy perspective. For example a radio that has a low power listen mode, and a high power transmission mode. Supporting both these modes could be made easier with variable energy storage.

Software: The discovery, calibration, and firmware toolchain could also be streamlined to reduce the time and configuration cost the developer must pay to get started. Certain harvesters or peripherals may only work at certain voltage levels. If the compiler has knowledge of these levels, it can review the configuration file supplied by the developer, as well as the peripherals on the Compute Core, and detect potential errors before any code development or deployment is done. Treating hardware configuration errors as compiler warnings or errors. We also imagine that certain configurations could be baked into the Flicker software for getting new users started with the platform, decreasing the barrier to entry and increasing user comfort.

Community Building: Flicker is an open source, open hardware initiative that seeks to empower sensing experts and non experts alike to build comprehensive batteryless sensing applications for the vision of the Internet-of-Things. Important to our effort is developing materials that help our community, and generate We anticipate immediate future work centered around designing documentation, writing tutorials and new libraries, formalizing the batteryless sensing toolchain, and engaging in community building and outreach.

4.6 Conclusions

Batteryless, energy harvesting, intermittently powered sensors are an emerging class of device that defines and enables the vision of the Internet-of-Things. Despite the importance of these devices, current sensing platforms are application specific, lack recent advances in energy management and timekeeping, and are limited in flexibility.

In this chapter we have presented Flicker, an open-source, open-hardware prototyping platform intentionally built for batteryless, energy harvesting, intermittently powered sensing. Flicker depends on Federated Energy and Remanence Timekeepers, two solutions presented in this chapter. With Flicker, developers and system designers can quickly prototype devices for new applications in many fields, with many energy harvesting modalities. Flicker is comprised of a reconfigurable hardware platform that lets designers replace sensor, harvester, and communication peripherals at will, without hardware experience or design abilities. Flicker hardware manages harvested energy in a novel, and dynamic way, allowing for easy adjustment of charging thresholds and interrupt routines depending on application, energy harvester, peripheral, or any other developer constraint. Flicker also includes software tools to streamline the prototyping process, all the way through to deployment.

Flicker is extensible by platform and software developers who want to add new sensors, new runtime techniques, or even new operating systems. We implemented Flicker in a small form factor for multi source harvesting from RFID, Solar, and Kinetic energy sources. Our Flicker implementation supports a broad range of environmental and motion sensors, and communicates through Bluetooth LE, low frequency radios, or RFID backscatter. We believe Flicker will support the emerging batteryless sensing community and bring about exciting new applications, and research directions.

CHAPTER 5

MAYFLY: LANGUAGE SUPPORT FOR BATTERYLESS SENSING

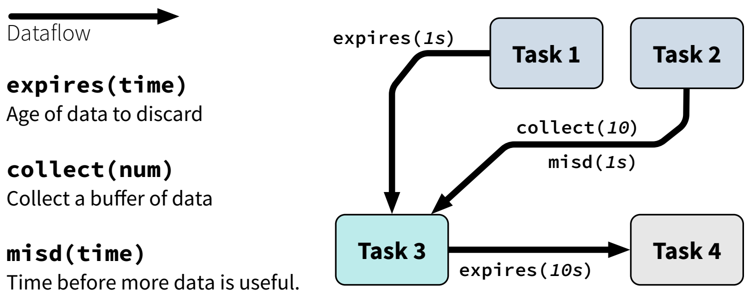


Figure 5.1: Mayfly programs are made up of a graph of connected tasks with clearly specified timing constraints on the data generated by each task.

Tiny intermittently powered computers can monitor objects in hard to reach places maintenance free for decades by leaving batteries behind and surviving off energy harvested from the environment—avoiding the cost of replacing and disposing of billions or trillions of dead batteries. However, creating programs for these sensors is difficult. Energy harvesting is inconsistent, energy storage is scarce, and batteryless sensors can lose power at any point in time—causing volatile memory, execution progress, and time to reset. In response to these disruptions, developers must write unwieldy programs attempting to protect against failures, instead of focusing on sensing goals, defining tasks, and generating useful data in a timely manner. To address these shortcomings, we have designed Mayfly, a language and runtime for timely execution of sensing tasks on tiny, intermittently-powered, energy harvesting sensing devices. Mayfly is a coordination language and runtime built on top of Embedded-C that combines intermittent execution fragments to form coherent sensing schedules—maintaining forward progress, data consistency, data freshness, and data utility across multiple power failures. Mayfly makes the passing of time explicit, binding data to the time it was gathered, and keeping track of data and time through power failures. We evaluated Mayfly against state-of-the art systems, conducted a user study, and implemented multiple real world applications across application domains in inventory tracking, and wearables.

An intermittently-powered sensor may fail at any time, between any two lines of code, with little warning, for unpredictable lengths of time. Time can be difficult to measure, and execution times can be difficult to predict. Programs may lose data, corrupt data, or fail to make forward progress on long-running computations. Forward progress can be preserved, if applications checkpoint their state to nonvolatile memory (*e.g.,*Flash or FRAM) before a failure [107]. With programmer defined memory fences [82] programmers can keep non-volatile data structures consistent. Breaking programs into tasks and putting global data in channels, can help lighten the developer's cognitive load [23]. Developers can even use physical hardware properties to estimate how long a device spends without power [102]; however, responding to dramatically unpredictable execution delays remains a daunting challenge for developer, in spite of these advances.

As a sensor executes over time, data age, priorities change, and opportunities come and go. Sensor data, once urgent, may only be useful for a few minutes or even seconds. When data expire after long outages, partial computations may need to be discarded and possibly restarted. After short outages, an application may pick-up where it left off. On power-up, an application's priorities may have changed. A time sensitive task may take precedence over a work-in-progress. A task that has repeatedly failed to complete (due to expired data), might be swapped out for a low-power alternative.

Each of these cases are easy to understand, and easy to handle with traditional battery-powered sensors, however, each of these cases are difficult to implement with today's languages and runtime tools on intermittently powered, batteryless sensors. Common imperative programming languages, like C, ignore time and how it relates to data. Traditionally, this has not mattered as programmers expect tasks to run quickly and sequential instructions to execute close together in time. An intermittent C program that discards expired data, schedules tasks appropriately in spite of power failures, or adapts to changing energy conditions will invariably be full of explicit time checks and cluttered with extensive branching logic.

This chapter argues that 1) those who use existing languages to program intermittently-powered devices are doomed to frustration and complication, 2) those complications prevent capable programmers from creating compelling, sophisticated intermittent applications, and 3) programming for intermittent power can (and should) be simple. We call on the research community to rethink how we write programs for intermittent devices, and we specifically address the issue of time as a key to enabling sophisticated applications to intermittent batteryless sensors.

In the following sections, we describe why it is difficult to program intermittent sensors and introduce the Mayfly[1] Language and Runtime. The Mayfly *language* is a declarative, graph based programming language

[1]Mayfly is named after the short-lived aquatic insect.

that enables developers to focus on application policy, and sensing goals, and timing of sensing tasks while reducing the cognitive burden of intermittent programming (shown in Figure 5.1. The Mayfly *runtime* is a task scheduler that maintains temporal aspects of data automatically across power failures. We evaluate Mayfly against state-of-the-art systems and explore the language in the context of real-world applications in active RFID based inventory tracking, and activity recognition on wearables. Additionally, we present the results of a user study comparing the utility of our approach and the standard approach to programming batteryless sensors.

5.1 Time and Data for Batteryless Sensing

Despite the promise, batteryless sensors are difficult to program, debug, and deploy, because they lose power frequently and often unpredictably. Each power failure resets the device's volatile memory, stack pointer, registers, and sense of time. Once enough energy is harvested to turn back on, the sensor returns control to the start of the program (`main`). This is shown in Figure 5.2. This style of execution is shown (from programmers perspective) in Figure 5.3, where a source program is executed using both a continuous power supply and an intermittent power supply. With frequent failures and unreliable power, programming becomes a best-effort probabilistic game. As energy becomes available, developers piece together execution fragments hoping to satisfy developer constraints. This **intermittent computing model** causes developers to struggle with what are generally simple tasks—like timestamping data, ensuring that data structures remain consistent, and maintaining forward progress on long-running computations.

When a power failure is approaching, developers can preserve forward progress with checkpointing — saving some [107, 130] or all [5] of the program's state to nonvolatile memory (like Flash or FRAM). Checkpointing, as shown in Figure 5.3, allows programs to correctly continue where they left off after the last power failure. However, checkpointing is costly in energy and memory, as demonstrated in Chain [23]. Chain proposes that developers divide programs into discrete *tasks* that share data (kept consistent by Chain) through *channels*, eliminating checkpointing cost. Each of these methods work well in stringing together execution fragments into cohesive programs, enabling long running computation. However, each of these methods ignores the fact that an application's success or failure often depends on when tasks are executed and when data are collected, processed, and communicated. Application designers often understand these constraints (albeit imperfectly), but lack effective tools for communicating them in code.

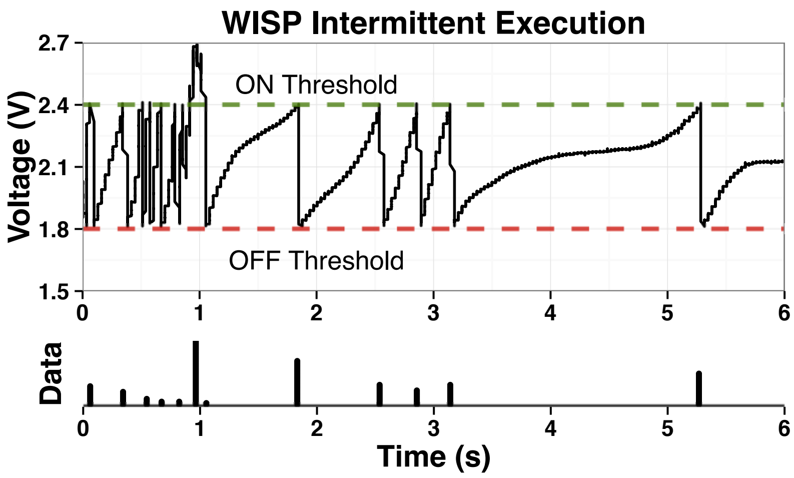
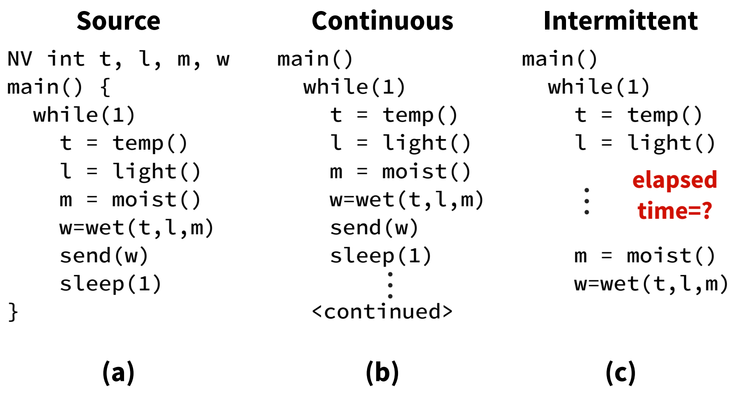


Figure 5.2: **Voltage supply of a WISP, powered by RFID reader gathering acceleration readings across power failures for activity recognition. Data is gathered at a variable sample rate depending on the *available energy*. This volatility may negatively influence quality of applications and reduce accurate recognition.**



```
NV int t, l, m, w
main() {
  while(1)
    t = temp()
    l = light()
    m = moist()
    w=wet(t,l,m)
    send(w)
    sleep(1)
}
```

```
main()
  while(1)
    t = temp()
    l = light()
    m = moist()
    w=wet(t,l,m)
    send(w)
    sleep(1)
      ⋮
  <continued>
```

```
main()
  while(1)
    t = temp()
    l = light()
    ⋮
    m = moist()
    w=wet(t,l,m)
```

Figure 5.3: Execution of a greenhouse monitoring program (a). The expected continuous execution (b) is shown versus the intermittent execution (c) caused by volatile energy harvesting environments. Forward progress is preserved—however if enough time has elapsed between lines of code, it may not be worth continuing with execution as intermediate results may not be relevant.

