Peripheral and Power Management in Batteryless, Energy-Harvesting Systems

Submitted in partial fulfillment for the requirements for
the degreee of
Doctor of Philosophy
in
Electrical & Computer Engineering

Emily K. Ruppel
B.S., Electrical & Computer Engineering, University of Maryland, College Park
M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August 2022

Abstract

Batteryless, energy-harvesting devices enable deeply embedded sensing and computing deployments without the size, weight, or maintenance constraints of batteries. These devices harvest energy into a small capacitor bank to support short bursts of execution at a time, with power failures in between. Peripheral sensors and radios are essential to batteryless device deployments; they allow a device to sense and report information about its surroundings. However, peripherals trigger concurrent accesses to memory and account for a large percentage of the total device energy budget. The challenge for programmers is a tight coupling emerges between device hardware characteristics and application peformance. A software developer needs to have an understanding of both to write programs that behave as expected. This thesis reduces the burden of integrating peripherals into batteryless applications by managing the shared state between peripherals and the primary microcontroller.

We first address the problem of shared memory between peripheral triggered interrupts and a program's main loop by defining a programming model that prevents concurrency control violations due to power failures. Peripheral operations also affect the state of the system by changing the total operating power, e.g. by turning a sensor on or off. We demonstrate that peripheral mismanagement leads to a new class of energy bugs that cause hard-stop failures and waste energy in batteryless systems. To correct these bugs, we model changes in peripheral power over a program's execution. However, peripheral power does not fully capture the hardware state of the device. A device's power system characteristics also determine when it is safe to access a peripheral. Using a lightweight model of the power system, we correct existing scheduling algorithms for batteryless devices to accommodate high-current peripherals on volume-constrained systems. Finally, we apply the lessons learned throughout this thesis to develop the failure-aware hardware and software support for a miniature batteryless satellite. The ease with which we integrate failure-agnostic peripheral subsystems demonstrates the value of power and peripheral management in batteryless devices.

Acknowledgments

This thesis was made possible by the guidance, curiosity and encouragement of far more people than I can reasonably name. First, I would like to thank my advisor, Brandon Lucia, for his mentorship throughout my PhD. Brandon's enthusiasm is infectious, and my approach to research has been shaped by his willingness to shoot for the moon, literally and figuratively. Beyond guiding my research, Brandon was extremely generous with his time and effort. Whether I was learning to write papers, give talks, or interact with the research community, he was there every step of the way. So thanks for everything– for the professional opportunities, the travel, the debates about Pittsburgh coffee shops and for setting an excellent example.

This thesis sits at the intersection of several fields of research, and the members of my committee have been invaluable in navigating these spaces. Limin Jia's feedback on many a paper-draft has ironed out any number of unstated assumptions and pushed me to actually think about the shape of a paragraph. I greatly appreciate Josiah Hester's enthusiasm and advice when pivoting portions of this thesis to the world of ubiquitous computing. Thanks to Anthony Rowe for encouraging me to consider how batteryless systems fit into the wider world of sensing. And thanks for convincing me to get started on the next chapter of my career while finishing up this one. Portions of this thesis might still be in bulleted form without the resulting clarity and time sequestered on an airplane.

Beyond my committee, I was supported in graduate school by many members of the Carnegie Mellon community. I want to start by thanking all the members of the ABSTRACT research group for providing thought-provoking questions, feedback and camaraderie. Special thanks to Alexei Colin for pulling me into the world of hardware prototyping and showing me what it means to carry out a top-notch evaluation. Thanks to the other early ABSTRACT members, Kiwan Maeng and Vignesh Balaji, for their steadfast presence during many paper deadlines and conferences. Thanks to Milijana Surbatovich for patiently describing the syntax for formal modeling no matter how many times I asked and for the succinct writing that brought Culpeo to life. Thanks to Harsh Desai for the whiteboard-sessions about all things hardware and for the many conversations beyond MCUs. Thanks to Brad Denby for sharing his enthusiasm for satellites. Thanks to Nathan Serafin for his attention to detail, especially in last minute design reviews. Thanks to Graham Gobieski, McKenzie van der Hagen and Souradip Ghosh for being excellent travel companions. More broadly, I want to thank the many participants of the weekly lunch meetings. Thanks to Nathan Beckmann for his thoughtful, direct questions about this work and to the members of the CORGI research group. In particular, Brian Schwedock and Sara McAllister have listened to and provided feedback on so many talks, even as my work veered far from anything that looked like a cache. I would also be remiss if I did not acknowledge the actual corgis (Arya and Baphomet) for being delightful.

Thanks to the many CMU students, faculty and staff I have learned from, whether it was inside or outside of a classroom. Thank you to the talented students outside of ABSTRACT with whom I have collaborated, including Vaibhav Singh, Mara Kirdani-Ryan, Chad Taylor, Shize Che and Fayyaz Zaidi. Beyond direct collaboration, thanks to the members of the WiTech and WiSE labs for accommodating my inevitably last minute requests to fill tool or wireless knowledge gaps, Artur Balanuta, Elahe Soltanaghai, Akarsh Prabhakara, John Miller, Akshay Gadre, Atul Bansal, Jingxian Wang and Junbo Zhang. Special thanks to Diana Zhang and Adwait Dongare for being a remote well of laughs, cooking know-how, and RF intuition. The many denizens of the fourth floor of the CIC always offered stimulating conversations over coffee, especially Ankur Mallick, Samarth Gupta, Rachata Ausavarungnirun, Saughata Ghose and Utsav Drolia. Thanks to the members of EGO for their dedication to getting graduate students to talk to each other, especially Joe Melber, Elliott Binder, Mark Blanco, Mohammad Bakhshalipour, Anuva Kulkarni and Thom Popovici. I am grateful that Michael Kleyman, Brad Johnson, Kim Jin, Shannon Gallagher, Abby Smith and Robin Dunn got me out of the lab on many occasions. Thanks to Dimitrios Skarlatos and Akshitha Sriraman for the sage job advice exactly when I needed to hear it.

Beyond the Carnegie Mellon community, this work would not have happened without professional support from many sources. This work was funded by the National Science Foundation and the Semiconductor Research Corporation (SRC)[1]. Within SRC, the CONIX research center facilitated many opportunities to interact with students from across the country. The Computer Architecture Student Asso-

[1]This work was supported by National Science Foundation Award #1751029 and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

ciation provided wonderful connections to other students while conferences were on pause for the COVID-19 pandemic. My early research mentors in and around Maryland, including Bruce Jacob, Robert Newcomb and Allen Goldstein, showed me what research can be. Numerous internship mentors shaped me as an engineer and taught me how to communicate across different levels of the stack, among them are Jim Blair, Tige Young, Stephen Glancy, Dave Cadigan, Rona Yaari, Supreet Jeloka and Brian Cline. Thanks to Tom Puzak for showing me around IBM Watson and kindling my interest in industry research. Finally, thanks to the members of the Gemstone Team BIKES for showing me what Terps can accomplish if they simply refuse to quit.

I am deeply grateful to the friends and family who provided the non-academic support that enabled this thesis. Thanks to LeAnne Young, Tammy Walsky and all of the friends from Maryland who could make a brief dinner or run between semesters feel like home. Thank you to all of the people who adventured around Pittsburgh with me, whether running, biking, backpacking, brewery hopping or a combination thereof. In particular, I want to thank Jen Brown, Megan Sharretts, Kristin Yahner, Tom Weir, Heather Adams, Nate Nypaver, Shawn Kelly, Barrie Slaymaker, Joanna Wolfe, Julie and Brett Lynch for always putting things in perspective.

Most importantly, none of this work would have been possible without the unfailing support of my family. Tom, thank you for being my sounding board, rubber duck and most trusted source on how a transistor works. In all seriousness, your love and encouragement through the highs and lows of this process mean the world to me. Thank you to my twin, Abbie, for showing me how much fun Pittsburgh could be in the first place, and for demonstrating that success is primarily living life to the fullest. Thanks to my grandmother, Helen, and my extended family for their continued support, no matter how long it seemed I would be in school. Finally, I want to thank my parents, Joe and Lisa, for encouraging a meticulous and inquisitive mindset that started with dissecting Harry Potter and has culminated in this thesis. Thank you for being in my corner from the beginning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the Internet of Things (IoT) expands, it is pushing more and more deeply embedded sensors out into the world. These sensors gather critical data from their environment and perform some actuation in response or transmit the data to a more powerful base station to relay across a network. It is these tiny nodes well beyond "the edge" [214] of a distributed network that provide the essential information to make a smart and connected world a reality. However, as sensor data volumes grow, these devices do more than basic sense-and-send tasks. To improve reliability [132], reaction time [99], and energy efficiency [98] deeply embedded sensors can use onboard compute resources to process the sensed information before acting on it.

In typical sensor network applications for IoT, battery-powered motes schedule sensing, processing, and compute tasks with sleep operations that drop the device into a low-power wait state [84, 141, 209]. Programming models for battery-powered motes are designed to help the programmer maximize battery life given a set of applications. The goal is to do as little as possible and still meet application demands. Eventually, though, the battery is fully discharged and the device is dead until the battery is replaced. In contrast, energy-harvesting sensor nodes greatly extend device lifetime by using rechargeable batteries or capacitors to store energy captured from the environment [7, 59, 88, 124, 128, 180, 216]. Energy-harvesting systems capture sources like solar, RF, vibration, or temperature gradients [180, 229, 260, 279].

However, rechargeable batteries do not solve all challenges associated with ubiquitous computing initiatives. Reducing system weight, volume, or cost can preclude the use of batteries entirely in applications like miniature satellites [285], inside the body [62, 119, 156, 211] or object tagging [208]. Uncontrolled environments also can be out of scope for batteries due to temperature constraints [171, 194] or humidity [8]. Further, size and weight constraints limit the harvestable power in addition to the available energy by capping the size of harvesters. For instance, solar harvester power is determined by the area of the panel– the power drops off as the size is reduced to fit in a tiny envelope. If the power to run the sensor, including the central microcontroller (MCU), and the peripheral sensors, radios, and actuators, exceeds the harvested power, the device must draw the energy from its energy buffer, a capacitor. In highly constrained systems, the capacitor discharges rapidly and the device is forced to power off. Once the device is off and no longer using power, the harvester can refill the energy buffer and the device can once again turn on. As strange as a device that intentionally turns on and off may seem, prior work has succeeded in building complex applications including maintenance-free sensing [8, 44, 110, 122, 217, 229], gaming [64], image processing [191, 193], miniature robotics [233] and machine learning [99, 121, 193]. This thesis focuses on these *batteryless* devices that use a capacitor for an energy buffer and harvest too little power to operate continuously.

The key innovation from prior work that enabled long running applications (i.e. those that take more energy than is ever available in the energy buffer) is an execution model called *intermittent* computing. Intermittent computing extends the lifetime of a software task across hardware power failures. This work started in the domain of Computational RFID tags (CRFIDs), and allowed CRFIDs to compute very slowly [44, 217, 229, 290, 292] with minimal harvested power. Novel circuit-level hardware support for intermittent computing pursues low-power persistent memory technologies tightly integrated with the processor [170] or in the memory [127, 261, 262], integrated checkpointing [269] and cheap voltage monitoring [280]. However, this thesis focuses on software and printed-circuit board (PCB) level support for intermittent systems, working only with commodity MCUs, sensors, actuators and power system components, composed in novel ways. Focusing on commodity hardware enables batteryless device usage in the short term and defines strategies that can reduce cost in the future. For instance, supporting the mixture of volatile and non-volatile memory in commodity MCUs [268] allows batteryless devices to use cheap, well established, low-power MCUs instead of an exotic fabrication technology [170].

The original runtime support for intermittent execution was designed for applications with tens of milliseconds of active time on each reboot. Subsequent software work followed suit and developed *task-based* programming models [55, 168, 174] that minimize the overhead of computing across power failures. These models divide the program into atomic regions of code and guarantee that the regions will eventually complete from start to finish, without an intervening power failure, and do so idempotently. The problem though, is this work only focused on how to extend compute across power failures. It relied on extremely basic peripheral sensors and actuators, including nearly passive radio communication [164], on-chip ADCs [168] and LEDs [168]. Such restrictions are necessary for CRFIDs– doing any more than quickly reading from a peripheral is prohibitively expensive for truly tiny devices. However, peripherals are absolutely critical to the mission of batteryless systems. Putting a computer in a hard-to-reach physical location is only useful because it is co-located with peripherals to sense data that cannot be acquired by other means. Radios or actuators (which are peripherals) then allow the device act on the data.