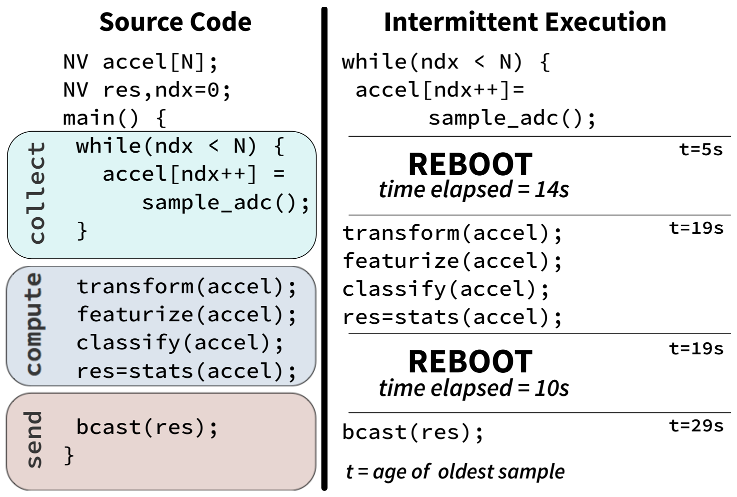
5.1.1 Complexity of Timekeeping

With remanence-based timing techniques [102, 54], batteryless devices can measure time across power outages; however, *reasoning* about the impact of unexpected delays in intermittent software is a complicated and error-prone process. Nearly all sensing applications have temporal requirements, and whether we are monitoring a user's heart rate or a plant's water needs, the data we gather is often only useful when those requirements are met. A batteryless program's progress is difficult to predict, and current programming models ignore the relationship between data and time, forcing programmers to deal with the added complexity. As developers add explicit checks that consider data expirations, sensing rates, and temporal signal properties, their programs become difficult to maintain, debug, and understand. We discuss the specific issues below:

Real data often ages. If a sensor gathers data, dies, then turns back on, the time before that data is delivered (and can be used) might be seconds, minutes, or even hours. If data is used to control a process (plant watering in a greenhouse) or report in-the-moment information (like the user's current heartrate), data gathered may not be useful if they are too old. After a power outage, applications may continue processing fresh data but save energy and time by discarding incomplete results derived from old data. One task may, of course, depend on another, and preserving data freshness requires time stamp checks throughout an application's code. This is shown in Figure 5.4—an accelerometer based wearable activity recognition program that **collects** data, **processes** that data to identify an activity, then **sends** its results over the radio. Because of long power failures, that data has aged significantly and may or may not not be useful to the application. Either way, it is not specified explicitly in the program. Compounding the problem, the data gathered in **collect** is *mixed sampling rate with an untrustworthy clock*—untrustworthy because the device may lose power before recording the time and assigning that time to the data point, mixed sampling rate because power outages may affect the sampling rate. Using timing information to inform sensing allows batteryless sensors to use energy more effectively, sense only when they need to, and not transmit or process old data. However, how this timing information is used is currently application- and developer-dependent because timing information is implicit in current programming models.

Real applications often tolerate temporal variation. A pedometer's step-counting algorithm may call for 30 accelerometer samples collected at 10 Hz. Depending on how it detects steps, that same algorithm may give accurate results as long as the 30 readings fall within a 4 s window and as long as no two readings are taken within 80 ms of each other. These relaxed requirements will be much easier to satisfy with frequently-failing hardware, but more complicated for application designers to implement. Meeting stringent timing requirements is difficult in batteryless sensors that exist in unpredictable energy situations.

Intermittent data complicates programs. When devices are tethered or battery-powered and power supplies are stable, data collection and management is difficult, but straightforward. When devices operate intermittently, programmers must add code to track when data are generated, when data have expired, and when it's advantageous to gather more data. Writing code that knows when to discard the whole, or part of a buffer of data, that properly timestamps data and checkpoints execution is tedious. Tedious programming tasks lead to



```
NV accel[N];
NV res,ndx=0;
main() {
 while(ndx < N) {
   accel[ndx++] =
      sample_adc();
 }
 transform(accel);
 featurize(accel);
 classify(accel);
 res=stats(accel);
 bcast(res);
}
```

```
while(ndx < N) {
 accel[ndx++]=
      sample_adc();
transform(accel);
featurize(accel);
classify(accel);
res=stats(accel);
bcast(res);
```

Figure 5.4: On left, source code of wearable activity recognition program, labeled into tasks *collect, compute, send*. On the right, a possible execution of the activity recognition shows how data ages through power failures. If the system that uses this data makes data-driven decisions every 10s, the 29 second old data may not be worth transmitting. Sensing data becomes mixed sample rate, and clocks become unreliable because of intermittent execution and variable length outages.

sloppy implementations, which leads to bugs in deployment.

Languages assume continuity. When using current programming models, programmers assume that sequential instructions will execute one after another with effectively zero time between them. With intermittent execution, the actual time between two instructions (or tasks, for languages like Chain) could be microseconds, seconds, minutes, hours, or even days because of unpredictable placement of power failures. If the time between failure, and resuming execution (the time between the device losing power, checkpointing, and then resuming once power is restored) is *almost zero*, then most likely continuing from where execution left off is best. However, when longer delays occur, intermediate execution and results might not be useful going forward. The programmer's **intent** is not explicit—as regards time—with current programming models, so the **runtime** has to make assumptions. Current languages ignore the timeliness property of data caused by intermittent execution, leaving the developer with few ways to strongly and safely express time constraints of data.

Tedium leads to errors. With existing languages, developers can use remanence timekeeping methods [102, 54] to timestamp data. While this considers the problem of time with intermittently powered devices, it still causes a cognitive and tedious burden on the developer. Short sensing programs that perform simple sensing tasks quickly explode with timing and consistency checks, recovery methods, data collection, and age management heuristics. These heuristics are the same every time: the developer is going to re-implement specific time checks for similar situations in every program they write. Worse, this is a painful implementation process that is difficult to debug, maintain, and verify.

A New Approach: Due to the limitations mentioned in this section, we and others have struggled to build interesting batteryless applications. In answer, the next section describes Mayfly, a task based language that does away with checkpointing and integrates time management as a first order concern, designed specifically to help designers create sophisticated batteryless sensing applications. With Mayfly, developers can coordinate time-sensitive data and tasks without getting lost in checkpoints, memory fences, and timekeeping. Mayfly automatically takes care of common issues like data expiration and time sensitive data collection. Additionally, Mayfly handles computational dependencies and ensures they are co-located in time despite interruption by power failures.

5.2 Mayfly Language & Runtime

We developed the Mayfly language and runtime to enable developers to easily reason about the relationship between time and sensor data in intermittently-powered, batteryless sensing applications. Mayfly is designed to simplify or eliminate the time management and intermittent programming challenges described in Section 5.1 in three key ways:

1. **Simplify Time Management:** Developers should not concern themselves with recording and managing timestamps and dealing with timing uncertainty at a low level. In Mayfly programs, the developer can focus on the high level application sensing goals instead of low level, error prone, timekeeping drivers.
2. **Enable Dataflow Control:** Gathering, computing, and transmitting accurate and timely data is a core function of all sensors. Despite intermittent execution, Mayfly must provide the ability to coordinate dataflow in spite of power failures and unpredictable energy environments.
3. **Provide a Usable Programming Model:** Language and tools are useless if no one can use them. Mayfly focuses on (and evaluates) providing a high level of usability, reducing distractions and enhancing developer understanding of program execution and purpose.

The Mayfly *language* simplifies the development of batteryless sensing applications—enabling programmers to confidently write intermittent programs. The Mayfly *runtime* efficiently schedules programmer defined tasks in an unstable energy environment, carefully timekeeping, managing relevant data and check-

LIST OF LISTINGS 5.1: A Mayfly program for greenhouse monitoring

```
// Optional global policies
{: scheduling_policy(FINISH_FLOW); :}

// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int light, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge constraints
light -> wet {expires(10s)}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10)}
```

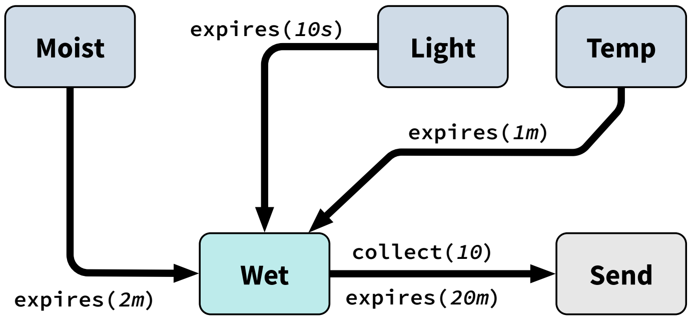


Figure 5.5: This figure shows the three constructs Mayfly needs in a program; task definitions, flow definitions, and constraints. Also shown are optional policy information.

pointing progress. Together, the Mayfly language and runtime support emerging batteryless applications for the Internet-of-Things and beyond.

5.2.1 Language Overview

A Mayfly program is a directed data-flow graph; where nodes are tasks and edges define the flow of data and their associated temporal constraints. *Tasks* are connected by *Flows*, with data flowing through the graph as tasks execute. Edges are annotated with *Constraints* that describe how to treat data as it flows through the task graph. Each set of connected tasks is a *Task Graph*. The Task Graph is executed by the runtime opportunistically, one task at a time, from the top-most tasks (the tasks with no inputs) to the bottom-most tasks (the tasks with no outputs). Representing a program as a task graph allows developers to quickly and clearly see the structure and purpose of an intermittent program without having to worry about the effects of intermittent execution. From these logical constructs each program is composed of three mandatory syntactic constructs, 1) task definitions, 2) flows, and 3) constraints, and optional policy information, all detailed below and shown in Figure 5.5 and Listing 5.1.

5.2.2 Tasks and Flows

Tasks encapsulate a single purpose; they are a logical grouping of Embedded-C code that accomplishes a single objective—for example, collecting temperature readings, processing a buffer of data, or interfacing with a radio to send a packet. These tasks correspond to C functions written by the programmer (possibly taken from an existing code base), and can involve computation, use of external sensors, or communication. A task's inputs are defined by its incoming edges, and a task's outputs are defined by its outgoing edges.

Tasks are treated as atomic units of computation, and must be executed without being interrupted by a power failure. If a tasks execution is interrupted, the runtime will rollback any intermediate results, and try to execute the task again at the next available opportunity. After a task completes, its results are stored in nonvolatile memory, where they become available as inputs to other tasks. In this way, Mayfly guarantees that forward progress will be preserved, with the caveat that forward progress is only useful *as long as timing requirements on data (specified by the programmer) are met*. Figure 5.4 shows how a wearable activity recognition program (in C) could be divided into the tasks `collect, compute` and `send`. These tasks could be further subdivided if developers expect low energy availability in deployment causing many interruptions. For instance `collect`, could be broken into `transform`, `featurize`, `classify`, and `stats`. Tasks are specified using the following syntax in a Mayfly program. These task definitions precisely define the data types and sizes each task expects as input, and the data the task generates.

```
// Task definition
task_name (TYPE input, ...) -> (TYPE output, ...)
```

Flows give structure to the task graph, defining the relationship between tasks. Developers connect the tasks to each other, giving explicit dependency information to the runtime. This also has the effect of making program execution explicit, despite the frequent power failures batteryless devices go through. This is shown in the first example, in the code listing below. Flows enable conditional execution between tasks, through the use of predicates between tasks. With predicates, a programmer can adjust program behavior in response to intermediate results, environmental input, or even user actions. The second example below shows how a predicate flow can be used to divert information to either of two actions in a wearable activity recognition program. The `compute` task has two outputs, a measure of the classification `error`, and the identifier of the `activity` recognized (in this example, `WALK` or `RUN`). The predicate states that if the compute task generates a `RUN` output, then route data to the `send` task for broadcasting the information to a basestation or the users phone. Otherwise route to the `log` task, to record the data locally.

```
// (1) Flow for activity recognition
collect -> compute -> send

// (2) Predicate
// compute -> (int error, int activity)
collect -> compute[_,RUN] -> send
compute[_,WALK] -> log
```

Policy information for the program is optionally given at the top of the program. Policy settings allow developers to specify global attributes of the program itself, select runtime heuristics, and pass on hints to the runtime about what the developer expects of the environment and the application. This policy could define the scheduling method to use, choosing between a scheduler that prioritizes moving new data through a graph, or one that focuses on finishing the processing of older data through the graph. Policy can also change which metric to use for determining priority, or which flow to prioritize.

```
// Program policies
{: scheduling_policy(FINISH_FLOW); :}
```

Tasks, and Flows (with some help from policy) guarantee that developers will make forward progress in a program (assuming energy is available to complete individual tasks). Additionally this structure lets developers quickly and easily understand the logical execution and purpose of a Mayfly program, while providing a structure which can be annotated to explicitly define timing related constraints for data generation and processing.