1.1 Peripherals in Batteryless Devices

To understand the challenge of adding peripherals to an intermittent execution, we must consider the typical hardware architecture of a batteryless device. To the best of our knowledge, all batteryless systems give a single MCU the responsibility to restore state after power failures, coordinate peripheral power access [57, 61, 284] and manipulate peripheral configuration [33, 41, 220]. Peripherals may be complete Systems on Chip (SoCs) in their own right, but we consider any chip that is not failure-aware (i.e. the central MCU) a peripheral. It is thus the MCU's job to manage all of the constraints a peripheral places on a program and all of its *state*. State in this thesis comprises the memory, and configuration of a peripheral. The MCU must persist concurrent changes to peripheral state and satisfy peripheral *constraints*, including timeliness and energy requirements, while the whole device is executing intermittently.

Managing peripherals on batteryless devices is more difficult than on a continuously powered device for two major reasons: first, peripheral state is cleared on a power failure, second, peripherals consume a large percentage of the total energy budget of a batteryless device. When power fails, each peripheral's configuration, i.e. the operating mode that results from commands sent by the MCU, is assumed to be lost because the voltage supplied to the peripheral (likely) falls below its minimum operating voltage. Upon reboot, the MCU must carefully reinitialize peripherals before using them to prevent the program from hanging due to a misconfigured peripheral. Further, the dynamic software state shared with the peripheral, e.g. memory accesses and code in a peripheral triggered interrupt service routine (ISR), is also lost in the event of a power failure. If a power failure interrupts an ISR, the ISR is not restarted on reboot– the source of the interrupt is gone. The loss of state forces the programmer to judiciously use peripherals with predictable access patterns, e.g. polling, and their relatively high power raises the cost of mismanaging a peripheral.

Figure 1.1 shows the percentage of the total power each peripheral individually consumes on the Capybara [57] batteryless sensing and computing platform. On a single platform, the cost ranges from 10% to 40% to run one peripheral at a time. That said, the actual impact of a peripheral (i.e. the same sensor) will vary between devices and deployment scenarios. For instance, if a peripheral deployed on a solar powered sensor that generates 90 mW [166] is moved to an RF harvesting node (10 mW [183]), its effect on the system behavior will be more pronounced. Beyond the total system power, the incoming power determines which sensors will have a noticeable impact on end-to-end efficiency. Similarly, decreasing the frequency of the central MCU can change the system's baseline power by more than 8x [268], which increases the percentage of system power attributed to a peripheral. As a result, it is difficult for a software developer to know when to prioritize peripheral state management, i.e. strategically putting the peripheral to sleep to save power, and the effort to manage peripherals is non-trivial.

Previous work in intermittent computing guaranteed correctness strictly in terms of the memory state of the MCU [55, 174, 254], and subsequent models addressed other aspects of running peripherals including timeliness [63, 111, 148, 253], atomic operation [57, 176, 253], and peripheral configuration [33, 41, 220]. The problem is that these models are not designed to work together. In fact, we show that combining seemingly independent pieces of system support for using peripherals on batteryless systems causes buggy execution.

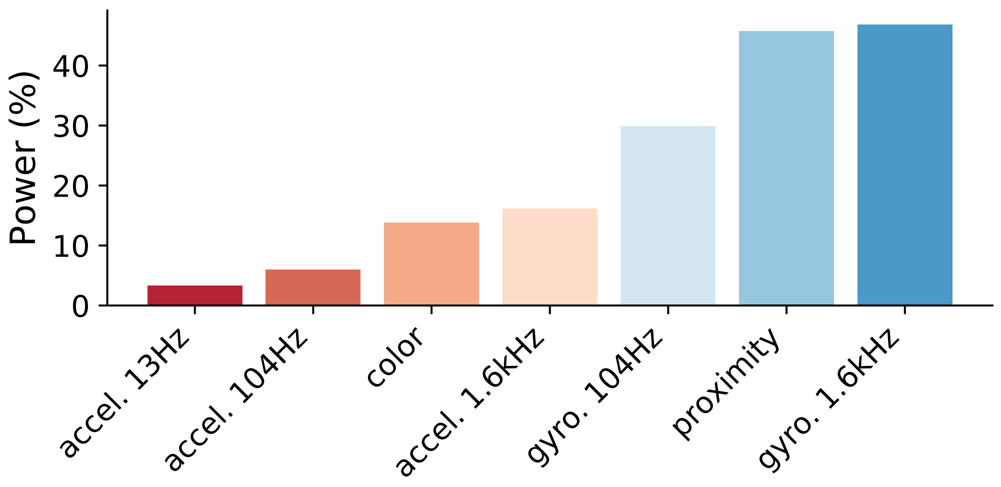


Figure 1.1: Peripheral Power Percentage. The peripherals on a Capybara device consume a large percentage of the total power, but there is a wide range based on the exact peripheral configuration.

1.2 Thesis Statement & Contributions

The goal of this thesis is to manage the shared state that exists between peripherals and the central MCU. On a batteryless device, the failure-aware MCU explicitly coordinates a large set of failure-agnostic peripherals from the main loop of a program. However, additional state is shared between the MCU and peripherals, e.g ISR accesses to global memory, that implicitly allow peripherals to affect the program running on the MCU. This thesis identifies unintended interactions that emerge between peripherals and the MCU in batteryless devices and builds programming models, compiler tooling, and light-weight hardware models to correct them. In the course of demonstrating the interactions, removing the problem, and evaluating the results on batteryless hardware, we demonstrate the validity of the following thesis:

> System support for managing the data, power, and energy state that are shared by peripherals and the central microcontroller improves the reliability, performance, and programmability of batteryless, energy-harvesting devices.

The key contributions of this thesis are as follows:

1. We identify concurrency control bugs in task-based languages for intermittent execution that prevent programmers from correctly synchronizing accesses to shared

memory from peripheral-triggered interrupt service routines. We then propose a programming model that provides correct, low-overhead access to shared data.

2. We describe a new category of energy bugs due to changes in peripheral operating power that cause hard-stop failures in batteryless systems and demonstrate the effects in hardware. We propose a programming interface and compiler tool that reports potential hard-stop energy bugs.

3. We demonstrate that peripheral mismanagement in the context of batteryless energy harvesting systems manifests as application-wide slowdowns. We develop a failure-aware runtime module that automatically configures peripherals to minimize their total energy consumption.

4. We show that energy-based assumptions for peripheral event scheduling fail when using dense supercapacitors as an energy buffer. We develop a lightweight model to accurately predict the effect of a software task (e.g. peripheral access) on the state of the energy buffer at runtime.

5. We apply the peripheral-induced bug awareness accrued in this thesis to build an energy-harvesting power system and failure-aware control system for a batteryless nanosatellite. The failure-aware subsystems support a wide range of failure-agnostic peripherals that allow for greater flexibility and reduced development effort when building complex batteryless systems.

Detailing the range of problems that occur as system developers integrate more capabilities into batteryless systems clarifies the gap between where system support for intermittent execution is, and where it needs to be. The tooling presented in this thesis takes one step towards closing that gap, and empowers developers to build impactful batteryless applications.

1.3 Outline

The remainder of this thesis is organized into six chapters. This introduction explained the basics of intermittent execution as a model of computation to motivate the importance of this work's contributions. Chapter 2 will delve into the specifics of the hardware and software that underpin batteryless applications before exploring trends in intermittent computing literature that motivate this thesis' focus on peripherals. With the background in place, we move on to Chapters 3-5 which each describe a different type of state shared between peripherals and the central MCU.

Chapter 3 poses the question, how should intermittent execution models support arbitrary accesses to shared memory? After exploring the software bugs that can emerge while trying to control concurrent accesses to shared memory, this chapter ultimately answers “it shouldn’t”. We demonstrate in this chapter that supporting arbitrary access to shared memory while allowing flexible semantics has a prohibitively high overhead in many cases. Instead, we propose Coati [224], a programming model and runtime implemented in C that restricts the quantity of data that is actually shared between asynchronous and scheduled code. The restriction allows Coati to minimize the overhead of shared state and achieve concurrency control with little programmer effort.

Chapter 4 continues the theme of bugs due to concurrent peripheral accesses, but we study the system-wide effects of an individual peripheral’s power consumption instead of its accesses to shared memory. In this study, we find that a programmer’s decision of whether or not to turn off a peripheral after using it has a substantial effect on the performance and correctness of an application. In particular, we examine the consequences of system support for intermittent execution that does not consider peripheral power. We show that the resulting systems are not composable without introducing the potential for fail-stop bugs. We then develop a methodology called Pudu to remove the bugs that result from peripheral power. Pudu comprises a set of annotations, implemented in C, that indicate changes in peripheral power as well as compile-time and runtime components implemented, respectively with LLVM [153] and C. Using the Pudu methodology we identify bugs that cause hard-stop failures and remove the inefficiency of poor power management code.

Chapter 5 addresses the problem that modeling peripheral power is not sufficient to model the hardware behavior of recent batteryless devices. Recent systems moved to dense supercapacitors to increase the energy availability, but existing models do not account for the chemistry of supercapacitors. We show that simplistic capacitor models do not correctly predict when an operation will complete without a power failure, but we also show that detailed static profiling of supercapacitors is impractical and inaccurate. In the end, we build Culpeo, an interface for expressing basic power system characteristics to software and a runtime library written in C that predicts a safe energy buffer voltage for starting an operation. We show that Culpeo is essential for event-triggered peripheral operations because it prevents power failures due to sudden spikes in peripheral-power.

Using the lessons learned throughout this thesis, Chapter 6 discusses a case study to build the “ideal” batteryless system support for a PocketQube nano-satellite [6]. The trouble with building the failure-aware support for a device as complicated as a nano-satellite, is it needs to enable other subsystems with as little integration effort as possible. The simpler the integration, the more flexible the resulting system and the more parallel the development of the subsystems can be. This requirement precludes failure-aware modifications to the code running on the other subsystems. We use the knowledge of the hardware-software bugs explored in this thesis to correctly support a range of capable subsystems. For instance, the failure-aware handling of interrupts from other subsystems was informed by Coati’s conclusion that a narrow interface minimizes programming effort and overhead. The peripheral voltage rail design is motivated by the challenge of managing peripheral power explored in Pudu, and bounding the behavior of applications running on the satellite is not accurate without Culpeo.

Finally, Chapter 7 summarizes the results of the work explored in this thesis and defines several directions for new research beyond what is discussed in the preceding chapters.

Chapter 2

Background

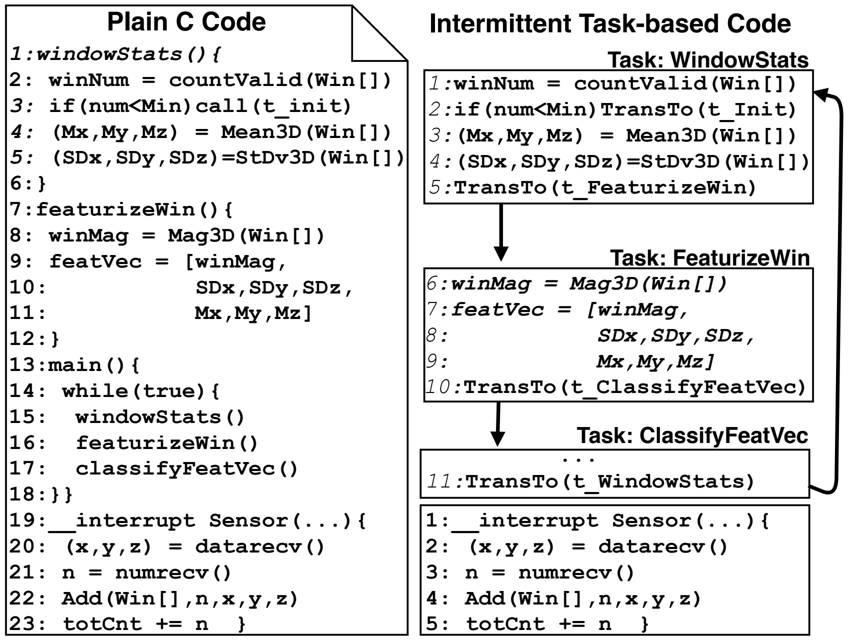
Batteryless systems differ from battery-powered systems in ways that affect both the software execution and hardware design-space trade-offs. This chapter will first describe the software execution model batteryless devices experience and describe how prior work solved the challenges it presents. We then present the energy harvesting power system architecture that is typical of batteryless devices, which we will reference throughout this thesis. After establishing the underlying hardware and software requirements for batteryless systems, this chapter explores existing techniques for enabling peripherals in batteryless devices.

2.1 Intermittent Execution

The defining characteristic of software on a batteryless device is the intermittent execution model [168] that it experiences. Batteryless devices harvest all of the energy they will use from their environment, but operating power is typically higher than harvestable power. As a result, batteryless devices need to store energy in an energy buffer that they can draw from to power the application hardware (load-side components) including the MCU, sensors, and radios. As software runs, the energy buffer rapidly depletes, forcing the load-side components to either power down or enter a deep sleep state to allow the energy buffer to recharge.

The challenge for software running on a batteryless, energy-harvesting device is that code executes only intermittently, making progress during operating bursts. Further, this thesis considers commercially available MCU's that mix byte-addressable, non-volatile main memory (NVM) [263, 265, 268], with a volatile processor architecture. For MCUs with hybrid non-volatile and volatile memories, each power failure causes the execution to lose volatile program state (e.g., stack, registers) and retains non-volatile program state (e.g., some globals). Power failures compromise forward progress by clearing the program counter (PC) and restarting the program. Power failures can also leave program state inconsistent by forcing the state of the volatile memory out of sync with the data stored in NVM [168, 217, 252]. However, increasingly well established programming models for intermittent systems allow application developers to overcome the challenges of losing volatile state to complete long running programs piece-by-piece [254].

Figure 2.1, left, shows an excerpt of a plain C program that performs activity recognition, using sensor data. The code loops over a rolling window of data, computing statistics about the data, assembling a feature vector, and classifying the data. Figure 2.2, left, shows the program intermittently executing. As power fails, the intermittent execution does not make forward progress, and repeatedly restarts from `main()`.



```
windowStats(){
 winNum = countValid(Win[])
 if(num<Min)call(t_init)
 (Mx,My,Mz) = Mean3D(Win[])
 (SDx,SDy,SDz)=StDv3D(Win[])
}
featurizeWin(){
 winMag = Mag3D(Win[])
 featVec = [winMag,
              SDx,SDy,SDz,
              Mx,My,Mz]
}
main(){
 while(true){
  windowStats()
  featurizeWin()
  classifyFeatVec()
}}
__interrupt Sensor(...){
 (x,y,z) = datarecv()
 n = numrecv()
 Add(Win[],n,x,y,z)
 totCnt += n  }
```

```
winNum = countValid(Win[])
if(num<Min)TransTo(t_Init)
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv3D(Win[])
TransTo(t_FeaturizeWin)
```

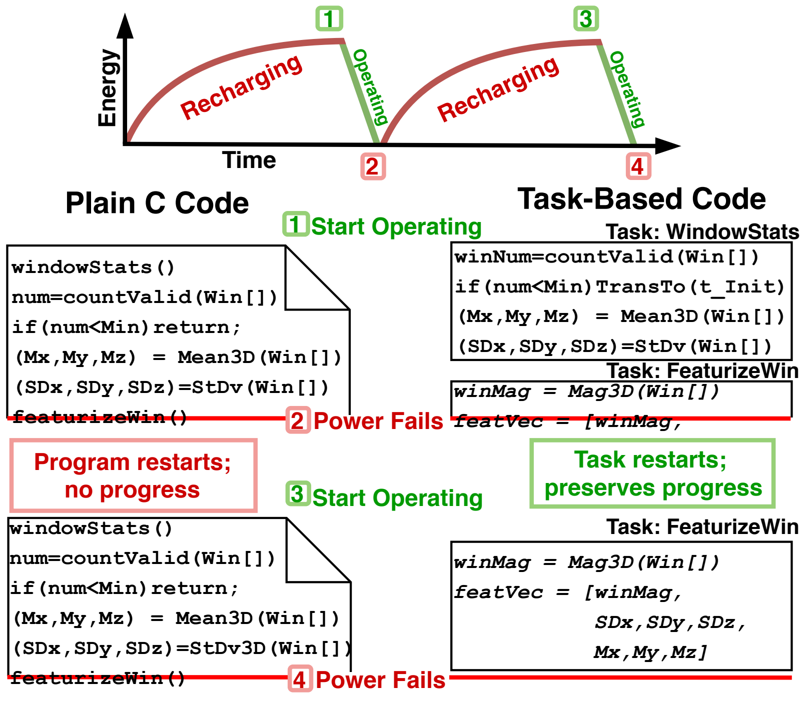
```
winMag = Mag3D(Win[])
featVec = [winMag,
              SDx,SDy,SDz,
              Mx,My,Mz]
TransTo(t_ClassifyFeatVec)
```

```
...
TransTo(t_WindowStats)
```

```
__interrupt Sensor(...){
 (x,y,z) = datarecv()
 n = numrecv()
 Add(Win[],n,x,y,z)
 totCnt += n  }
```

Figure 2.1: Activity Recognition code. The code at right translates the C code at left into task-based code for intermittent execution.

However, numerous systems have been developed to help make progress through



```
windowStats()
num=countValid(Win[])
if(num<Min)return;
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv(Win[])
featurizeWin()
```

```
winNum=countValid(Win[])
if(num<Min)TransTo(t_Init)
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv(Win[])
```

```
winMag = Mag3D(Win[])
featVec = [winMag,
```

```
windowStats()
num=countValid(Win[])
if(num<Min)return;
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv3D(Win[])
featurizeWin()
```

```
winMag = Mag3D(Win[])
featVec = [winMag,
           SDx,SDy,SDz,
           Mx,My,Mz]
```

Figure 2.2: Plain C code vs intermittent task-based code. Task-based code makes progress despite power failures.

a program that requires more energy than will ever be available at once on the batteryless device. For instance, Figure 2.1 shows the example program re-written to use *tasks* from the Alpaca language [174], short functions with statically annotated control flow that execute atomically. The code's main functions map to tasks and arrows indicate control-flow. The interrupt is its own task and has no control-flow arcs because it is asynchronous. Figure 2.2 shows the task-based program executing intermittently. A task runtime buffers updated values until it completes, commits its updates, and transitions to another task; making progress one task at a time. After a power failure, execution restarts at the most recent task, instead of `main()`; in the figure, `FeaturizeWin` resumes after the failure. While all system support for intermittent execution is designed to support forward progress, the exact implementation of the runtime system affects program properties ranging from correctness to power efficiency.

2.2 Software Support for Intermittent Execution

Prior work has developed many different software approaches for preserving progress that can be grouped into task-based, software checkpointing or hardware checkpointing systems. Figure 2.3 summarizes the different systems. This section will explore the advantages and disadvantages that stem from the type and implementation of the intermittent runtime a program uses.

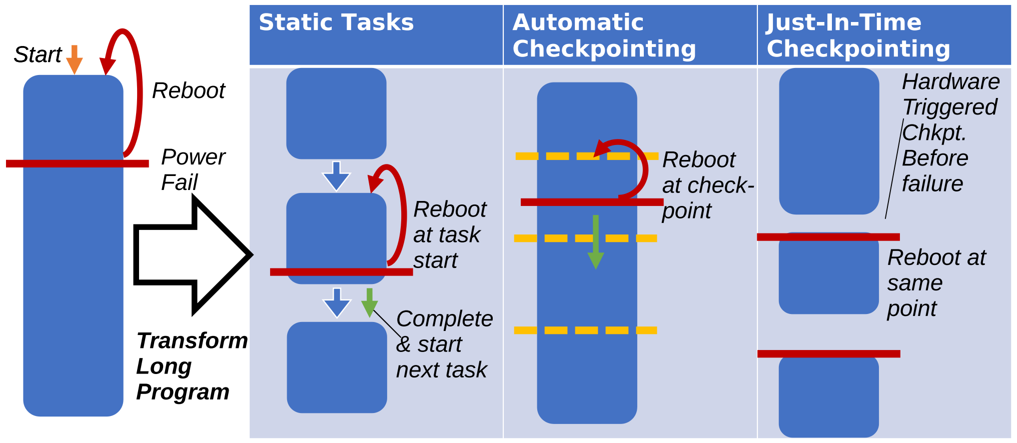


Figure 2.3: Software Support for Intermittent Execution. Strategies for intermittent computing all divide a long running program into shorter segments, but they vary in their hardware and programmer requirements.

A task-based intermittent programming system requires the programmer to break a program into regions of code that the programming model guarantees will execute atomically and idempotently. The programmer ensures that a task will finish within the device's energy budget by conservatively testing the code before deployment [57, 111, 174, 175, 284] or using energy debugging tools [54, 56]. The state-of-the-art method to ensure each task completes is to fully charge a device's energy buffer, disconnect harvested power, and then initialize the code and run a task in isolation. A task-based model's runtime system implementation ensures task atomicity by ensuring that repeated re-executions of a task are *idempotent*. An idempotent task always produces the same result when re-executed, never overwriting and losing the input values to the task. On reboot, the runtime system replays a log of checkpointed values (if required by the task programming model) and jumps to the beginning of the failed task.

A key challenge for systems developers expanding on task-based programming models is each system ensures idempotent operation differently, which results in slightly different guarantees. For instance, DINO used programmer inserted task boundaries to form tasks and preserves idempotence by logging global variables involved in write-after-read (WAR) conflicts. However, a programmer must consider control flow between boundaries when placing them to avoid tasks that are too long, e.g. because a programmer left a loop exit unmarked. In contrast, several task based runtimes, including Alpaca mentioned above, define tasks as separate, top level functions that communicate via some WAR-aware interface to global memory [55, 57, 111, 174, 175, 284]. Even among statically defined tasks, the exact procedure for avoiding write-after-read conflicts affects the runtime's susceptibility to subtle bugs. Alpaca reduced runtime overhead further than Chain by privatizing only variables involved in a WAR dependence, but this policy leaves it open to RIO bugs [252] that occur when a variable is not written on all paths from an input dependent branch. Systems that place checkpoints in software automatically at compile time suffer from the similar problems– balancing overheads with guarantees is difficult.

A primary question software checkpoints must answer is how far apart checkpoints should be placed to keep the checkpointing overhead at a minimum, while avoiding non-termination [56]. Checkpoints may be placed by the programmer [217] or the compiler [25, 34, 56, 64, 113, 186, 269]. Like tasks, checkpoints preserve state so that on reboot, the state can be replayed and the execution can resume from the most recent checkpoint [217]. Unlike statically defined tasks, the overhead of checkpointing includes saving the registers as well as the stack in addition to any variables that may become inconsistent as a result of a power failure. The memory overhead of checkpointing varies with the placement of the stack in hardware. If the stack is placed in non-volatile memory, only the registers need to be persisted, so the overhead is low [176], otherwise the entire stack must be persisted to NVM. Other checkpointing strategies reduce overhead by using differential checkpointing to only save the difference from the last checkpoint [64].

Both software checkpoints and tasks suffer from the same problem of increasing runtime overheads as the task size or distance between checkpoints shrinks. Ideally both would be matched to the hardware platform a program is running on, but in reality the effort to rewrite the program for each new platform is prohibitive. Instead, just-in-time (JIT) checkpoints use a hardware triggered interrupt to indicate an imminent power failure. The JIT runtime records a checkpoints before powering down and resuming from the exact point of the power failure [30, 31, 126, 176]. The interrupt is often triggered by an internal comparator on the MCU that is monitoring the energy buffer voltage. If the energy buffer voltage falls too low the interrupt fires. The advantage of JIT checkpointing is that it does not require re-execution. After a power failure the application resumes from the exact point where the interrupt fired. Beyond reducing wasted work, JIT also removes the need to log values involved in WAR conflicts because code always resumes where it left off– it is not possible to read a different value than what was written. The primary drawback to JIT checkpoints is they assume that the energy cost to persist all volatile state (i.e. stack, registers) is insignificant compared to the energy buffer size, which does not hold true for very small systems. Several proposals have argued for a non-volatile stack to reduce the overhead and complexity of checkpointing [176], but other work showed the inefficiency of mapping the entire memory to NVM [127]. Additionally, JIT checkpointing requires integration with the power system of the device so it can measure the energy buffer voltage.