5.2.3 Timely Data Constraints

Constraints describe how data is treated as it flows through program tasks (the Task Graph). Constraints acknowledge that not all data is equal, and in fact the value often depends on the time the data was gathered, or how much repetitive data is available in the same timespan. Constraints are provided by annotating the edges between tasks with three possible constraints: *expires*, *minimum-inter-sample-delay*

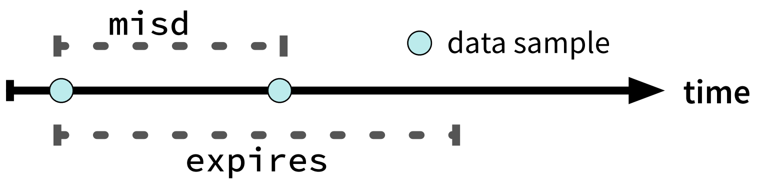


Figure 5.6: This shows how the constraints `expires` and `minimum-inter-sample-delay` (or `misd`) work together, letting the runtime know exactly which data is most valuable to gather and when.

(MISD), and *collect*. By using these three constraints, developers can specify what valued data means to their final application.

`Expires` tells the runtime how long data can sit on an edge before it loses value to the application. In programs with a stable power, processing is usually predictable, and developers take appropriate design-time measures to ensure that data is delivered while it is fresh, instead of explicitly checking timestamps.

In intermittent devices, as start-to-end processing times vary more dramatically, timestamp checking becomes both essential to efficient operation and tedious, requiring time checks at least after every reboot. In checkpointing systems, this timestamp gathering is also prone to failure as it is easily interrupted, meaning that data may appear to be timely, but in fact has an errant timestamp.

`MISD`, or minimum-inter-sample-delay throttles the data rate so that the runtime does not generate more data than needed by the application. When power is reliable, developers use timers to regulate sampling rates—an approach ill suited to intermittent operation (reboots reset timers). Instead, Mayfly developers explicitly tell the runtime how long after generation before more data is useful. MISD is a runtime hint from the developer that makes data collection more efficient. MISD acknowledges that gathering new data that has been specified by the developer as useless, is counterproductive and wastes energy. The best thing to do is wait (put the processor in sleep mode) till new data is useful again (according to the developer), or enough energy has been gathered to execute a different task. This mechanism is shown alongside `expires` in Figure 5.6.

`Collect` takes the tediousness of gathering a set of data off the developer, and onto the runtime. Rather than look at a single data point, many sensing applications collect multiple data and perform operations on them. With reliably-powered sensors, getting a buffer of data for processing, that is all fresh, and was gathered at a *consistent* sample rate is trivial. When execution becomes fragmented creating these buffers gets difficult, as now the programmer must replace expired samples in addition to all other sensing tasks. Memory is limited. So, buffers should not contain redundant data (data that violates a `MISD` constraint) or data points gathered too frequently or too sparsely. Both scenarios provide data of little value to the application. The `collect` constraint allows developers to simply gather useful windows of data, coming out of a task.

These three constrains provide the core ways developers interact with sensing data inside their programs. These constraints provide a way for developers to declare what data is most valuable in a structured way, that masks the intermittent operation of batteryless sensors. We anticipate there will be more constraints added (or current constraints extended) as the language develops and new needs arise.

5.2.4 Ancillary Language Details

5.2.4.1 Multiple Task Graphs

Mayfly programs can be made up of multiple task graphs, each of which comprises multiple dependent tasks. These can all be defined in the same program. With multiple task graphs, developers can describe different sequences of actions to take during operation, under differing parameters. Each task graph has an implicit priority assigned to it, based on the ordering of the task graphs flows in the input file containing the source code. This means that the highest priority task graph will be checked first for any possible tasks to execute, if non are found, the next highest priority task graph is checked. To ensure that low priority task graphs are executed, developers need to apply `misd` constraints to the highest priority task graph, to leave time and energy for the other tasks graphs.

5.2.4.2 Memory Model

Mayfly prescribes a task-local memory model (similar to Chain[23]) Mayfly programs have no global memory that is accessible, or writable, from individual tasks. Tasks are only allowed to use volatile memory (local variables) internal to the task itself, as well as the read only inputs, which are explicitly defined by the program. Tasks generate output, which is accessible on the edges only once the task has completed. Since tasks cannot alter system or non-volatile memory, tasks will avoid consistency issues associated with mixed memory volatility systems. This task-local memory model also simplifies the programmer interface, as programmers only need concern themselves with the input and outputs of the task.

5.2.4.3 Hardware Interactions

The task-local memory model removes the possibility of memory inconsistency for the computational device (usually a microcontroller). However the memory and initialization state of the connected hardware peripherals such as sensors, radios, and storage devices, can cause problems. For example, any interactions a task has with external components will change those components in a non-deterministic way from the perspective of the next task to execute. To mitigate this issue, Mayfly developers must write tasks that build up, and break down hardware state, or simply reset hardware peripherals before use to reduce consistency errors.

5.3 Implementation

This section describes the Mayfly Runtime and implementation. We also describe the software and hardware used to evaluate, and deploy Mayfly, as well as applications developed with Mayfly. All software and hardware designs are freely available at our website.

5.3.1 Code Generation

Writing a new language from scratch is unnecessary, and potentially hurts adoption by the community. The Mayfly language is instead a coordination language built on Embedded-C, meaning developers can reuse common libraries and functions.

The Mayfly compiler goes through multiple phases to create a device specific firmware from a Mayfly program and user libraries. The architecture of the compiler is shown in Figure 5.7. The compiler is written in Java, the Java CUP library is used as a parser generator, with JFlex as scanner generator. A templating system based on Mustache templates is used to construct the Mayfly runtime instance for each program. Using templates makes porting to other processors and platforms easier. The compilation phases are described below.

Parse and Validate Task Graph: The Mayfly program code is parsed, and checked for syntax errors. The compiler checks that inputs of tasks match edge data, and that constraints, tasks, and policies are all defined. After validation, graphs of tasks are constructed.

Graph Annotation and Analysis: In this stage, task graphs are annotated with the constraints and policies given by the developer, and graph structure is checked that no cycles exist. These constraints are analyzed for logic errors or potential problems: for example, if the expiration of a source node is too short, or if an edge is trying to gather a prohibitive amount of data for the time period..

Runtime Generation: Once task graphs are validated, annotated, and analyzed, code generation can begin. Task and edge data structures are created from the Task Graph(s), and written to the runtime templates. This approach allows for flexibility in scheduling algorithms. At the end of this stage, a complete Embedded-C

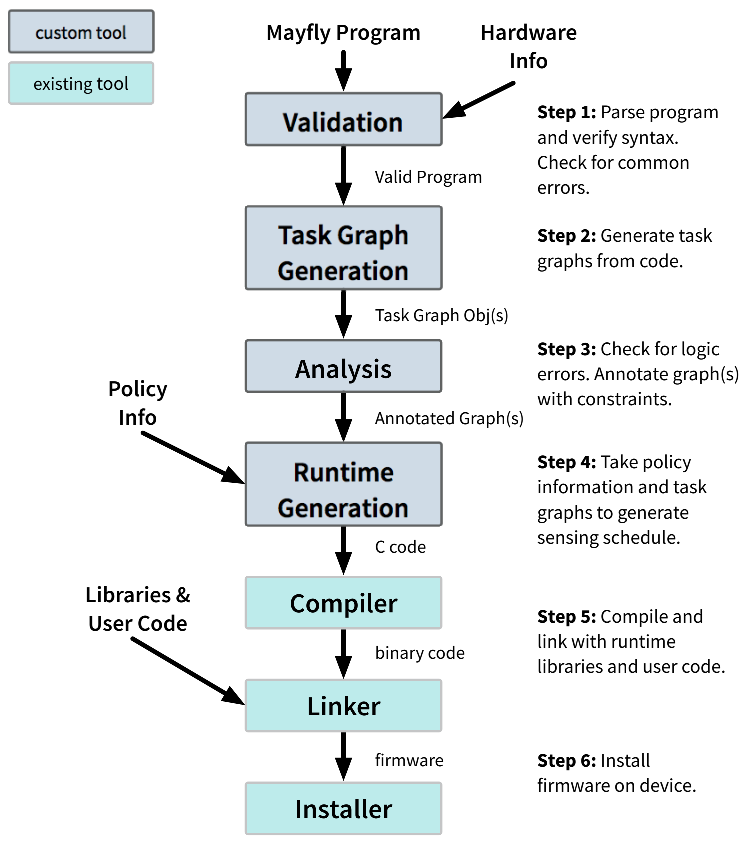

custom tool
existing tool
Mayfly Program
Hardware Info
Validation
Step 1: Parse program and verify syntax. Check for common errors.
Valid Program
Task Graph Generation
Step 2: Generate task graphs from code.
Task Graph Obj(s)
Analysis
Step 3: Check for logic errors. Annotate graph(s) with constraints.
Policy Info
Annotated Graph(s)
Runtime Generation
Step 4: Take policy information and task graphs to generate sensing schedule.
C code
Compiler
Libraries & User Code
Step 5: Compile and link with runtime libraries and user code.
binary code
Linker
firmware
Step 6: Install firmware on device.
Installer

Figure 5.7: Architecture of the Mayfly compiler, showing the steps in producing the firmware image for a given Mayfly enabled, batteryless sensing device.

program is generated.

Compile, Link, Install: The Embedded-C runtime is compiled, along with user code, hardware headers, and runtime libraries. This is all linked into a binary, and installed onto the batteryless sensing device.

5.3.2 Mayfly Runtime

The Mayfly Runtime is generated from the developers program and the language specification by the Mayfly compiler. The generated Embedded-C runtime is a statically defined schedule of tasks, with energy management, timekeeping, and checkpointing built-in. The schedule has to be static because of the extremely constrained resources of these devices. Energy is limited, so any energy used to execute runtime functions is taking away from potential user and sensing tasks. The Mayfly Runtime keeps track of three things through power failures: 1) local time, 2) the execution states in the progress thought the task graph(s), and 3) the data in each edge of each task graph. Using these three things, Mayfly's Runtime can execute tasks across power

LIST OF LISTINGS 5.2: Mayfly runtime operation pseudocode

```
1  main() {
2   time = get_time() // from RTC
3   if(timekeeper_reset())
4    rollback_full()  // Rollback all data
5
6   if(!state.atomic_lock)
7    rollback_one()  // Rollback last task
8
9   while(1)
10   t = next_task()
11   if(constraints_satisfied(t))
12     state.atomic_lock = FALSE
13     execute(t, input, output)
14     state.atomic_lock = TRUE
15     // User task ran
16     // Changes committed to task graph
17
18   if(t==NULL)
19     reset(t)
20     sleep() // Nothing todo
21 }
```

failures, while respecting temporal properties of sensor data. The runtime relies on architectural support in hardware for timekeeping and checkpointing to NVRAM, coupled with software techniques to persist tasks through failures.

Runtime Operation: Pseudo-code for the runtime operation is shown in Listing 5.2. After rebooting from a power failure (Line 1), the Mayfly runtime does three things (Lines 2-7), get the time, check if the external remanence timekeeper was reset, and check if the last task was completed. First, the Mayfly runtime updates the *local* system time using the external Remanence Timekeeper[54]. This timekeeper is an external capacitor or real-time-clock (RTC) with its own dedicated energy store—a small 1 µF capacitor. The timekeeper draws orders of magnitude less current than the microcontroller (MCU) while maintaining the clock, drawing less than 20 nA. This timekeeper might have reset if the time between a power outage and reboot was too long, in which case the runtime will rollback all time sensitive data as now it has no guarantees on the age of any previous data collected. This is preferable to continuing to process on useless (according to the developer constraints) data. The final check the runtime performs before executing tasks is determining if the last task executed was able to complete, and set the lock. If the lock is not set, then the tasks outgoing edges are rolled back, and the tasks is available to re-execute.

After successful recovery from a power failure, Mayfly begins looking for something to do. Tasks are checked in priority order, held in a static list defined at compile time. Task constraints are checked and if the constraints are satisfied, the task is executed. These constraints include all those listed in Section 5.3.2. Before the task is executed, the lock is released, and the output values from the incoming edges are placed into the task. Pointers to a temporary output buffer are also put into the task to receive any generated data. After a successful task execution, data is moved to the edges, the lock is set, and the program state is updated. This continues until either a power failure or a low voltage scenario, where nothing can be done, at which point the runtime puts the device to sleep until more energy becomes available (or the device dies), or a task becomes ready to be executed.

Data Management: Mayfly keeps all edge data in FRAM. FRAM is a low power non-volatile memory (NVRAM) with write speeds that allow it to be treated as RAM, enabling very fast checkpoints. Edge data is double buffered, so that old data is not overwritten by new data until the lock is set. This ensures that at any point, the execution can rollback to the previous task safely. Each edge also stores timestamps for each piece of data, array information for the `collect` constraint, and the last time the tasks was used to generate data for servicing the `misd` constraint.

Timekeeping: The runtime keeps track of time across power failures by using Remanence Timekeepers[54]. When energy runs out, the microcontroller, volatile RAM, and all clocks are reset. This means that any previous timestamps must be updated when power is regained. On each power-up, the runtime reads the timekeeper using either an ADC or the SPI bus (depending on the platform). After a read, the runtime charges a dedicated external small capacitor that maintains the Remanence Timekeeper. The key insight is that the remanence timekeeper will draw an order of magnitude less power than the the main sensing platform, and the draw is not dependent on sensor behavior. This means that it can maintain a granular sense of time throughout power failures.

5.3.3 Portability and Extensibility

The Mayfly runtime is intentionally made to be portable to other platforms, and easily extended. This will be crucial as platforms change often, and new technology renders hardware obsolete. The language parsing steps of the compiler are separate from the output step, and can be changed simply by editing template files, which are in the common Mustache format [79]. Mayfly programs can be compiled to another language besides C using the templating framework. These same templates can be modified to update, or change the scheduling mechanism. In this way emerging platform developers for batteryless-IoT devices can make use of the Mayfly language and runtime with only minor changes.

5.3.4 Applications

We implemented two complete applications with Mayfly. These applications demonstrate real problems that batteryless, energy harvesting systems can solve. Each application has time sensitive data generated or transformed by tasks that vary in their time and energy requirements. Each application encompasses multiple task nodes, and uses multiple Mayfly constraints to specify the program. These applications are variants of those presented in Chain [23], developed for the WISP [114] Computational RFID platform. We rewrite these applications using Mayfly, using with timing information gathered from domain insights that allow Mayfly to optimize data collection. We also cast these applications in the broader scope of energy harvesting platforms, beyond just RFID powered, in-place sensing, to extend to wearable computing and other domains.

Cold-Chain Equipment Monitoring (CEM): CEM systems continuously monitor temperature controlled environments, such as vaccine and biological sample storage. Temperature logging also has application in smart home technology and HVAC monitoring for commercial and industrial buildings. These logs could be read off the device at a later time using an RFID reader, physical access, or by broadcasting to a basestation. The CEM system implemented in Mayfly can safely assume that temperature will not change rapidly, meaning that the data generation rate can be throttled using the MISD constraint. Since the temperature is being logged, each data value has no expiration, but is tagged with a timestamp that persists through reboots.

Exercise Recognition: The health and wellness of a large aging population is a major concern in the USA. Exercises for elderly people, especially overweight elderly people, are often prescribed by doctors. Hip exercises are especially important, often including the sit to stand exercise, which helps prevent disability. Doctors do not have any data or confirmation that elderly patients are performing these exercises however, and if they did, better outcomes and decisions could be made to benefit the patient. Using a wrist-worn, batteryless wearable device equipped with an accelerometer is one way that these exercises could be tracked. By discarding the batteries, the wearable smaller, easier to wear, and does not have to be taken off to charge, which presents opportunities for losing the device. Activity Recognition (AR) can use the on-board accelerometer to determine sitting and standing states to tally exercise completion based on prior training. AR samples a sliding window, filters out noisy values, then extracts features and classifies as standing up or sitting down. Mayfly can take advantage of programmer timing knowledge to discern when old accelerometer data has expired, and is therefore not worth processing, or if it is too early after a standing or sitting action to gather new data (since it is physically impossible to sit or stand in a few milliseconds).

5.4 Evaluation

We evaluate Mayfly by examining the benefits of timekeeping when faced with intermittent power, in comparison to untimely languages. We make comparisons to other intermittent languages in terms of memory overhead, data utility, and usability, for a variety of real world applications outlined in Section 5.3. We introduce our experimental setup and metrics in Section 5.4.1, then outline the results of our experiments. We present execution overheads of the scheduler, as well as initialization costs of the runtime in Section 5.4.5. Finally, we investigate the usability of Mayfly and traditional Embedded-C for programming intermittent devices in a user study in Section 5.4.6.

5.4.1 Experimental Setup

Designing experiments for runtime systems for intermittent devices must be done with consideration of energy harvesting environment, leakage, measurement overhead, and available platforms. Because of low energy storage, measurement techniques must be non-invasive and energy free. In this section we describe the experimental design we use to compare each state-of-the-art runtime system with Mayfly.

Test Devices: The WISP [114] and Moo [136] are, to our knowledge, the only hardware platforms available for batteryless sensing. For evaluating Mayfly, we use a WISP augmented with a custom printed circuit board (PCB) that attaches to the connector, providing an RTC as remanence timekeeper for keeping time across power failures.

Runtime Systems: We compare implementations programs in Mayfly to implementations of the same or similar applications for DINO[81], and Chain[23]. The Chain artifact provided implementations of the CEM and activity recognition (exercise recognition) application in DINO, and Chain. In our experiments, we compare results for each application on every runtime system.

Measurement Setup: We used a number of tools to gather application success metrics, sensor data, and execution statistics without interfering with the execution of the devices under test. A Saleae logic analyzer and an Energy-Interference-Free-Debugger (EDB) [24] were used to record application success metrics by snooping a communication bus on the test platforms. Python scripts were used to coordinate the data collection, and start or stop test runs. An Impinj RF2500 Speedway RFID Reader was used as the energy source.

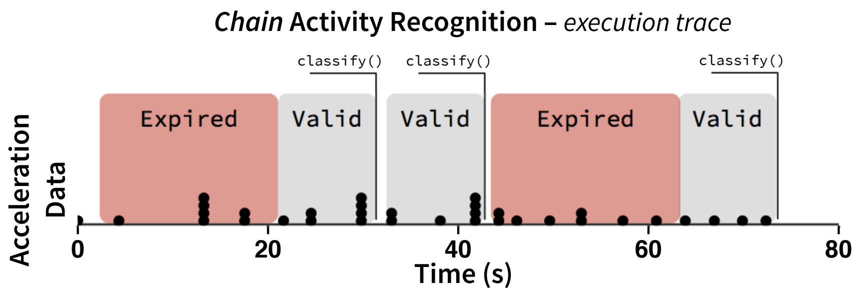


Figure 5.8: Forward progress is not always in the interest of application Quality of Service. Expired acceleration samples (blocked in red) are shown being used to recognize exercise activities using Chain.

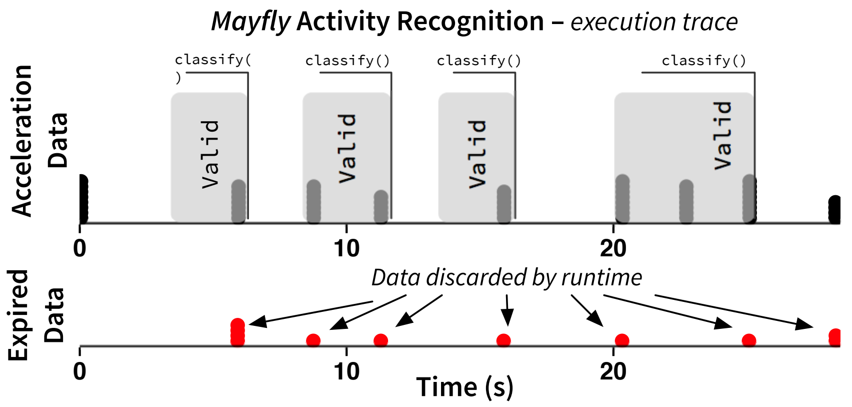


Figure 5.9: Mayfly discards, and replaces data (as specified by the developer) in its activity recognition. This shows the acceleration sample gathered in an actual execution of the Mayfly Exercise Recognition app.

5.4.2 Data Utility

Mayfly takes advantage of developer application knowledge and insights to deliver the same or better quality of service and data utility while doing less work. The exercise recognition application described in Section 5.3 was run multiple times on the RFID reader, with the Chain and Mayfly intermittent runtime systems. The WISP that the exercise recognition program was running on was placed 25 cm away from the mini guardrail antenna connected to the Impinj RFID reader. At that distance, harvestable energy is scarce and longer outages on the order of seconds are frequent. Figure 5.8 shows a representative trace of the Chain Exercise recognition app running through this experiment. Figure 5.9 shows a representative trace of the Mayfly Exercise recognition app running through this experiment.

Not all data are equal in determining the Quality of Service (QoS) of an application. These figures show that preserving forward progress without regard for elapsed time of power failures can hurt the quality of

Table 5.1: Non-volatile memory usage (KB) of each app and system.

App.	Mayfly	Chain	DINO
ER	2.9KB	2.5KB	4.2KB
CEM	3.1KB	4.1KB	5.8KB

service. Throughput does not linearly relate to QoS, which means that throughput can be traded off for energy without reducing quality of service for applications that have temporal data constraints. Figure 5.8 shows that Chain (and any other untimely runtime) will process old, often useless data, wasting cycles and energy. As shown in Figure 5.9, Mayfly discards old data using its external timekeeper, and only processes fresh data.

5.4.3 Memory Usage

Memory usage is important in embedded device in general, and batteryless sensors especially. These ultra constrained devices can't hold much data (the MSP430FR5969 on the WISP has 64KB of FRAM, 2KB of SRAM) so must be intelligent in their data management. We characterized the amount of memory used by each application implemented on each runtime. The memory usage of each application and its runtime implementation is shown in Table 5.1. Mayfly benefits from the absence of checkpointing, since developers specify which data is important, Mayfly only needs to store those data, not the entire stack. A mirrored memory space is not required, meaning that Mayfly will always have comparable or better memory usage to the Mementos and DINO approach. None of the approaches use a significant amount of memory in relation to the total memory on the WISP device. Mayfly benefits beyond other approaches in some cases since timing constraints controls the amount of data gathered and stored. Mayfly can get rid of excess data (that may have expired, or not be worth gathering) based on user defined constraints like `expires` and `MISD`.