2.3 Energy Harvesting Power Systems

A batteryless device relies on a power system that harvests energy from its surroundings, accumulates it in an energy buffer, and releases it to perform a short burst of computation. The hardware components of an EHD can be split in two parts: *supply-side* power-system components (regulators, capacitors) that collect and store energy and *load-side* components (microcontrollers, sensors, radios) that execute software. A range of intermittent computing platforms have been developed that span a variety of size and power delivery capabilities [57, 63, 64, 108, 110, 146, 167, 193, 229, 233, 243, 290]. In this thesis, we focus on the power systems that are typical for energy harvesting devices that rely on supercapacitors to support (relatively) high energy, high load current operations.

Figure 2.4 shows a typical energy-harvesting power system for this domain [57, 64, 193, 281], with an input and output booster as well as a supercapacitor for energy storage. Power is harvested from a weak supply that cannot power the device continuously. This thesis primarily evaluates applications using stable voltage sources like solar power from small panels [57, 72, 166] that provide inputs of just μWs, but this work is agnostic to the exact power source. The input booster regulates voltage from the energy harvester to steadily charge the energy-buffer (capacitor), up to a maximum voltage level (V^{high}). The input booster is required because the energy-buffer will only charge as high as the incoming voltage from the energy harvester. To maximize the energy stored in the capacitor, the input booster boosts the incoming voltage to V^{high}. To execute software, the output booster must be enabled. The output booster provides a stable voltage to the *load-side* components, discharging the capacitor and decreasing the capacitor's voltage level, V^{cap}. This output booster can only be enabled when V^{cap} is above a device-specific minimum value (V^{off}). In other words, software executes only when V^{cap} is between V^{high} and V^{off} (e.g. between 2.4V to 1.6V). When software deactivates (i.e. when V^{cap} falls below V^{off}), the system uses hardware to fully recharge to V^{high} before the output booster is re-enabled [57, 63, 110, 166, 193, 229].

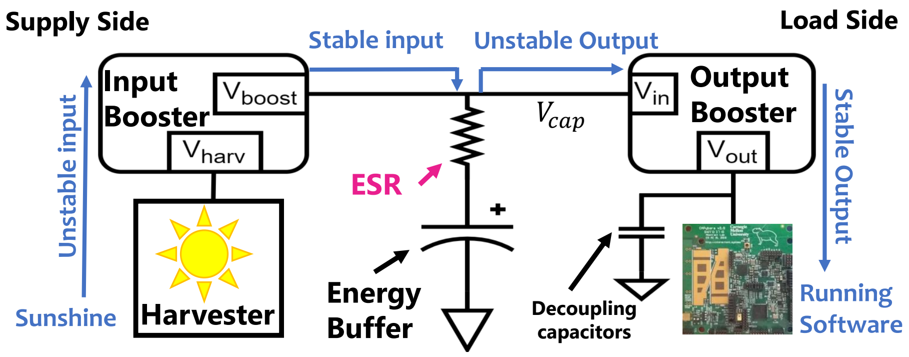


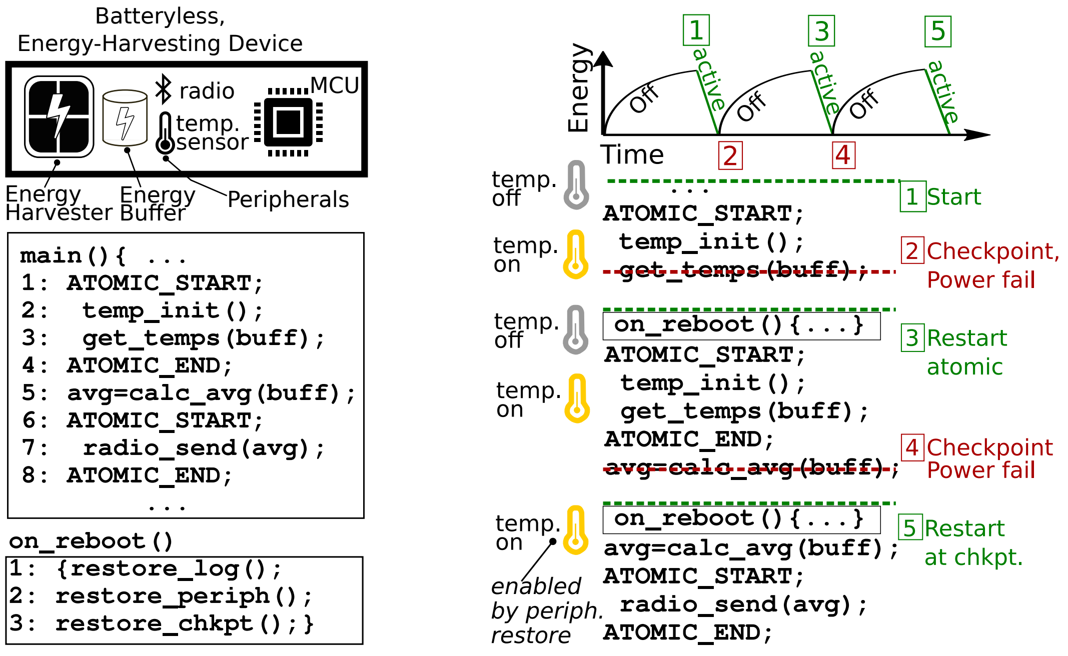
Figure 2.4: Annotated power system schematic. The supply side harvests energy into the energy buffer that the output booster accesses to provide a stable voltage to the load.

Power systems for recent batteryless energy-harvesting devices use supercapacitor arrays as their energy buffers instead of other capacitors or batteries [166, 193, 233]. Supercapacitors provide an attractive balance between energy capacity and lifetime; they provide much higher capacities than (e.g. ceramic) capacitors while lasting for decades [140, 198, 296]. In contrast, a rechargeable battery lasts only a few months under the high duty cycles typically observed in energy-harvesting systems. A challenge, however, is that the effective series resistance (ESR) of the dense supercapacitors considered in this thesis causes a substantial voltage drop on V^{cap} with even milliamps of current drawn by the load [23, 137, 235]. As we explore in Chapter 5, the output booster masks the ESR drops by continuing to provide a stable output voltage as V^{cap} drops, but unexpected power failures can still occur if V^{cap} dips below V^{off}. We demonstrate in the next section that batteryless devices can support a wide range of peripherals that enable useful applications, but peripherals come at a power cost.

2.4 Enabling Peripherals

Batteryless, energy-harvesting, intermittent systems, like other deeply embedded devices, rely on peripheral sensors and actuators to sense, process, and transmit data. The central MCU coordinates a potentially large set of off-chip peripherals using a combination of bus protocols (e.g. I2C, UART) and voltage triggers over GPIO pins. Peripheral management is challenging because the program running on the MCU is responsible for ensuring that all peripherals are in the correct operating mode at all times. Figure 2.5a, shows an example device and a simple program that reads and processes a buffer of samples from a temperature sensor before transmitting the result. Peripheral accesses are instrumented with `atomic_start` and `atomic_stop` decorators that prevent JIT checkpoints from occurring in the middle of manipulating a peripheral (lines 1-4 and 6-8).

Frequent power failures complicate peripheral management and make programming batteryless devices challenging because hardware state is lost when the device powers down. Without carefully re-initializing peripherals on reboot, the program may attempt to access a peripheral that is not active or pick up in the middle of a sequence of commands that should have been re-executed from the beginning. At best, the peripheral returns an incorrect result, at worst, the program hangs. Prior work in intermittent systems solves the problem of restoring peripheral state after a power failure [33, 41, 220] and ensuring peripheral activation sequences execute atomically [176, 253]. Figure 2.5b shows a trace of the temperature sensing program executing intermittently. The program relies on JIT checkpoints to preserve progress and is instrumented with peripheral restoration and atomic blocks. If power fails during an atomic block, 2, the device reboots from the start of the block, 3, and



```
main(){ ...
ATOMIC_START;
  temp_init();
  get_temps(buff);
ATOMIC_END;
avg=calc_avg(buff);
ATOMIC_START;
  radio_send(avg);
ATOMIC_END;
        ...
```

```
on_reboot()
{restore_log();
restore_periph();
restore_chkpt();}
```

(a) Device and app.

```
  ...
ATOMIC_START;
  temp_init();
  ~~get_temps(buff);~~
on_reboot(){...}
ATOMIC_START;
  temp_init();
  get_temps(buff);
ATOMIC_END;
~~avg=calc_avg(buff);~~
on_reboot(){...}
avg=calc_avg(buff);
ATOMIC_START;
  radio_send(avg);
ATOMIC_END;
```

(b) Intermittent Execution.

Figure 2.5: Batteryless Device Example Execution. A batteryless device supports peripherals, even though it experiences frequent power failures, using software support. The example code, shown left, executes intermittently on the right with JIT checkpoints to preserve progress. After each power failure (red, dashed lines), and corresponding restarts (green dashed lines), the sensor's operating mode is restored and the program resumes from the most recent checkpoint.

restores peripherals to their states when the block first began. At a Just-In-Time hardware checkpoint, 4, the program and peripheral state are persisted just before a power failure, and on reboot, 5, both are restored precisely.

KARMA [41], RESTOP [220], and Sytare [33] are peripheral restoration systems that dynamically track updates to peripherals so that the peripheral states can be restored on reboot. The major difference between the systems is the peripheral initialization policy after a reboot. RESTOP and Sytare both restore the peripheral state exactly as it was recorded at power failure. KARMA uses lazy initialization to postpone a peripheral's initialization until it is accessed, unless the peripheral is involved in an asynchronous event, e.g. a peripheral is set up to trigger an interrupt when new data are available. KARMA requires peripheral drivers to annotate peripheral operating mode changes (e.g. a peripheral transitions from "off" to "active" during initialization) so that KARMA may replay the changes on reboot.

Tools like peripheral restoration and atomic blocks pave the way for application developers to build complex applications on batteryless devices that process data captured from their surroundings.

2.5 Hardware/Software Co-Design for Peripherals

To concretely demonstrate the lengths to which systems have gone to satisfy application constraints set by peripherals, we present a deep dive into prior work building Capybara [57]. Capybara is a batteryless sensing and computing platform that uses hardware-software co-design to map peripheral requirements to a reconfigurable, hardware, energy buffer, shown in Figure 2.6.

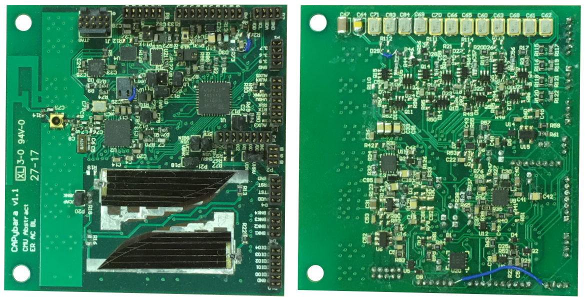


Figure 2.6: Capybara hardware prototype. The MCU, and peripherals are on the front side (left), and the power system with reconfigurable capacitor banks is on the back (right).

2.5.1 Peripheral Constraints

The need for Capybara comes about because peripherals in an application place a wide range of constraints on the energy-harvesting power system of the device. *Energy capacity constrained* tasks, such as sending a radio transmission, must complete atomically and require a minimum energy storage capacity in the power system. *Temporally-constrained* tasks require energy to be available on-demand to react to an external event. However, a fixed power system cannot serve all types of tasks at once. Energy storage capacity determines both energy availability and recharge time for a device. Energy availability determines whether a capacity-constrained task will complete atomically or fail given insufficient energy. Recharge time determines whether a temporally-constrained task will execute reactively because the system is powered off and unresponsive during recharge. Low capacity supports reactive tasks with a short recharge interval, but is insufficient for large atomic tasks. High capacity supports large atomic tasks, but causes long, inactive recharge intervals which compromise reactivity.

Figure 2.7 illustrates how fixed energy buffering fails to meet peripherals' demands. The application attempts to reactively collect a time series of 15 sensor samples and transmit the data by radio. The figure shows how stored energy buffer voltage varies with time when the application executes with two different capacities. With a small energy buffer (left), the application collects sensor samples reactively, with short recharge periods between sampling bursts. However, this system buffers insufficient energy to transmit by radio, and fails. With a large energy buffer (right), the application buffers can transmit, but it fails to sample the sensor reactively because it has long recharge times. What's worse is that tasks may have both temporal and capacity constraints– consider a radio transmission that happens in response to an interrupt. The task requires a large quantity of energy available immediately, which is simply not possible with a fixed energy buffer.