A significant portion of the memory footprint of the Mayfly runtime is from libraries and platform initialization routines. Table 5.2 shows the breakdown of Mayfly runtime memory. Most of the memory footprint comes from the generated scheduler. Constraints, and task data are directly created from user specified constructs. The scheduler tradeoff a higher memory footprint in order to be more efficient and easier for the compiler to optimize. We anticipate that further improvements to the scheduler will reduce the memory footprint.

Table 5.2: Mayfly memory breakdown per application.

App.	Task data	Scheduler	Constraints
ER	406b	884b	476b
CEM	270b	1128b	636b

5.4.4 Developer Effort and Usability

For many applications, Mayfly is easier to program with than other systems, because of the (1) reduced number of language constructs that must be hand coded, and (2) the top-down, simple to visualize development approach. Table 5.3 shows the difference in lines of code required to develope an app in Chain and Mayfly. Mayfly was specifically designed to make the job of writing time aware applications for intermittently powered sensors easier. As we demonstrate in our user study (Section 5.4.6), developers have a hard time with understanding intermittent programming when using Embedded-C, even with a reliable, external timekeeper. Mayfly programs are designed top-down. Developers define the input and outputs of tasks, then make connections between tasks, then finally assign constraints to the tasks or edges. This is assisted by a visualization tool that shows a graphical representation of the Mayfly program on compilation that looks similar to Figure 5.5. Once the Mayfly program has been successfully created, developers only need to include a separate source file with the task definitions implemented. These function definitions can be ported from existing code, use existing libraries, and are written in Embedded-C like every other runtime system we evaluated.

Mayfly has the advantage over other runtime systems by separating the global goals (the Mayfly program defining task graphs) from the actual implementation of tasks in Embedded-C. Other systems join the two; Chain for example, requires programmers to explicitly specify memory channels and then specify control flow inside the task definitions themselves, using new C language constructs. While this approach can greatly reduce the memory footprint, it obscures control flow, requiring users to search through a program to find where tasks lead to. This is mainly because of the approach; Chain is designed as a C library, Mayfly is a compiler, allowing for much greater freedom and flexibility in the input language, and compiled output. Compiling gives greater control, allowing Mayfly to generate intelligible error messages, and capture the most common errors in the validation stages.

Table 5.3: **LOC for language constructs in Mayfly and Chain.**

	Mayfly			Chain			
App	*tasks*	*flow*	*constraints*	*tasks*	*ch*	*flow*	*decl*
ER	5	1	13	11	49	19	61
CEM	9	3	17	12	63	19	82

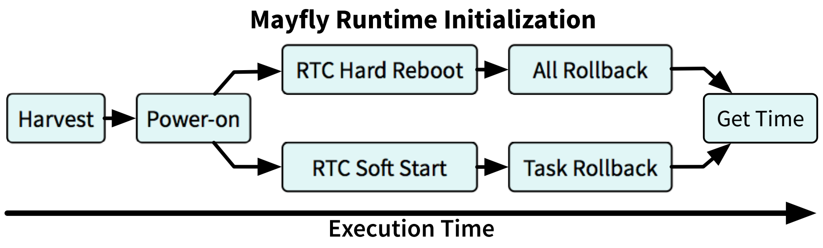


Figure 5.10: Runtime initialization flowchart. Each function has specific timing overhead. If a hard reboot happens where the Real-Time-Clock (RTC) is reset because of a long power failure, initialization cost increases.

5.4.5 Overhead

Certain implementation details are pertinent to the overall evaluation. We detail their effect on performance, note potential areas of improvement, and provide practical implementation costs in this section. Specifically we look at the runtime initialization costs (above the execution costs of user defined tasks), and the scheduling costs for determining which user defined task to run. The platform initialization scheme is shown in Figure 5.10, along with execution costs in Table 5.4.

In Figure 5.10, two paths are shown for initialization, the hard reset path, and the soft reset path. The soft reset path (“RTC Soft Start” in figure) simply polls the RTC, rolls back the last task if it was not completed and then gets the time. This happens fairly quickly. However, when the external timekeeper loses power completely (in addition to the MCU), the timekeeper resets its clock back to zero on next boot. This reboot (“RTC Hard Reboot” in figure) takes longer to initialize the timekeeper properly, than if it had not expired. Additionally, all timestamps (“All Rollback” in figure) must be set back to zero for each task output. In our current implementation for the MSP430FR5969 this hard reboot can take up to one second. This length of time is significant, and difficult to overcome with current RTC components available off-the-shelf, however, in our experiments the timer rarely loses power completely, as we used a conservatively sized storage capacitor of 10 µF for the timekeeper, which can time more than 17 minutes of of failure. This 17 minute timing ability is more than adequate for our applications, however, for applications with longer timing requirements, new

Table 5.4: **Initialization and scheduler runtime costs.**

Init function	**Time**
Power-on from brown-out	1000 µs
RTC hard reboot	708.5 µs
RTC soft start	144.6 µs
Rollback all Data	23.6 µs
Single task rollback	4.6 µs
Get time (seconds, minutes, hours)	246.4 µs
Get time (seconds)	80.6 µs
Scheduler function	
Process constraints for single task	4.5 µs
Scheduler cant find task to execute	56.3 µs
Task finished, commit results	7.0 µs

advances in zero power timekeeping are required.

Table 5.4 shows the costs of the scheduler functions. Before a task can be executed, its constraints must be satisfied, this check happens many times during the scheduling cycle and must be very fast. On average, this check happens in 4.5 µs, quick enough to not be a burden, and making the cost of not finding anything to do low. The other important function is committing data from a task to non-volatile memory so it can be preserved through a power failure. This function only takes 7.0 µs.

Energy and Cost: To use Mayfly with the WISP, it must be augmented with the custom PCB. The energy cost of maintaining the timekeeper, the price increase per unit, and the firmware are all overhead items for Mayfly. The initial charging of the timekeeper requires 28.8 µJ, then a constant trickle current of 54 nA. Adding the custom PCB, or designing a new PCB with required hardware only increases the price point by $1.05 per unit. Additionally, the memory overhead of supporting the hardware timekeeper is constrained to a small library that requires 1364 bytes.

5.4.6 User Study

We evaluated the usability of Mayfly on 11 participant drawn from a junior-level, university Computer Operating Systems course[2] Our findings show 1) Mayfly reduces the time needed to write intermittent programs, 2) Mayfly helps developers reason about intermittent behavior, and 3) the benefit of using Mayfly is high enough for encourage C developers to migrate.

Methodology: Participants were provided documentation describing methods for writing intermittent pro-

[2]This study was approved by our Institutional Review Board.

grams in embedded-C, relevant syntax, and function definitions provided by the research team for sensing and timekeeping. They were given 20 minutes to familiarize themselves, prior to the experiment. Participants were then provided 3 unique programming challenges, with 20 minutes to complete each. The research team manually compiled participants' code, and reported errors in syntax and timing related bugs. Participants repeated the above process for the same challenges, but this time using the Mayfly language. Participants received documentation describing methods for writing intermittent programs in Mayfly, relevant syntax, and function definitions, as above. Participants used Mayfly to complete the same 3 programming challenges, under the same conditions. The programming challenges were modeled after the three Mayfly timing constraints, `expires`, `misd`, and `collect`. After each set of 3 challenges, participants completed a survey rating the ease of the language used.

Sample: Our 11 participants ranged in class standing from junior to senior, with 3 to 7 years of formal computing education, and 3 to 8 years of total programming experience. Participants self-rated their overall programming abilities, comfort using C, and knowledge of computer architecture, as compared to other students, and average application developers. Overall, our participants' abilities were typical, with reported skills' means ranging from 2.55 to 3.64, where 3 indicated average ability.

Results: Mayfly reduces time needed to write intermittent programs. Participants using Mayfly unsuccessfully compiled an average of 0.64 fewer times per task, than users of C. On average, participants successfully completed 1.54 more challenges in Mayfly than in C. Mayfly makes overcoming the complexity of intermittent programs easier. None of our participants successfully completed our `expires` coding challenge in C, while all 11 completed the challenge in Mayfly. Five participants completed our `misd` challenge in C, while 7 did so in Mayfly. Four participants successfully completed our `collect` challenge in C, while 8 did so in Mayfly.

We believe our results show the benefit of using Mayfly is high enough to encourage developers using C to migrate. Using a series of 5-point Likert scales, participants rated the usability of each language. The items in this survey were developed using confirmatory factor analysis, to ensure validity. We summed and divided these ratings by the total possible, to create a percent language usability score for both C and Mayfly. Nine of our 11 participants reported C was their preferred programming language, prior to this study, and 7 participants indicated their C programming abilities are above average compared to other students at their university. However, working with Mayfly, 10 of 11 users rated its usability higher than C, for completing the assigned challenges. Additionally, Mayfly received a 23% higher mean score than C, on our usability survey.

5.5 Related Work

In this section we place the Mayfly language and runtime in the literature, and demonstrate the novelty of our approach.

Preserving Forward Progress: Checkpointing systems like Mementos [107], Hibernus [5], QuickRecall [59], and others[90, 9] have emerged to keep forward progress on intermittently-powered systems. Other solutions, like DINO [82], show that even with checkpointing, memory consistency is not maintained. Chain [23] provides a new model that eliminates the cost of checkpointing large parts of volatile memory, and appears to provide better programmability (although this was not evaluated with a user study). None of these solutions consider how the loss of timekeeping affects the duty cycle, and how the utility of sensed and computed data changes over time. The proposed Mayfly Runtime builds off DINO an previous checkpointing and scheduling libraries to preserve forward progress *and* manage the temporal aspect of sensing tasks.

Operating Systems & Runtimes Most operating systems for wireless sensor networks have assumed a stable power supply, and were not built for intermittent programs. However, recent advances with computational RFID have pushed for batteryless task management. Dewdrop[16] is an energy-aware runtime for Computational RFID tags like the WISP. It schedules tasks based on available energy, and attempts to get as many tasks done as possible. DEOS[141] is similar to Dewdrop as a lightweight energy aware scheduler. QuarkOS[138] is a low overhead operating system that divides every communication, sensing, and computation task into tiny fragments. It then sleeps between execution of these fragments to recharge. BY doing this, QuarkOS allows tasks to be executed on extremely small energy budgets. EnOS[129] is a kernel for energy-neutral systems. Energy-neutral systems rely on battery backups but function almost exclusively off harvested energy. EnOS allows for tasks to be organized into different criticality levels, helping manage blackouts. However, EnOS assumes a mostly stable supply, and does not consider the effects of time. In fact, all of the operating systems and runtimes mentioned do not consider the temporal aspects of sensor data.

Languages Mayfly is the first language designed for batteryless sensors that captures the temporal constraints associated with sensor data. NesC [42] was the first widely adopted language for wireless embedded sensors. Using an event driven model, NesC is an extension of C, and is the language overlaying TinyOS. Flask[83] is a domain specific language that is a subset of Haskell. Flask provides the power of functional programming to sensor networks. Numerous other languages have been created[92, 38, 80], most are closely related to or built on top of TinyOS and NesC, which assumes a stable power supply.

Eon[119] was the first programming language for sensors that was built to be energy-aware. Eon programmers used a declarative coordination graph to sequence and categorize tasks in terms of energy states. These tasks were then executed based on the energy available and dependencies of the task. Mayfly is closely related to Eon, using declarative coordination graphs as well as Eon, but with the intent of defining timing constraints and task atomicity.