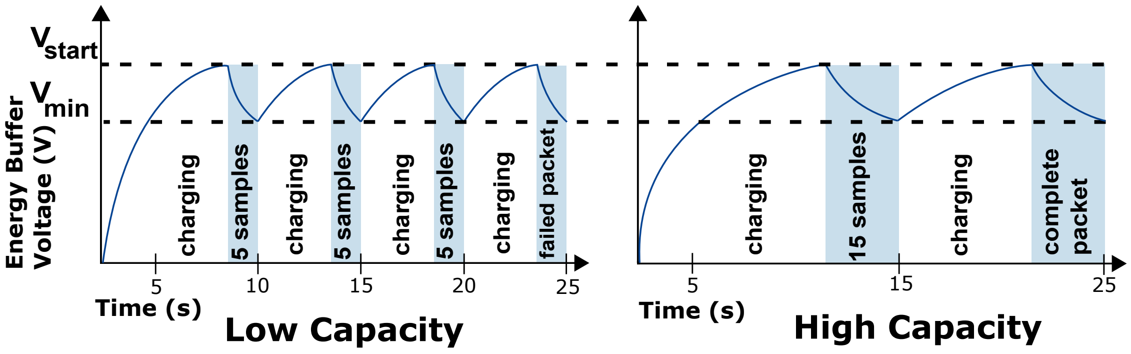
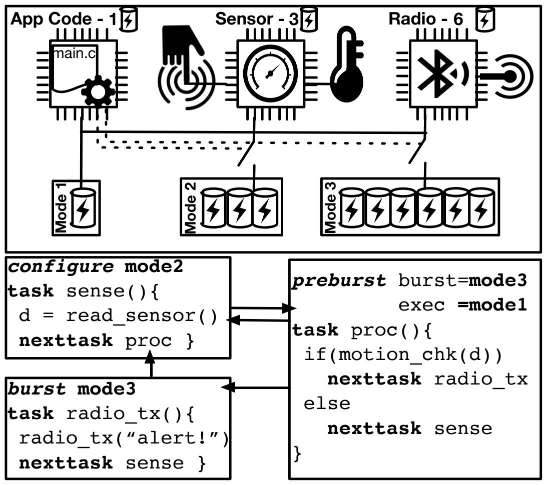


Figure 2.7: Execution with a low or high fixed capacity buffer. Small energy buffers support frequent sensing, but not high energy operations. Large energy buffers support high energy operations, but are powered off for long periods of time while recharging.

2.5.2 Capybara Design

To overcome the shortcomings of a fixed energy buffer, Capybara develops a reconfigurable energy buffer using persistent switches that allow the MCU to change the size of the energy buffer dynamically. The software interface to Capybara allows programmers to specify an energy *mode* for an application task and the underlying runtime reconfigures the buffers to provide the corresponding bank size. Capybara also supports applications with both temporal and capacity constraints by allowing programmers to pre-allocate a *burst* of energy that the MCU can spend in response to an unpredictable event. Figure 2.8 shows a high-level system overview of Capybara running an application (top) and the corresponding task-based program with energy mode annotations(bottom).



```
configure mode2
task sense(){
 d = read_sensor()
 nexttask proc }
```

```
burst mode3
task radio_tx(){
 radio_tx("alert!")
 nexttask sense }
```

```
preburst burst=mode3
         exec =mode1
task proc(){
 if(motion_chk(d))
   nexttask radio_tx
 else
   nexttask sense
}
```

Figure 2.8: Overview of Capybara. The platform has resources for computation, sensing, and communication. An example program has tasks annotated with energy mode requirements.

The programmer annotates a task with parameterized keywords to associate the task with an *energy mode* that describes the capacity and temporal constraints of the task. The `config` (`mode`) annotation indicates that the task should execute with the configuration of the hardware energy storage reservoir that corresponds to the identifier `mode`. The Capybara API includes `burst` and `preburst` annotations to support asynchronous tasks. A task annotated with `burst` (`mode`) requires the specific (possibly very large) amount of energy of the energy mode `mode` at a time in the future that is unpredictable. Just before a `burst` task executes, the runtime system re-activates the energy banks that implement the `mode` configuration and that had been charged ahead of time (by the mechanism explained next), and *immediately* begins executing the `burst` in its declared mode `mode`. To charge a `burst` task's mode ahead of time, the programmer annotates a task preceding the `burst` task with the `preburst` annotation. The pause to charge for the `burst` occurs before the `preburst` task, well in advance of the time critical `burst` task.

Giving the programmer tools to reconfigure the energy buffer in response to temporal and capacity constraints is critical to support peripheral-centric applications. Capybara enabled accuracy improvements in event detection applications of 2x-4x over fixed systems. Further, the `preburst`/`burst` annotations support reactive applications that fail without the ability to store energy for future use. Using Capybara, the response latency was kept within 1.5x of a continuously power baseline. Overall, Capybara demonstrates the importance of hardware/software co-design and the need to design runtime systems around peripherals. Both of these themes appear throughout this thesis.

Chapter 3

Concurrency Control for Task-Based Intermittent Execution

The initial work defining intermittent execution allowed for batteryless systems to run more complicated programs than the simple, one-at-a-time sensing operations [44] where CRFIDs began. With support for complex applications comes the need for language support for reactive processing, but early work in intermittent computing did not define support for concurrent accesses to shared memory. Prior work addressed input/output (I/O), ensuring that computations were *timely* in their consumption of data collected from sensors [57, 111, 284]. However, no prior work on intermittent computing provides clear semantics for programs that use *event-driven concurrency*, handling asynchronous I/O events in interrupts that share state with transactional computations that execute in a main control loop. The idiomatic use of interrupts to collect, process, and store sensor results is very common in embedded systems. The absence of this event-driven I/O support in intermittent systems is an impediment to developing batteryless, energy-harvesting applications.

Combining interrupts and transactional computations in an intermittent system creates a number of unique problems that we address using new system support. First, an interrupt may experience a power failure while updating persistent, shared state, leaving the state inconsistent on reboot. As Section 3.2.1 shows, the inconsistent shared state is likely to remain inconsistent because it is unintuitive to checkpoint and restart an event-driven interrupt's execution after a power failure. Second, task-based intermittent execution models assume that tasks will repeatedly attempt to execute *idempotently*, allowing them to selectively buffer data and commit it when a task ends[55, 113, 168, 174, 269]. An unmoderated interrupt may cause a task's re-execution to be non-idempotent, violating the underlying assumption of task-based intermittent execution systems that allows only selectively buffering state. Consequently, these prior approaches may lose updates or produce inconsistent state in an intermittent execution. An appealing alternative is to disable all interrupts during task execution, with behavior like TinyOS atomics [92, 159]. However, unlike the small amount of code typically protected by TinyOS atomics (e.g., synchronization), intermittent execution requires all code to be in a task; disabling interrupts during any task blocks interrupts for most of a program's execution.

This chapter presents Coati[1], which adds concurrency control for event-driven I/O to an existing task-based intermittent programming and execution model that does not support interrupts. The key contribution of Coati is to define an execution model that safely serializes atomic, transactional computations with concurrent, event-driven interrupts during an intermittent execution. Borrowing from prior work on event-handling in embedded operating systems (OS) [92, 159], Coati defines *events* as shown on the right of Figure 3.1, which are regions of code that atomically process I/O and occur asynchronously. Borrowing from prior work on intermittent systems [55, 111, 174], as well as embedded OS [92, 159], Coati defines *tasks*, which are regions of code that are atomic with respect to power failures and atomic with respect to events. Coati borrows from prior work on transactional memory [37, 102, 106, 107, 187, 237] defining *transactions*, which allow sequences of multiple tasks to execute atomically with respect to events. Coati's support for events and transactions is the main contribution of this work. Coati provides the critical ability to ensure correct synchronization across regions of code that are too large to complete in a single power cycle. Figure 3.1 shows a Coati program with three tasks contained in a transaction manipulating related variables `x`, `y`, and `z`, while an asynchronous event updates `x` and `y`. Coati ensures atomicity of all tasks in the figure, even if any task individually is forced to restart by a power failure.

This work explores the design space of transaction, task, and event implementations by examining two models that make different trade-offs between complexity and latency. Coati employs a *split-phase* model that handles time-critical I/O imme-

[1]**C**oncurrent **O**peration of **A**synchronous **T**asks with **I**ntermittence

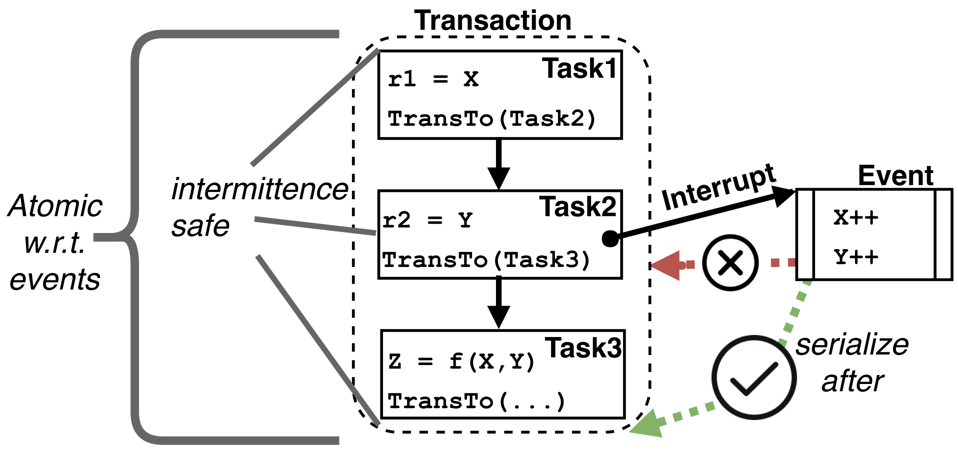


Figure 3.1: Coati program. The program contains three tasks encapsulated in a transaction and an asynchronous event. The event cannot violate the atomicity of the transaction.

diately in a brief interrupt handler, but defers processing the interrupt's result until after the interrupted task or transaction completes, ensuring the task or transaction remains atomic. We also examine an alternative *buffered* model that fully buffers all memory updates made in a transaction and immediately processes events, but on a memory conflict between an event and transaction the event's memory effects are discarded. In contrast, Coati's split-phase model is efficient, requiring neither full memory buffering nor conflict detection for transactions and events.

We prototyped Coati as a set of extensions to the C language and a runtime library that ensures safe, intermittent operation while supporting tasks, events and transactions. We evaluated Coati on a set of benchmarks taken from prior work [174], running on a real intermittent energy-harvesting system [57]. The data reveal that Coati prevents incorrect behavior of event-driven concurrent code in an intermittent execution. In contrast, we demonstrate that an existing, state-of-the-art task-based intermittent system produces an incorrect, inconsistent result in nearly all cases.

3.1 Motivation

This work is the first to simultaneously address the challenges of concurrency control for event-driven I/O and atomicity for computations in an intermittent system. Event-driven I/O faces the challenge of managing asynchronous interactions between a program's main computational work loop and operations in interrupts. Intermittent execution faces the challenge of spanning a program's execution across unpredictable power failures, while ensuring that memory and execution context remain consistent. Coati is motivated by the combination of these two challenges: handling asynchronous I/O in interrupts during a consistent, progressive intermittent execution. Together, these challenges lead to fundamental correctness problems that are not well-addressed by existing hardware or software systems for intermittent computing [30, 31, 55, 113, 170, 174, 269, 284].

3.1.1 Concurrency in Embedded Devices

Embedded systems in cyber-physical applications must asynchronously interact with unpredictable stimuli from their environment often using peripherals to perform I/O. Embedded systems typically handle such asynchronous operations using *interrupts*. An interrupt is a signal triggered by an asynchronous event that is moderated by hardware and eventually delivered to a software *interrupt service routine* (ISR). An ISR can perform application-specific event-handling operations, including interacting with peripherals (i.e., the one that triggered the interrupt), performing arbitrary computation and manipulating variables. An asynchronous ISR preempts the program's main thread of control, and may concurrently (although not in parallel) access program state. After an ISR completes, control returns to the point in the program at which the preemption occurred.

Event-driven concurrency of interrupt handlers and program code requires embedded software to synchronize accesses to shared data. Code may synchronize data using mutex locks, reader-writer locks, or semaphores to establish critical regions that atomically read or update data. TinyOS [92, 159] allows specifying `atomic` operations that, in effect, disable interrupts for their duration. One use of `atomic` is to synchronize direct access to shared data by an interrupt and a program. While atomic program operations execute, interrupts are disabled, instead of immediately being handled. TinyOS-style `atomic`s also allow building synchronization primitives, which may be useful when an application cannot disable interrupts for a long time (i.e., to remain responsive). A key problem addressed by Coati is that task-based intermittent programming systems do not support interrupts and existing concurrency control mechanisms do not gracefully handle intermittent operation.