Hardware Approaches: Other related approaches come from the hardware languages design field. Idetic[91] is a set of synthesis mechanisms for enabling long term computations on ASICs powered completely by energy harvesting. Idetic determines optimum checkpoint placement by parsing an intermediate graph language using synthesis techniques. Chlorophyll[98] similarly used syntax guided synthesis techniques, but for the purpose of programming ultra low power FPGAs. This approach increased both usability and confidence in solution. Both of these techniques show promise for synthesis with low power and intermittent systems. However, they are not directly translatable to current off-the-shelf sensor technologies, nor do they provide common sensing interfaces. Potentially, a Mayfly like programming model could take advantage of the native checkpoints provided by these systems, creating an even more robust batteryless sensor.

5.6 Discussion and Future Work

In this section we discuss the limitations of the Mayfly language, and runtime, and outline potential thrusts for future work.

5.6.1 Limitations and Challenges

The Mayfly language and runtime has some key limitations. Most importantly, the scheduling will rarely be optimal, as the energy constraints of the environment determine what is possible—which can often be nothing. This is made more difficult because the runtime only has access to a one-dimensional view of the current energy, and harvesting efficiency of the sensor platform (the voltage on the storage capacitors). Another limitation stems from user code organization. If user task are not made small enough, or overestimate the energy availability of the environment, tasks will fail much more often. This is difficult for Mayfly to anticipate in it's current form. Another difficulty comes from the lack of a general hardware platform for batteryless devices. Current platforms are generally application specific, or energy harvester specific. This makes it difficult for the runtime to make assumptions that could improve the scheduling efficiency.

5.6.2 Future Work

Mayfly is just the beginning step towards making batteryless sensing a mainstream sensor medium. We envision future work investigating more intelligent, and dynamic task scheduling with time sensitive data stream, generalized hardware platforms for many applications, and more sophisticated tooling to aid the amateur and expert developers of batteryless applications.

Dynamic Scheduling: Currently the runtime schedule of tasks is generated at compile time. While this achieves a low overhead and quick runtime functions, it is imagined that the scheduling could integrate contextual information about the energy state of the sensor, the previous energy costs of the current task, and other information to make better scheduling decisions on the fly.

Hardware Platforms: As mentioned, the lack of a general hardware platform makes decision making more difficult at runtime. Batteryless sensing needs a general platform, that is harvester independent, and provides the energy management and persistent timekeeping faculties outlined in this paper.

Tooling: Without more sophisticated debugging, simulation, and evaluatory tools, batteryless sensing will never make it to the mainstream of computing. We envision the Mayfly language and runtime integrating with current tools like EDB and Ekho. We also see vast potential for future work in automated unit testing of Mayfly programs using these tool.

5.7 Conclusions

The Mayfly language and runtime was created in response to the lack of batteryless platforms that 1) considered the effect of time on sensor data value and 2) the lack of intuitive programming models. The Mayfly language is a graph based, declarative programming model that allows developers to focus on application goals, outline timing requirements for data collection, and list task dependencies. The Mayfly runtime takes these tasks and constraints, and creates a schedule at compile time that can be executed on the sensor node. This scheduler manages timekeeping across power failures, and uses Federated Energy to handle tasks energy requirements. Devices programmed with Mayfly were deployed on RFID powered WISPs for activity monitoring, and for temperature logging. Additionally, a user study was run that revealed that participants had trouble visualizing program execution and keeping sensor data from expiring when programming in traditional Embedded-C, and were able to reason about tasks better with the Mayfly language, accomplishing more in the same amount of time, and having a more enjoyable experience.

Batteryless sensors are an indispensable part of the future of the Internet-of-Things. These devices promise to revolutionize sensing, and even computing. We view Mayfly as a positive step towards the manifestation of this vision.

CHAPTER 6

CONCLUSIONS AND DISCUSSION

Batteryless sensing is fundamentally different from conventional computing. These devices challenge one of the most basic assumptions of computing–a stable power supply. Energy harvesting is inconsistent and energy storage is scarce, so batteryless sensors *will* lose power intermittently. Programmers must figure out how to tie together fragmented execution opportunities to get useful work done each time a sensor is resurrected.

Batteryless, energy harvesting sensing presents critical challenges not encountered by tethered or battery- powered sensing. First, batteryless devices operate in unpredictable environments, where voltages vary and power failures can occur at any time—often devices are in failure for hours. Second, a devices behavior effects the amount of energy they can harvest—meaning small changes in tasks can drastically change harvester efficiency. Third, the programming interfaces of batteryless device are ill-defined and non- intuitive; most developers have trouble anticipating the problems inherent with an intermittent power supply. Finally, the platform and community support does not exist in a coherent standard, reducing usability and adoption.

This dissertation on batteryless sensing has touched on multiple layers of the system stack to address these problems. Specifically this work has 1) developed tools for repeatable experimentation with batteryless computers, 2) designed a hardware platform to simplify task scheduling, protect against failures, and support rapid device prototyping in a usable manner and 3) created a language and runtime that captures the temporal attributes of sensor data on intermittently executing sensing platforms. Each of these pieces has simplified the development process; enabling more robust, long lived computation, on sustainable, energy harvesting sensors.

6.1 Themes

This dissertation has focused on fundamental questions of *sustainable computing*, and *energy efficiency*. This work, the methods, the systems, and the results, will enable sustainable, responsible computing, laying the groundwork for the future of sensing. In this future devices are small, cheap, and are useful for the entire lifetime of the things they monitor. These devices will be easy to program, straightforward to test, and will be deployed confidently. This future will push forward the boundaries of sensing, with a focus on relieving the cognitive burden on programmers to the point where even amateurs can confidently program, test, and deploy, batteryless sensors.

6.2 Future Work

This dissertation work has laid the groundwork for what is a rich and exciting new field of study. Long term future directions can seek to develop the ideas around sustainable computing and sensing, advancing the vision of the Internet-of-Things for the benefit of science and society. In each of the following future directions, building and deploying real systems for emerging applications will be the focusand the main form of establishing scientific validity. The following research questions are core to the future of sustainable, batteryless sensing and computing, each has significant and exciting unsolved problems.

Intermittent Toolchain: The batteryless sensing toolchain has matured to the point that developers have simulation, emulation, and some debugging tools. However, the cognitive burden of designing and testing applications that effectively handle intermittent power is still high. The toolchain lacks definitive tools such as testbeds, network simulators, and general hardware platforms. Execution on batteryless devices depends on random environmental factors, causing operation to be probabilistic. In spite of this, developers need to better understand how the code they write will execute in deployment. There are two potential tools that can facilitate user understanding and confidence in deployment. First, the idea of energy aware code coverage–meaning developers can automate the generation of energy environments that will exercise the full program functionality. Second, the idea of a hardware debugging tool that completely reproduces the energy environment, data traces, and allows non-invasive step debugging and tracing of batteryless sensors. Additionally the development of generalized hardware platforms with built in Ekho support is fundamental to continuing success. Backing this, would be a web based infrastructure that brings together thousands of energy harvesting environments recorded by researchers in the field using Ekho, shared across institutions and research labs.

Intermittent Networks: With the Flicker platform now available, the potential for networking research has dramatically increased. Multi-hop batteryless networks do not currently exist because of the incredible difficulties of coordinating and syncing multiple intermittently powered devices. These networks would be of significant value to the Internet-of-Things. Much future work lies in developing a dedicated batteryless sensing testbed that integrates Flicker and Ekho. This testbed would be open to the scientific community, where researchers could try out different approaches and protocols geared towards multi-hop intermittent networks.

Applications: Deploying new and exciting applications for batteryless computing keeps this research relevant and grounded. Expanding the realms of battery-free sensing past RFID applications, to include wearables, occupancy sensing, wildlife tracking, greenhouse monitoring, and infrastructure support are necessary. These new applications necessitate collaboration with domain scientists in other fields such as health and biology. This will create a positive dialog about the future of sensing, and further inform this work work. In the short term, plans to adapt this work for use on mobile health wearable platforms such as the Amulet [50] are ongoing. These devices could support battery free operation for health and fitness applications like sleep monitoring, and step counting.

6.3 Final Thoughts

In summary, battery-free sensing promises to revolutionize science and society. This dissertation has only scratched the surface of this deeply challenging, deeply interesting new field. Future research will continue to push the boundaries of sensing, and attempt to tackle some of the most pressing problems of the Internet-of-Things. Work on responsible, sustainable sensing has the potential for positive impacts and collaborations with many areas, including health services and patient care, commercial and consumer applications, wildlife conservation, industrial and infrastructure management and monitoring, and many other fields. No longer will the Internet-of-Things be a concept of the future, with this dissertation, now the Internet-of-Things will be more than just hype.

BIBLIOGRAPHY

[1] Abracon Corporation ab08x5 real-time clock family. http://abracon.com/Precisiontiming/AB08X5-RTC.PDF, 2015. Last Viewed December 10, 2015.

[2] M. H. Alizai, Q. Raza, Y. Chandio, A. A. Syed, and T. M. Jadoon. Simulating intermittently powered embedded networks. 2016.

[3] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, et al. Numerical python, 2001.

[4] G. Avoine, K. Kalach, and J.-J. Quisquater. ePassport: Securing international contacts with contactless chips. In G. Tsudik, editor, *Financial Cryptography and Data Security*, pages 141–155. Springer-Verlag, 2008.

[5] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embedded Systems Letters, IEEE*, 7(1):15–18, 2015.

[6] S. Bandyopadhyay and A. P. Chandrakasan. Platform architecture for solar, thermal, and vibration energy combining with mppt and single inductor. *IEEE Journal of Solid-State Circuits*, 47(9):2199–2215, 2012.

[7] A. Bangor, P. T. Kortum, and J. T. Miller. An empirical evaluation of the system usability scale. *International Journal of HumanComputer Interaction*, 24(6):574–594, 2008. DOI 10.1080/10447310802205776.

[8] D. Benedetti, C. Petrioli, and D. Spenza. Greencastalia: an energy-harvesting-enabled framework for the castalia simulator. In *Proceedings of the 1st International Workshop on Energy Neutral Sensing Systems*, page 7. ACM, 2013.

[9] N. Bhatti and L. Mottola. Efficient state retention for transiently-powered embedded sensing. In *Proceedings of the 13th ACM International Conference on Embedded Wireless Systems and Networks (EWSN) Graz (Austria)*, 2016.

[10] S. Bobovych, N. Banerjee, R. Robucci, J. P. Parkerson, J. Schmandt, and C. Patel. Sunaplayer: High-accuracy emulation of solar cells. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 59–70, New York, NY, USA, 2015. ACM. DOI 10.1145/2737095.2737110.

[11] Bosch. Xdk cross domain development kit. http://xdk.bosch-connectivity.com/, October 2016.

[12] J. Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[13] I. Buchmann et al. *Batteries in a portable world*. Cadex Electronics Richmond, 2001.

[14] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An Energy-Aware Runtime for Computational RFID. In *Proc. 8th USENIX Conf. Networked Systems Design and Implementation (NSDI'11)*, pages 197–210, Boston, MA, USA, Mar. 2011. ACM.

[15] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An Energy-Aware Runtime for Computational RFID. In *Proc. 8th USENIX Conf. Networked Systems Design and Implementation (NSDI'11)*, pages 197–210, Boston, MA, USA, Mar. 2011. ACM.

[16] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[17] B. Campbell and P. Dutta. An energy-harvesting sensor architecture and toolkit for building monitoring and event detection. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pages 100–109. ACM, 2014.

[18] B. Campbell, B. Ghena, and P. Dutta. Energy-harvesting thermoelectric sensing for unobtrusive water and appliance metering. In *Proceedings of the 2Nd International Workshop on Energy Neutral Sensing Systems*, ENSsys '14, pages 7–12, New York, NY, USA, 2014. ACM. DOI 10.1145/2675683.2675692.

[19] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher. Virtual Battery: An Energy Reserve Abstraction for Embedded Sensor Networks. In *Proc. 29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 1–11, Barcelona, Spain, Nov.–Dec. 2008. IEEE.