3.1.2 Benefits of Interrupts in Intermittent Systems

Event-driven interrupts are crucially important for intermittent systems applications. Recent work has demonstrated the value of local DNN inference on intermittent devices to enable complex, event-driven applications [98]. Without interrupts, event-driven applications must alternate between processing event data and polling for new events. Computationally intensive event processing causes long unresponsive periods because computation monopolizes the processor. The device will not observe a new event until it processes an older one. Intermittent execution increases the length of the unresponsive periods because the application must frequently wait to recharge after depleting its buffered energy.

Figure 3.2 shows data from a high level software simulation of an event-driven image processing application that captures bursts of events (e.g. a pack of coatis passing by a wildlife camera). The simulation compares the fraction of events captured over an hour using interrupts (the int-* lines) versus polling (the poll-* lines) for a continuously-powered (*-CP) and energy-harvesting (*-EH) system. A burst of 5 events (e.g. coatis in close proximity) occurs with an expected interarrival time of 3 seconds. An event lasts 1.2 seconds, twice the device's recharge time (i.e., recharging does not cause missed events). Our simulation models powered-on, recharge, and data collection times using measurements of the Capybara platform with its onboard MCU at 8MHz [57]. The simulated workload models the intermittent MNIST implementation from prior work [98, 155].

Each event requires 6 seconds of computation while the device is powered on, and the device may enqueue up to 16 events (a 16KB event queue). We assume a low power camera with a separate energy buffer and harvesting source [109]. The simulation shows that even with continuous power, polling captures less than 60% of the events because the system misses events that occur while processing prior events. The effect is exacerbated under harvested energy because the time to recharge extends the time to process the first event in the burst, and prevents the system from capturing any additional events. Introducing interrupts allows the system to capture all events with continuous power. Interrupts on harvested energy (int-EH) converges to capturing 100% of events once the burst interarrival time is long enough to prevent the event queue from saturating. The data emphasize the need for interrupts in intermittent systems.

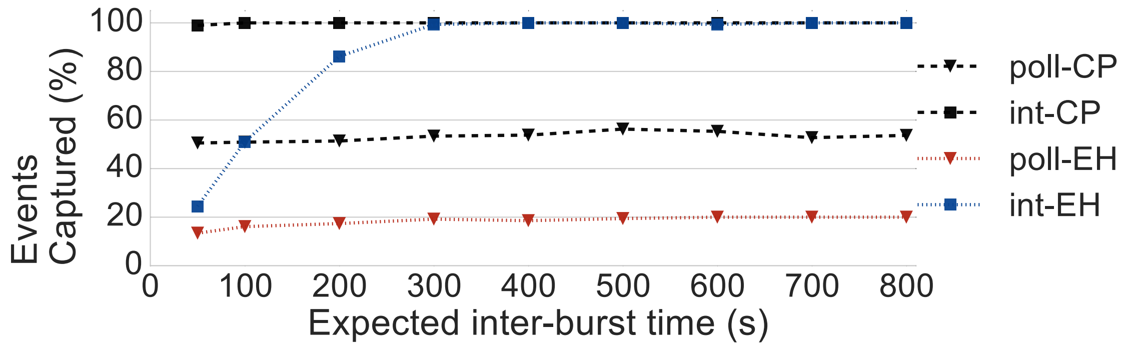


Figure 3.2: Interrupts support bursts of events. Using interrupts, the simulated energy harvesting device (int-EH) outperforms the continuously powered, polling baseline (poll-CP) if there is time to recharge between bursts

3.2 The Challenge of Intermittent Event-Driven Concurrency

Naively combined, existing solutions for intermittence and event-driven concurrency can cause incorrect and unintuitive behavior. Event-driven, concurrent execution is incorrect in the presence of intermittence: simply using TinyOS-style `atomic` concurrency control in an intermittent system behaves incorrectly when power fails. Additionally, both static checkpointing and task-based intermittent execution can be incorrect or inefficient in the presence of interrupts.

3.2.1 Interrupts + Intermittent Operation

Event-driven code behaves incorrectly if power fails during an interrupt. The key problem is that even using `atomic` operations, if power fails during an ISR, the ISR may have only partially updated a multi-byte data structure. On reboot, intermittent execution resumes in the most recently executing *task* or from the most recent checkpoint. If the program accesses the data partially updated by the ISR, the program behaves incorrectly. An (unintuitive) alternative approach is to restart execution after a power failure in the context of the ISR. The problem with restarting in an ISR after a power failure is that important device state may be unavailable on reboot (because a peripheral reset). Moreover, the device may have been inoperational for an arbitrary duration, violating timeliness constraints on the ISR [111].

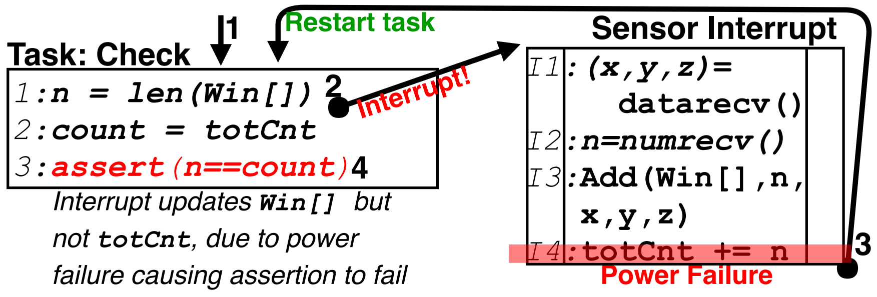


Figure 3.3: The "Interrupt Interrupted" problem. The interrupt updates `Win[]`'s length without updating `totCnt`, leaving the two inconsistent.

Figure 3.3 illustrates this "Interrupt Interrupted" problem with example task-based code. The figure adds a task to the program that checks the consistency of the size of `Win[]` and `totCnt`, which should always be equal. The interrupt adds new entries to `Win[]`, but power fails before it updates `totCnt`. When control returns to the task, the assertion fails because the data are inconsistent.

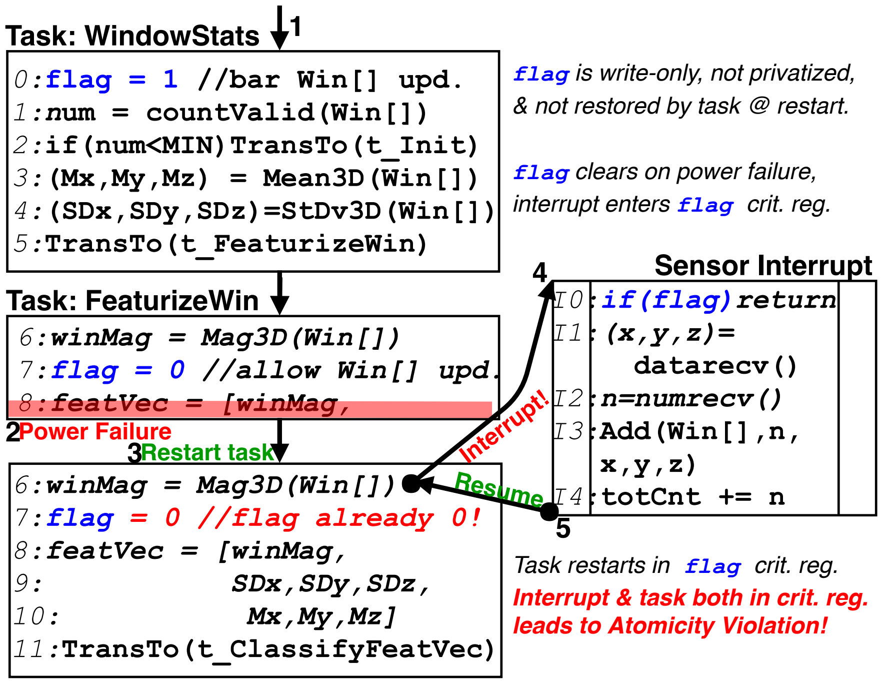
The ISR can produce partial updates *even with* the privatization analysis of a task-based intermittent execution framework [174, 269], treating the ISR as a task. Privatization analysis identifies data to *privatize* to a task. A task buffers updates to privatized data and commits updates only when the task completes. Privatization analyses in intermittent execution frameworks assume that a task repeatedly executes until successfully completing. To reduce buffering and commit cost, the analysis only privatizes data that are read, then written by the task (i.e., finding WAR dependences). Such "WAR"-based privatization prevents data written by a failed attempt to execute the task from being visible to a read in a re-execution of that task. Accesses to data not involved in a WAR dependence *directly update memory*. Privatization analysis works correctly for sequential intermittent programs because they always re-attempt a failed task until it completes. Existing privatization analyses [113, 174, 269] are incorrect with interrupts: if power fails during an ISR that only writes to a multi-byte data structure, a partial update is unbuffered and made visible to the continuation of the interrupted task after power resumes.

3.2.2 Synchronization + Privatization

Synchronization is also complicated by intermittent execution. TinyOS-style `atomics` are useful for building synchronization primitives, such as flags and locks, enabling synchronization operations to perform read-modify-update operations that an ISR will not interrupt. Often a program cannot disable interrupts during a long region of code, barring use of `atomic` and requiring use of such a synchronization primitive. An intermittent task may leave a critical region by updating a synchronization variable (unsetting a flag or releasing a lock). If the intermittent task system does not privatize the synchronization variable (i.e., because it is not read then written by the task), the update directly modifies main memory. If power fails after a task leaves a critical region, execution resumes from the start of the task *which may be inside the critical region.* The problem is that the update made to the synchronization variable to *leave* the critical region remains in memory. After the task restarts in the critical region, an ISR may successfully enter its critical region, leaving both the task and the ISR in the critical region, which is incorrect. This problem also occurs in a checkpointing system if a programmer uses a lock to span multiple static checkpoints. If power fails after the code has released the lock and before it has reached the next checkpoint, on reboot the scheduled code and the ISR will both be able to enter the critical region.

Figure 3.4 shows the "False Flag" problem, which illustrates how flag synchronization is complicated by intermittent operation. The `WindowStats` and `FeaturizeWin` tasks compute related properties of `Win[]`. Both should see the same values in `Win[]`, requiring them to be in a critical region protecting `Win[]`. The flag `flag` controls access to the critical region, preventing the interrupt from entering while `WindowStats` and `FeaturizeWin` are executing. It is reasonable to use a flag across multiple tasks, because one very long task would exhaust the device's energy supply, impeding progress.

The execution sets `flag` and proceeds through `WindowStats`. The first attempt to run `FeaturizeWin` fails just after clearing `flag`. The task restarts *without* restoring `flag=1` because `flag` is write-only; privatization does not restore write-only data on reboot [168, 174, 269]. After restart, the task is *in* the critical region. The interrupt immediately fires, checks `flag`, sees it clear, and also enters the critical region. The interrupt's updates to `Win[]` violate the critical region's atomicity, leading to an inconsistency between values computed by `WindowStats` and `FeaturizeWin`.



```
flag = 1 //bar Win[] upd.
num = countValid(Win[])
if(num<MIN)TransTo(t_Init)
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv3D(Win[])
TransTo(t_FeaturizeWin)
```

```
winMag = Mag3D(Win[])
flag = 0 //allow Win[] upd.
featVec = [winMag,
```

```
winMag = Mag3D(Win[])
flag = 0 //flag already 0!
featVec = [winMag,
              SDx,SDy,SDz,
               Mx,My,Mz]
TransTo(t_ClassifyFeatVec)
```

```
if(flag)return
(x,y,z)=
   datarecv()
n=numrecv()
Add(Win[],n,
   x,y,z)
totCnt += n
```

`flag` is write-only, not privatized, & not restored by task @ restart.

`flag` clears on power failure, interrupt enters `flag` crit. reg.

Task restarts in `flag` crit. reg.
Interrupt & task both in crit. reg. leads to Atomicity Violation!

Figure 3.4: The “False Flag” problem. When power fails after `FeaturizeWin` clears `flag`, the task and the interrupt are both in the critical region, violating atomicity.

Privatization causes a symmetric "False Flag" problem when the interrupt needs to send a signal to the task. A common programming pattern in embedded systems is to wait in the main thread until ISR code updates a shared variable that signals the main thread to continue. For instance, the main thread might wait until a signal variable is set that indicates new data has arrived, then the main thread will process the data and clear the signal. However, in an intermittent execution, if the read and the clear of the signal variable happen in the same task, privatization redirects all accesses of the variable to a private copy. Since the task only accesses its private copy, writes to the variable in the ISR are not visible until the task completes or the device powers down and restarts the task.

3.3 Intermittent Interrupts with Coati

Coati is a programming API and runtime that allows a programmer to control event-driven concurrency in a task-based intermittent execution system. Figure 3.5

shows an overview of Coati's use. Like prior intermittent execution systems (Coati builds on Alpaca [174]), Coati asks the programmer to write their program as a collection of *tasks*. A task is a function with no callers that can include arbitrary code and explicitly transfers control to another task. Tasks communicate by accessing global variables stored in persistent memory. Coati tasks are atomic with respect to power failures because Coati buffers a task's updates to memory and commits them on task completion, discarding them on a failure. A task interrupted by a power failure idempotently re-executes until progressing to the next task.

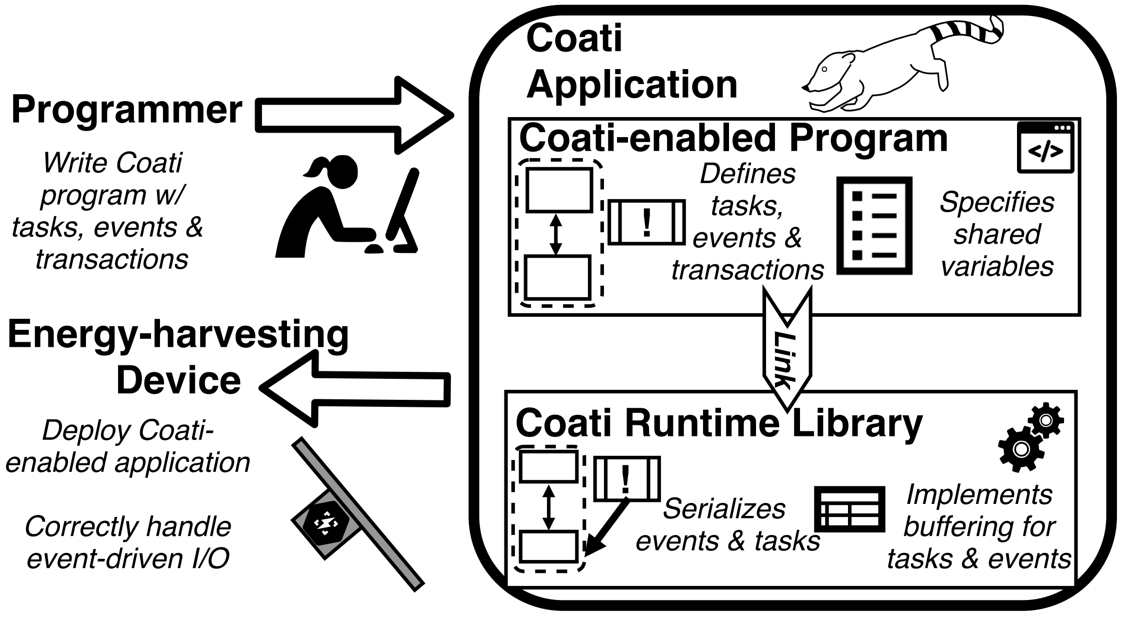

Coati Application
Programmer
Write Coati program w/ tasks, events & transactions
Coati-enabled Program
Defines tasks, events & transactions
Specifies shared variables
Link
Energy-harvesting Device
Deploy Coati-enabled application
Correctly handle event-driven I/O
Coati Runtime Library
Serializes events & tasks
Implements buffering for tasks & events

Figure 3.5: Overview of Coati. The programmer codes using Coati primitives and links the Coati-enabled program to the Coati runtime, handling both intermittence and interrupts so the app executes correctly when deployed.

Coati allows the programmer to specify *events*. An event is a special task that a programmer can use to process an asynchronous interrupt. An event in Coati is similar to an event in TinyOS [92, 159] in that an event may be concurrent with a task and an event may share state with a task. The primary difference between tasks and events is that a task does not explicitly transfer control to an event. Instead, an event is associated with an asynchronous interrupt and invoked automatically by Coati on that interrupt's asynchronous occurrence. Coati also allows the programmer to define a *transaction*, which is a sequence of tasks that execute together atomically with respect to events, while remaining individually atomic (and idempotently restartable) with respect to intermittent power failures. Like TinyOS [159], Coati assumes that events are short and relatively infrequent so that the application can make forward progress. Further, Coati assumes that the programmer correctly balances transaction length with responsiveness requirements for processing events.

To correctly include events in an intermittent execution, Coati must meet several requirements. First, Coati must preserve task atomicity and idempotence despite asynchronous events by serializing tasks' and events' updates to task-shared variables. Second, Coati must support atomic regions that extend beyond one reboot. Finally, Coati should not impose a prohibitive overhead in terms of runtime or memory.

3.3.1 Interaction Between Tasks and Events

The main contribution of this work is defining how Coati's tasks, events, and transactions interact. We first describe task-event interactions using the *split-phase* serialization model used by Coati's final design. Split-phase serialization forces events to serialize *after* an interrupted task completes. Returning to Figure 2.1's example code, Figure 3.6 shows how events and tasks serialize.

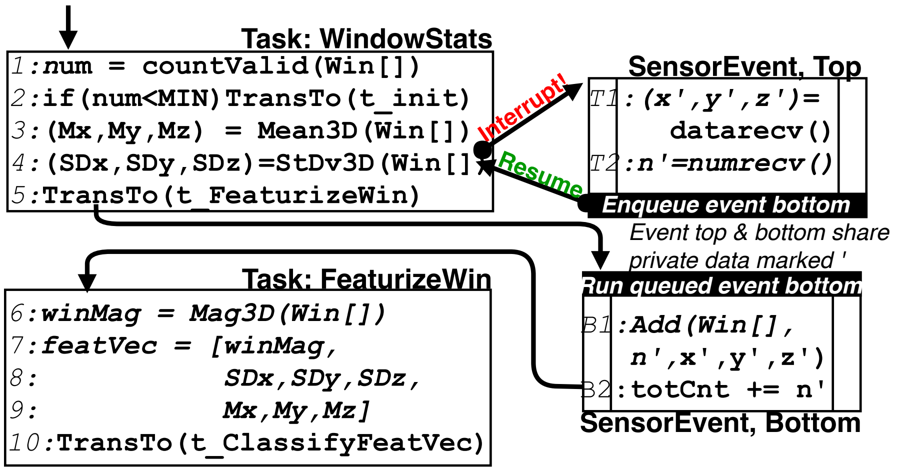

```
Task: WindowStats
1:num = countValid(Win[])
2:if(num<MIN)TransTo(t_init)
3:(Mx,My,Mz) = Mean3D(Win[])
4:(SDx,SDy,SDz)=StDv3D(Win[])
5:TransTo(t_FeaturizeWin)
```
```
SensorEvent, Top
T1:(x',y',z')=
     datarecv()
T2:n'=numrecv()
```
```
SensorEvent, Bottom
B1:Add(Win[],
    n',x',y',z')
B2:totCnt += n'
```
```
Task: FeaturizeWin
6:winMag = Mag3D(Win[])
7:featVec = [winMag,
8:             SDx,SDy,SDz,
9:             Mx,My,Mz]
10:TransTo(t_ClassifyFeatVec)
```

Figure 3.6: Task and split-phase event interaction. Split-phase events are separated into a top half which executes immediately, and a bottom half which runs after the interrupted task commits.

Split-phase Serialization. Split-phase interactions decouple the asynchronous part of an event (i.e., the ISR and peripheral manipulations) and the shared data manipulation associated with the event. A split-phase event has a *top*, which runs asynchronously at the interrupt, and a *bottom* which is scheduled to run after the completion of the task that was interrupted by the top of the event. Similar to tasklets in Linux [162], event bottoms allow an interrupt to defer latency tolerant work and quickly return to the interrupted task. In Figure 3.6, the top of the event interrupts `WindowStats` and executes immediately, while the bottom executes after `WindowStats` completes. After the top of the event completes, execution resumes from the point of the interrupt in the interrupted task, as shown in the figure. The top of a split-phase event is not allowed to access any global, shared state, avoiding the risk of violating a task's idempotence. Instead, the top of the event can privately buffer data to be processed (e.g., sensor data, received radio packets), preparing them for use by the event's bottom. In the figure, private data have a `'`: `x'`, `y'`, `z'`, and `n'` are only accessible by the interrupt's top and bottom. The event's bottom, serialized after the interrupted task, is allowed to access arbitrary state. The event bottom buffers all memory updates, like a task, and atomically commits those updates on completion. The figure shows how the event's bottom adds to `Win[]` the data sensed by the event's top and increments `totCnt`.

Split-phase serialization gives Coati flexibility on power failure. Coati has a configuration option that aborts event bottoms for programs in which an event's bottom must execute in the same operating period as its top. However, Coati can avoid unnecessarily discarding events by repeatedly executing an event bottom if power fails during the event bottom. Because the part of the interrupt that must be timely is likely to occur in the top of the event handler, the bottom should be able to execute safely even with the delay of a power failure.

If multiple split-phase events occur during a single task's execution, Coati allows their event bottoms to *queue* and wait for the task to complete. When a task completes, it commits as usual and processes the event queue, executing queued event bottoms in order.

3.3.2 Multi-Task Transactional Execution

Coati allows the programmer to sequence multiple tasks together into a *transaction*. A transaction is atomic with respect to interrupting events in the same way that an individual task is atomic with respect to an interrupting event. A transaction is *not* atomic with respect to power interruptions, but its constituent tasks individually are.

Applications need multi-task transactions because some operations that should be atomic with respect to interrupting events may individually consume more energy than the intermittently operating device can buffer. If tasks were the only unit of atomicity with respect to events, then a program using such energy-hungry tasks would have to decide: to split the operations across tasks, allowing an interrupting event to violate atomicity, or to include both operations in the same task, preventing the task from completing because it consumes more energy than the device can supply. Multi-task transactions allow tasks to execute sequentially without interruption by an event, eventually committing updates from all tasks.

Coati Transactions. Coati supports split-phase serialization for transactions as shown in Figure 3.7. The figure shows the case where multiple events interrupt during the transaction's execution, each executing its top half and enqueueing its bottom half to execute after the transaction completes. Each pair of event top and bottom shares its own private buffer (distinguished in the figure using `'` vs. `"`). The operation of split-phase serialization with transactions is identical to the task case, except Coati does not execute event bottoms until after the entire transaction completes, as opposed to awaiting only the interrupted task. The simple implementation of transactions in the split-phase model avoid several key drawbacks of the buffered implementation that is discussed in Section 3.5.2.

3.4 Implementation Details

Coati's implementation is a runtime with an API (Table 4.1) for control flow, persistent state access, and synchronization of events and tasks. We assume hardware with byte-addressable non-volatile memory and atomic word writes.