[20] J. A. Carr, J. C. Balda, and H. A. Mantooth. A Survey of Systems to Integrate Distributed Energy Resources and Energy Storage on the Utility Grid). In *Proc. IEEE Energy 2030 Conf. (ENERGY'08)*, pages 1–7, Atlanta, GA, USA, Nov. 2008. IEEE.

[21] G. Chen, M. Fojtik, D. Kim, D. Fick, J. Park, M. Seok, M.-T. Chen, Z. Foo, D. Sylvester, and D. Blaauw. Millimeter-scale nearly perpetual sensor system with stacked battery and solar cells. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2010. DOI 10.1109/ISSCC.2010.5433921.

[22] P. H. Chou, C. Park, J. Park, K. Pham, and J. Liu. B#: a battery emulator and power profiling instrument. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 288–293. ACM, 2003.

[23] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 514–530, New York, NY, USA, 2016. ACM. DOI 10.1145/2983990.2983995.

[24] A. Colin, A. P. Sample, and B. Lucia. Energy-interference-free system and toolchain support for energy-harvesting devices. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '15, pages 35–36, Piscataway, NJ, USA, 2015. IEEE Press. Online at http://dl.acm.org/citation.cfm?id=2830689.2830695.

[25] M. Computing. Usb-201 data acquisition usb daq device 12-bit, 100 ks/s. http://www.mccdaq.com/usb-data-acquisition/USB-201.aspx. [Online; accessed 30 March, 2015].

[26] A. Corporation. AVR XMEGA microcontrollers. http://www.atmel.com/products/microcontrollers/avr/avr_xmega.aspx, October 2013. [Online; accessed 11 October, 2013].

[27] R. Dall'Ora, U. Raza, D. Brunelli, and G. P. Picco. Senseh: From simulation to deployment of energy harvesting wireless sensor networks. In *Local Computer Networks Workshops (LCN Workshops), 2014 IEEE 39th Conference on*, pages 566–573. IEEE, 2014.

[28] R. Dall'Ora, U. Raza, D. Brunelli, and G. P. Picco. SensEH: From Simulation to Deployment of Energy Harvesting Wireless Sensor Networks. In *Proceedings of the 9th IEEE Workshop on Practical Issues in Building Sensor Network Applications (SenseApp'14)*, pages 566–573, Edmonton, Canada, Sept. 2014. IEEE.

[29] S. DeBruin, B. Ghena, Y.-S. Kuo, and P. Dutta. Powerblade: A low-profile, true-power, plug-through energy meter. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 17–29. ACM, 2015.

[30] C. Donovan, A. Dewan, D. Heo, and H. Beyenal. Batteryless, wireless sensor powered by a sediment microbial fuel cell. *Environmental science & technology*, 42(22):8591–8596, 2008.

[31] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors (EmNets-I)*, Tampa, FL, USA, Nov. 2004. IEEE Computer Society.

[32] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based On-line Energy Estimation for Sensor Nodes. In *Proceedings of the 4th Workshop on Embedded Networked Sensors (EmNets'07)*, pages 28–32, Cork, Ireland, Jun. 2007. ACM.

[33] P. Dutta, J. Taneja, J. Jeong, X. Jiang, and D. Culler. A building block approach to sensornet systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 267–280. ACM, 2008.

[34] EnOcean. Enocean: Self powered iot. https://www.enocean.com/en/, October 2016.

[35] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. MSPsim—an extensible simulator for MSP430-equipped sensor boards. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*. Springer, 2007.

[36] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. Cooja/mspsim: Interoperability testing for wireless sensor networks. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27:1–27:7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). DOI 10.4108/ICST.SIMUTOOLS2009.5637.

[37] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools'09)*, Rome, Italy, Mar. 2009. ACM.

[38] R. Flury and R. Wattenhofer. Slotted programming for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 24–34, New York, NY, USA, 2010. ACM. DOI 10.1145/1791212.1791216.

[39] M. Furlong, J. Hester, K. Storer, and J. Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, pages 23–26, New York, NY, USA, 2016. ACM. DOI 10.1145/2996884.2996889.

[40] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library: Reference Manual*. Network Theory Ltd., Feb. 2003. Online at http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0954161734.

[41] S. Ganeriwal, S. Čapkun, C.-C. Han, and M. B. Srivastava. Secure time synchronization service for sensor networks. In *Proceedings of the 4th ACM Workshop on Wireless Security*, WiSe '05, pages 97–106, 2005.

[42] 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 *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI'03)*, pages 1–11, San Diego, CA, USA, Jun. 2003. ACM.

[43] M. Gorlatova, P. Kinget, I. Kymissis, D. Rubenstein, X. Wang, and G. Zussman. Energy Harvesting Active Networked Tags (EnHANTs) for Ubiquitous Object Networking. *IEEE Wireless Communications*, pages 18–25, Dec. 2010.

[44] M. Gorlatova, R. Margolies, J. Sarik, G. Stanje, J. Zhu, B. Vigraham, M. Szczodrak, L. Carloni, P. Kinget, I. Kymissis, and G. Zussman. Energy Harvesting Active Networked Tags (EnHANTs): Prototyping and Experimentation. Technical Report 2012-07-27, Electrical Engineering, Columbia University, New York, NY, USA, Jul. 2012.

[45] M. Gorlatova, R. Margolies, J. Sarik, G. Stanje, J. Zhu, B. Vigraham, M. Szczodrak, L. Carloni, P. Kinget, I. Kymissis, and G. Zussman. Prototyping Energy Harvesting Active Networked Tags (EnHANTs). In *Proc. 32nd IEEE Int'l Conf. Computer Communications (INFOCOM'13)*, pages 585–589, Turin, Italy, Apr. 2013. IEEE.

[46] M. Gorlatova, J. Sarik, G. Grebla, M. Cong, I. Kymissis, and G. Zussman. Movers and shakers: Kinetic energy harvesting for the internet of things. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 407–419. ACM, 2014.

[47] 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. ACM, 2010.

[48] 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*, pages 195–208. ACM, 2010.

[49] J. Gummeson, S. S. Clark, K. Fu, and D. Ganesan. On the Limits of Effective Hybrid Micro-Energy Harvesting on Mobile CRFID Sensors. In *Proc. 8th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys'10)*, pages 195–208, San Francisco, CA, USA, Jun. 2010. ACM.

[50] J. Hester, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, K. Freeman, S. Lord, R. Halter, D. Kotz, and J. Sorber. Amulet: An energy-efficient, multi-application wearable platform. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 216–229, New York, NY, USA, 2016. ACM. DOI 10.1145/2994551.2994554.

[51] 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. ACM. DOI 10.1145/2668332.2668336.

[52] J. Hester, T. Scott, and J. Sorber. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors. In *Proc. 12th ACM Conf. Embedded Network Sensor Systems (SenSys'14)*, pages 1–15, Memphis, TN, USA, Nov. 2014. ACM.

[53] J. Hester, L. Sitanayah, and J. Sorber. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proc. 13th ACM Conf. Embedded Network Sensor Systems (SenSys'15)*, Seoul, Korea, Nov. 2015. ACM.

[54] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, and J. Sorber. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4), 2016.

[55] T. S. Heydt-Benjamin, D. V. Bailey, K. Fu, A. Juels, and T. OHare. Vulnerabilities in first-generation RFID-enabled credit cards. In *Proceedings of Eleventh International Conference on Financial Cryptography and Data Security, Lecture Notes in Computer Science, Vol. 4886*, pages 2–14, February 2007.

[56] N. Instruments. Ni x series multifunction data acquisition. http://sine.ni.com/ds/app/doc/p/id/ds-163/lang/en, October 2012. [Online; accessed 11 October, 2013].

[57] T. Instruments. MSP430FRxx FRAM Microcontrollers. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430frxx_fram/overview.page. Accessed: 2015-10-13.

[58] T. Instruments. http://www.ti.com/tool/ez430-rf2500, October 2013. [Online; accessed 11 October, 2013].

[59] 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 *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pages 330–335. IEEE, 2014.

[60] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, 2007. Online at http://portal.acm.org/citation.cfm?id=1236386.

[61] X. Jiang, J. Polastre, and D. Culler. Perpetual Environmentally Powered Sensor Networks. In *Proc. 4th Int'l Symp. Information Processing in Sensor Networks (IPSN'05)*, Los Angeles, CA, USA, Apr. 2005. ACM.

[62] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. Online at http://www.scipy.org/.

[63] A. Juels. Minimalist cryptography for low-cost RFID tags (extended abstract). In C. Blundo and S. Cimato, editors, *Security in Communication Networks*, volume 3352 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2005.

[64] A. Juels. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communications*, 24(2):381–394, February 2006. DOI 10.1109/JSAC.2005.861395.

[65] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for &ldquo;smart dust&rdquo;. 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. ACM. DOI 10.1145/313451.313558.

[66] D. H. P. Kang, M. Chen, and O. A. Ogunseitan. Potential environmental and human health impacts of rechargeable lithium batteries in electronic waste. *Environmental science & technology*, 47(10):5495–5503, 2013.

[67] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems*, 2006.

[68] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava. Power Management in Energy Harvesting Sensor Networks. *ACM Trans. Embedded Computing Systems (TECS)*, 6(4), Sept. 2007.

[69] A. E. Kouche, H. S. Hassanein, and K. Obaia. Wsn platform plug-and-play (pnp) customization. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6, April 2014. DOI 10.1109/ISSNIP.2014.6827642.

[70] Y.-S. Kuo, S. Verma, T. Schmid, and P. Dutta. Hijacking power and bandwidth from the mobile phone's audio interface. In *Proceedings of the 1st Annual Symposium on Computing for Development (DEV)*, 2010.

[71] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[72] D. Larcher and J.-M. Tarascon. Towards greener and more sustainable batteries for electrical energy storage. *Nat Chem*, 7(1):19–29, 01 2015. Online at http://dx.doi.org/10.1038/nchem.2085.

[73] E. A. Lee, B. Hartmann, J. Kubiatowicz, T. S. Rosing, J. Wawrzynek, D. Wessel, J. M. Rabaey, K. Pister, A. L. Sangiovanni-Vincentelli, S. A. Seshia, et al. The swarm at the edge of the cloud. *IEEE Design & Test*, 31(3):8–20, 2014.

[74] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 126–137, Los Angeles, CA, USA, Nov. 2003. ACM.

[75] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Wireless Sensor Networks. In W. Weber, J. M. Rabaey, and E. Aarts, editors, *Ambient Intelligence*, volume II, pages 115–148. Springer Berlin Heidelberg, 2005.

[76] D.-J. Li and P. H. Chou. Maximizing efficiency of solar-powered systems by load matching. In *Low Power Electronics and Design, 2004. ISLPED'04. Proceedings of the 2004 International Symposium on*, pages 162–167. IEEE, 2004.

[77] K. Lin, J. Hsu, S. Zahedi, D. C. Lee, J. Friedman, A. Kansal, V. Raghunathan, and M. B. Srivastava. Heliomote: Enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of ACM Sensys*, 2005.

[78] Y. Lin, D. M. Sylvester, and D. T. Blaauw. A sub-pW timer using gate leakage for ultra low-power sub-Hz monitoring systems. *Custom Integrated Circuits Conference*, 2007.

[79] Mustache Logic-less templates. http://mustache.github.io/, 2017. Last Viewed March 22, 2017.

[80] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 211–224, New York, NY, USA, 2008. ACM. DOI 10.1145/1460412.1460434.

[81] 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 '15, pages 575–585, New York, NY, USA, 2015. ACM. DOI 10.1145/2737924.2737978.

[82] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 575–585, New York, NY, USA, 2015. ACM. DOI 10.1145/2737924.2737978.

[83] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, volume 43, pages 335–346. ACM, 2008.

[84] W. Mao. Timed-release cryptography. In *Selected Areas in Cryptography VIII (SAC'01*, pages 342–357. Prentice Hall, 2001.

[85] R. Margolies, M. Gorlatova, J. Sarik, G. Stanje, J. Zhu, P. Miller, M. Szczodrak, B. Vigraham, L. Carloni, P. Kinget, et al. Energy-harvesting active networked tags (enhants): Prototyping and experimentation. *ACM Transactions on Sensor Networks (TOSN)*, 11(4):62, 2015.

[86] S. Meninger, J. O. Mur-Miranda, R. Amirtharajah, A. Chandrakasan, and J. H. Lang. Vibration-to-electric energy conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1), 2001.

[87] G. V. Merrett, N. M. White, N. R. Harris, and B. M. Al-Hashimi. Energy-aware simulation for wireless sensor networks. In *Proceedings of the 6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'09)*, pages 1–8. IEEE, 2009.

[88] Microchip. 7/8-bit single/dual spi digital pot with volatile memory. http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Components/General%20IC/22060b.pdf, October 2013. [Online; accessed 11 October, 2013].

[89] M. Minami, T. Morito, and H. Morikawa. Biscuit: A battery-less wireless sensor network system for environmental monitoring applications. In *In Proc. of the 2nd International Workshop on Networked Sensing Systems*. Citeseer, 2005.

[90] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Automated checkpointing for enabling intensive applications on energy harvesting devices. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pages 27–32. IEEE Press, 2013.

[91] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 216–224. IEEE, 2013.

[92] L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, Apr. 2011. DOI 10.1145/1922649.1922656.

[93] NXP Semiconductors SPI real time clock/calendar. http://www.nxp.com/documents/data_sheet/PCF2123.pdf, 2012. Last Viewed February 18, 2012.

[94] P. Pannuto, Y. Lee, Y.-S. Kuo, Z. Foo, B. Kempke, G. Kim, R. G. Dreslinski, D. Blaauw, and P. Dutta. Mbus: an ultra-low power interconnect bus for next generation nanopower systems. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 629–641. IEEE, 2015.

[95] C. Park and P. H. Chou. AmbiMax: Autonomous Energy Harvesting Platform for Multi-Supply Wireless Sensor Nodes. In *Proc. 3rd Ann. IEEE Comm. Society Conf. Sensor, Mesh and Ad Hoc Communications and Networks (SECON'06)*, pages 168–177, Reston, VA, USA, Sept. 2006. IEEE.

[96] C. Park and P. H. Chou. Empro: an environment/energy emulation and profiling platform for wireless sensor networks. In *Sensor and Ad Hoc Communications and Networks, 2006. SECON'06. 2006 3rd Annual IEEE Communications Society on*, volume 1, pages 158–167. IEEE, 2006.

[97] C. Park, J. Liu, and P. H. Chou. B#: a battery emulator and power-profiling instrument. *IEEE design & test of computers*, 22(2), 2005.

[98] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM. DOI 10.1145/2594291.2594339.

[99] PJRC. PJRC teensy 3.2 microcontrollers. https://www.pjrc.com/teensy/teensy31.html, April 2016. [Online; accessed 10 April, 2016].

[100] V. Pop, H. Bergveld, P. Regtien, J. O. het Veld, D. Danilov, and P. Notten. Battery aging and its influence on the electromotive force. *Journal of the Electrochemical Society*, 154(8):A744–A750, 2007.

[101] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava. Design Considerations for Solar Energy Harvesting Wireless Embedded Systems. In *Proc. 4th Int'l Symp. Information Processing in Sensor Networks (IPSN'05)*, pages 457–462, Los Angeles, CA, USA, Apr. 2005. ACM.

[102] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. Tardis: Time and remanence decay in sram to implement secure protocols on embedded devices without clocks. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 221–236, Bellevue, WA, 2012. USENIX. Online at https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati.

[103] A. Rahmati, M. Salajegheh, D. E. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. Tardis: Time and remanence decay in sram to implement secure protocols on embedded devices without clocks. In *USENIX Security Symposium*, pages 221–236, 2012.

[104] M. Raju. UltraLow Power RC Timer Implementation using MSP430. In *Texas Instruments Application Report SLAA119*, 2000.

[105] D. Ranasinghe, R. Shinmoto Torres, A. Sample, J. Smith, K. Hill, and R. Visvanathan. Towards falls prevention: A wearable wireless and battery-less sensing and automatic identification tag for real time monitoring of human movements. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 6402–6405, Aug 2012. DOI 10.1109/EMBC.2012.6347459.

[106] B. Ransford, S. Clark, M. Salajegheh, and K. Fu. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *USENIX Workshop on Power Aware Computing and Systems (HotPower '08)*, Dec. 2008.

[107] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[108] B. Ransford, J. Sorber, and K. Fu. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 159–170, Newport Beach, CA, USA, Mar. 2011. ACM.

[109] B. Ransford, J. Sorber, and K. Fu. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 159–170, Newport Beach, CA, USA, Mar. 2011. ACM.

[110] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[111] L. Rousseau. Secure time in a portable device. In *Gemplus Developer Conference*, 2001.

[112] B. Rumberg, D. W. Graham, S. Clites, B. M. Kelly, M. M. Navidi, A. Dilello, and V. Kulathumani. Ramp: accelerating wireless sensor hardware design with a reconfigurable analog/mixed-signal platform. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 47–58. ACM, 2015.

[113] M. Salajegheh, Y. Wang, K. Fu, A. A. Jiang, and E. Learned-Miller. Exploiting half-wits: Smarter storage for low-power devices. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[114] 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 Trans. Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

[115] 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, 2008.

[116] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys'04, 2004.

[117] V. Shnayder, M. Hempstead, B.-R. Chen, and M. Welsh. PowerTOSSIM: Efficient power simulation for tinyos applications. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004. Online at http://www.eecs.harvard.edu/~shnayder/ptossim/.

[118] F. Simjee and P. H. Chou. Everlast: Long-life, Supercapacitor-operated Wireless Sensor Node. In *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED'06)*, pages 197–202, Tegernsee, Germany, Oct. 2006. IEEE.

[119] 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 ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[120] T. Stathopoulos, D. Mclntire, and W. J. Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'08)*, pages 383–394. IEEE, 2008.

[121] O. Stecklina, D. Genschow, and C. Goltz. Tandemstack-a flexible and customizable sensor node platform for low power applications. In *SENSORNETS*, pages 65–72, 2012.

[122] K. Sun, P. Ning, and C. Wang. TinySeRSync: secure and resilient time synchronization in wireless sensor networks. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 264–277, 2006.

[123] R. Swanson and J. D. Meindl. Ion-implanted complementary MOS transistors in low-voltage circuits. *International Solid-State Circuits Conference*, May 1972.

[124] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In H. Karl, A. Wolisz, and A. Willig, editors, *Wireless Sensor Networks*, volume 2920 of *Lecture Notes in Computer Science*, pages 307–322. Springer Berlin Heidelberg, 2004. DOI 10.1007/978-3-540-24606-0_21.

[125] R. Tessier, D. Jasinski, A. Maheshwari, A. Natarajan, W. Xu, and W. Burleson. An energy-aware active smart card. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, 2005.

[126] S. Thomas, J. Teizer, and M. Reynolds. Electromagnetic Energy Harvesting for Sensing, Communication, and Actuation. In *Proc. 27th Int'l Symp. Automation and Robotics in Construction (ISARC'10)*, Bratislava, Slovakia, Jun. 2010. IAARC.

[127] N. F. Tinsley, S. T. Witts, J. M. Ansell, E. Barnes, S. M. Jenkins, D. Raveendran, G. V. Merrett, and A. S. Weddell. Enspect: A complete tool using modeling and real data to assist the design of energy harvesting systems. In *Proceedings of the 3rd International Workshop on Energy Harvesting & Energy Neutral Sensing Systems*, pages 27–32. ACM, 2015.

[128] E. Vittoz. Low-power design: Ways to approach the limits. *International Solid-State Circuits Conference*, May 1994.

[129] P. Wägemann, T. Distler, H. Janker, P. Raffeck, and V. Sieh. A kernel for energy-neutral real-time systems with mixed criticalities. 2016.

[130] J. V. D. Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, GA, 2016. USENIX Association. Online at https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude.

[131] T. Xiang, Z. Chi, F. Li, J. Luo, L. Tang, L. Zhao, and Y. Yang. Powering indoor sensing with airflows: a trinity of energy harvesting, synchronous duty-cycling, and sensing. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 16. ACM, 2013.

[132] Y. Yang, L. Wang, D. K. Noh, H. K. Le, and T. F. Abdelzaher. SolarStore: Enhancing Data Reliability in Solar-Powered Storage-Centric Sensor Networks. In *Proc. 7th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys'09)*, pages 333–346, Krakow, Poland, Jun. 2009. ACM.

[133] D. Yeager, F. Zhang, A. Zarrasvand, N. T. George, T. Daniel, and B. P. Otis. A 9 μA, Addressable Gen2 Sensor Tag for Biosignal Acquisition. *IEEE Journal of Solid-State Circuits*, 45(10):2198–2209, Oct. 2010.

[134] L. Yerva, B. Campbell, A. Bansal, T. Schmid, and P. Dutta. Grafting Energy-Harvesting Leaves onto the Sensornet Tree. In *Proc. 11th Int'l Conf. Information Processing in Sensor Networks (IPSN'12)*, pages 197–208, Beijing, China, Apr. 2012. ACM.

[135] X. Zeng, L. Zheng, H. Xie, B. Lu, K. Xia, K. Chao, W. Li, J. Yang, S. Lin, and J. Li. Current status and future perspective of waste printed circuit boards recycling. *Procedia Environmental Sciences*, 16:590–597, 2012.

[136] H. Zhang, J. Gummeson, B. Ransford, and K. Fu. Moo: A batteryless computational RFID and sensing platform. Technical Report UM-CS-2011-020, UMass Amherst Department of Computer Science, June 2011.

[137] H. Zhang, M. Salajegheh, K. Fu, and J. Sorber. Ekho: Bridging the gap between simulation and reality in tiny energy-harvesting sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower'11, pages 9:1–9:5, New York, NY, USA, 2011. ACM. DOI 10.1145/2039252.2039261.

[138] P. Zhang, D. Ganesan, and B. Lu. Quarkos: Pushing the operating limits of micro-powered sensors. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 7–7, Berkeley, CA, USA, 2013. USENIX Association. Online at http://dl.acm.org/citation.cfm?id=2490483.2490490.

[139] T. Zhu, Y. Gu, T. He, and Z. L. Zhang. eShare: A Capacitor-Driven Energy Storage and Sharing Network for Long-Term Operation. In *Proc. 8th ACM Conf. Embedded Networked Sensor Systems (SenSys'10)*, pages 239–252, Zurich, Switzerland, Nov. 2010. ACM.

[140] T. Zhu, A. Mohaisen, Y. Ping, and D. Towsley. DEOS: Dynamic Energy-Oriented Scheduling for Sustainable Wireless Sensor Networks. In *Proc. 31st Ann. IEEE Int'l Conf. Computer Communications (INFOCOM'12)*, pages 2363–2371, Orlando, Florida, US, Mar. 2012. IEEE.

[141] T. Zhu, A. Mohaisen, Y. Ping, and D. Towsley. Deos: Dynamic energy-oriented scheduling for sustainable wireless sensor networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 2363–2371. IEEE, 2012.

[142] T. Zhu, Z. Zhong, Y. Gu, T. He, and Z. L. Zhang. Leakage-Aware Energy Synchronization for Wireless Sensor Networks. In *Proc. 7th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys'09)*, pages 319–332, Krakow, Poland, Jun. 2009. ACM.

[143] T. Zhu, Z. Zhong, T. He, and Z. Zhang. Energy-Synchronized Computing for Sustainable Sensor Networks. *Ad Hoc Networks*, 11:1392–1404, 2013.

[144] T. Zhu, Z. Zhong, T. He, and Z. L. Zhang. Feedback Control-Based Energy Management for Ubiquitous Sensor Networks. *IEICE Trans. Communications*, E93-B(11):2846–2854, 2010.