3.4.1 Control Flow

Coati maintains a *program context* that defines the current control state of the program, allowing the system to progress through tasks, enter and exit transactions, and execute and return from events. Tasks are C functions with no return value and no arguments. Programmers use the `next_task` or `tx_next_task` statements to transition between tasks, respectively, both outside and inside of a transaction. To ensure task transitions are robust to power failures, the runtime system explicitly

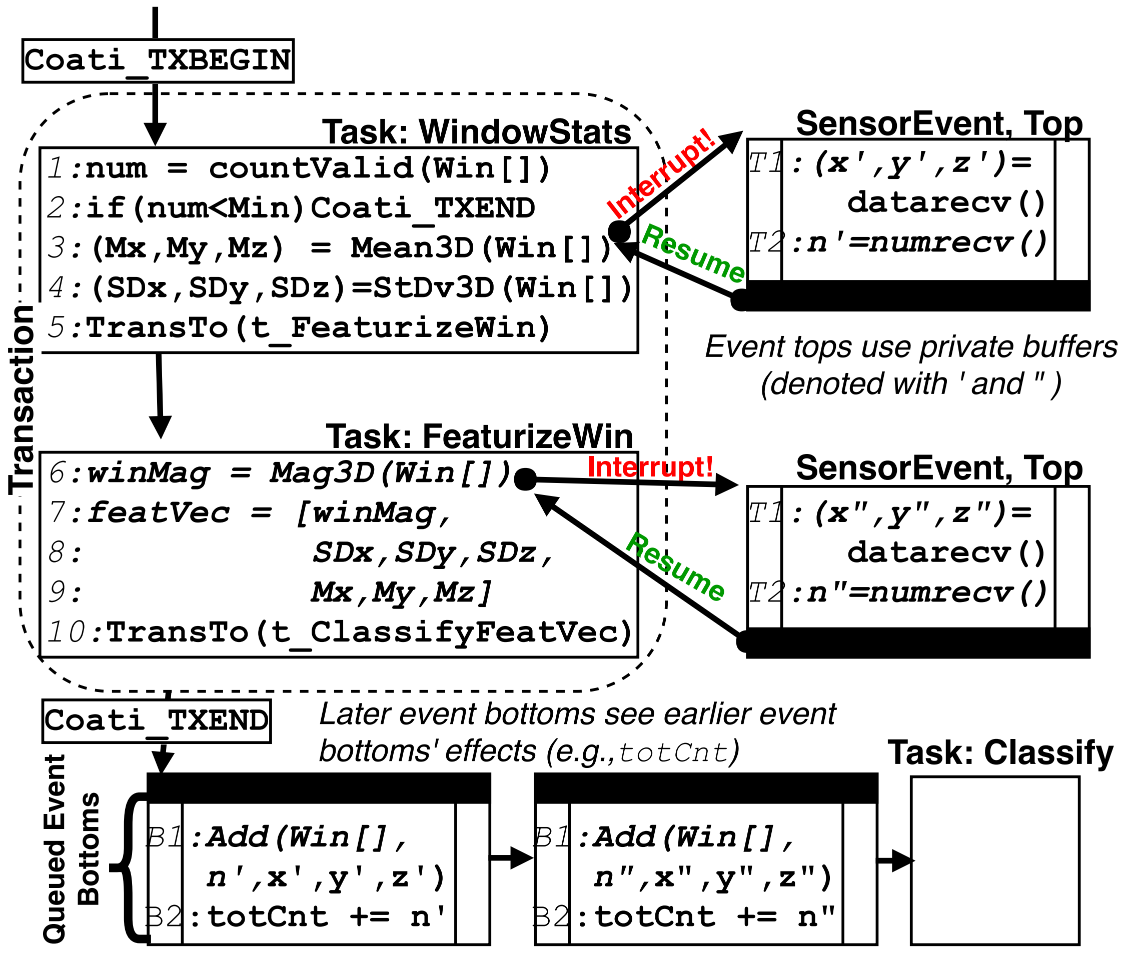

Coati_TXBEGIN
Task: WindowStats
1:num = countValid(Win[])
2:if(num<Min)Coati_TXEND
3:(Mx,My,Mz) = Mean3D(Win[])
4:(SDx,SDy,SDz)=StDv3D(Win[])
5:TransTo(t_FeaturizeWin)
Transaction
Interrupt!
Resume
SensorEvent, Top
T1:(x',y',z')= datarecv()
T2:n'=numrecv()
Event tops use private buffers (denoted with ' and ")
Task: FeaturizeWin
6:winMag = Mag3D(Win[])
7:featVec = [winMag,
8: SDx,SDy,SDz,
9: Mx,My,Mz]
10:TransTo(t_ClassifyFeatVec)
Interrupt!
Resume
SensorEvent, Top
T1:(x",y",z")= datarecv()
T2:n"=numrecv()
Coati_TXEND
Later event bottoms see earlier event bottoms' effects (e.g.,totCnt)
Task: Classify
Queued Event Bottoms
B1:Add(Win[], n',x',y',z')
B2:totCnt += n'
B1:Add(Win[], n",x",y",z")
B2:totCnt += n"

Figure 3.7: A transaction using split-phase serialization. Each time a sensor event occurs, the top half captures data and enqueues a bottom half to execute after the transaction.

maintains a *task context object*, stored in persistent memory. The task context object holds the address of the start of the current task and the current task's *commit state* bit, which indicates whether the task is currently committing. The task context object also holds the context of an ongoing transaction containing the current task and the state of any queued event bottoms that are waiting to execute. Task transition statements update the task context atomically by double buffering the context and swapping the value of the runtime's global `current_context` pointer, which points to the active context. To start a transaction, the programmer inserts the `tx_begin` keyword at the start of the first task in a transaction. `tx_begin` sets the current context's transaction state to active. To commit a transaction, the programmer uses a `tx_next` statement, which atomically commits the updates from the last task in the transaction and redirects control to a specified task.

Coati Event Implementation. Coati's implementation relies on the existing ISR control mechanism and does not require explicitly tracking control transfer between the interrupted task and an event's top half. Instead, the programmer writes the event's top half directly in the ISR. At the start of the event's top half code, the programmer must include Coati's `th_start` primitive. `th_start` ensures that there are resources available to buffer the event's bottom half, aborting the interrupt if there are not. The programmer uses Coati's `th_return` to register an event's bottom half. In contrast to the top half, an event bottom half is a C function with no return value and no arguments and is associated with an event context object. `th_return` takes a pointer to an event context object as an argument and adds that pointer to Coati's *event queue*, which tracks queued bottom halves in order. After an interrupted task or transaction completes, it checks the event queue. If the queue contains any event bottom halves, control transfers to each of them in FIFO order before moving on to the next task.

3.4.2 Memory Access

Coati provides atomic updates by buffering *all* updates to persistent memory in nonvolatile buffers. Each buffer is a statically pre-allocated list of writes to memory (without duplicates). Each entry contains the address, size, and new value for each update. Coati maintains one buffer that is reused for tasks, tasks in transactions, and events because split-phase serialization does not allow an event bottom to run until after the ongoing task or transaction has committed. As we describe below, an update by an event or task of a non-volatile memory location (to a "task-shared" variable [174]) is stored in its respective buffer until commit. Buffers are statically allocated (which is common in embedded systems) and it is an error to access more data in a task, event, or transaction than any buffer can hold.

Writes. To write memory, the programmer uses the `write` primitive. A task or event write performs a linear search in the private buffer for an entry for the variable being written. If the variable's address is not in the buffer, Coati attempts to allocate space in the buffer. Exceeding buffer capacity is a programming error and the programmer

Table 3.1: A summary of the Coati API. `t` is a task, `ev` is an event bottom, `x` a variable, `val` a value, and `type` a C type.

Control Flow	**Data Access**	**Synchronization**
`next_task(t)`	`read(x,type)`	`tx_begin()`
`th_start()`	`write(x,val,type)`	`tx_next(t)`
`th_return(ev)`		`th_write(x,val)`
		`bh_read(x)`

should specify that Coati should use larger buffers and re-compile. After allocating space in the buffer, Coati writes the update's address, size, and value to the buffer.

Reads. To read memory, the programmer uses the `read` primitive. The runtime first linearly searches the private buffer for an updated version of the variable. If the search succeeds, the read returns a reference to the buffered value. If the search fails, then the read returns a reference to the value stored in main memory.

Split-Phase Accesses. In Coati's split-phase serialization model, events are split into top halves and bottom halves (Section 3.3.1). The top half of an event can share data with the bottom half of an event through fields in a Coati event context object that the programmer specifies using `evt_var`. Each field is automatically, statically replicated by Coati a number of times equal to the maximum number of enqueued event bottoms. The top half can store to one of these shared fields using `th_write`. The bottom half can load from one of these shared fields using `bh_read`. When an event bottom accesses a field shared with its event top, Coati automatically maps the field to the correct replica based on the event bottom's position in the event queue. All field data are statically allocated and the memory overhead of queue and fields is programmer-configurable; we used a queue of 16.

3.4.3 Commit

Coati uses two-phase commit to atomically commit buffered updates. Commit begins when execution traverses a programmer inserted `next_task` or `event_return` statement, which ends a task or event respectively. Ending a task or event sets its commit state bit and prepares the task context object for the next task that will execute. The runtime points the `current_context` pointer at this new context and begins writing the buffered updates from the just-completed task or event-bottom to memory. During commit, the runtime uses a non-volatile counter to track the number of buffer entries remaining to commit, decrementing the counter only after the write is complete to ensure that all entries correctly commit, despite power failures.

3.5 Buffi: A Buffering-Based Alternative

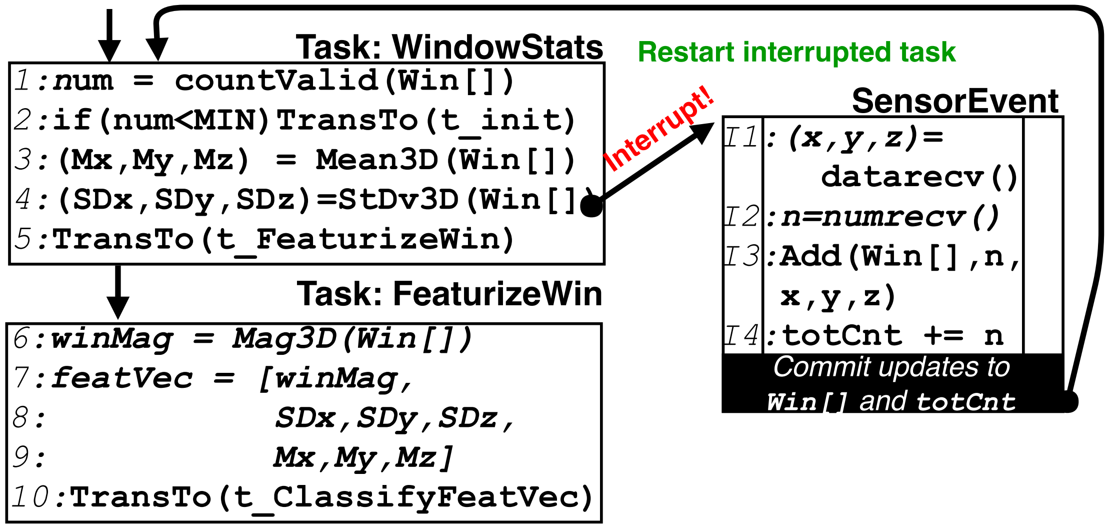
To explore the design space of systems supporting intermittent execution and transactional concurrency, we developed an alternative implementation called Buffi. Coati uses split-phase serialization to order events with tasks and transactions, Buffi, in contrast, uses *buffered serialization*. Buffi buffers all event, task, and transaction state to serialize concurrent updates to shared memory. Buffered serialization necessitates several design and implementation changes.

3.5.1 Buffering and Serialization

Buffi tasks and events buffer *all* shared memory updates. As a result, each event may be written as a single block of code that executes immediately after its associated interrupt fires. Events may perform timely operations and manipulate task-shared variables. In Figure 3.8, the event reads new data from the sensor and updates `Win[]` and `totCnt` immediately after it is triggered by the interrupt. The event commits the updates when it completes. After an event commits its updates to memory, control resumes from *the beginning* of the interrupted task and execution continues; in the figure, the event completes and `WindowStats` restarts from the beginning. If power fails during a Buffi event, Buffi discards its updates and does not attempt to re-execute it.

3.5.2 Buffi Transactions

To support buffered-serialization for transactions, Buffi introduces a transaction buffer. Buffi tasks in a transaction consecutively commit their updates to a transaction-private *commit buffer* that is distinct from main memory. On completion, the transaction commits this transaction commit buffer to main memory, making the transaction's tasks' updates visible to subsequent tasks and events. Buffi events may preempt transactions, but only the interrupted *task* must be restarted. Completed



```
num = countValid(Win[])
if(num<MIN)TransTo(t_init)
(Mx,My,Mz) = Mean3D(Win[])
(SDx,SDy,SDz)=StDv3D(Win[])
TransTo(t_FeaturizeWin)
```

```
winMag = Mag3D(Win[])
featVec = [winMag,
            SDx,SDy,SDz,
            Mx,My,Mz]
TransTo(t_ClassifyFeatVec)
```

```
(x,y,z)=
   datarecv()
n=numrecv()
Add(Win[],n,
 x,y,z)
totCnt += n
```

Figure 3.8: Task & buffered event interaction. Buffered serialization forces tasks to restart after an event. If the event interrupts a transaction, only the interrupted *task* must restart.

tasks in the transaction do not need to be rerun. However, events' updates must serialize *after* the entire preempted transaction. To preserve the atomicity of the transaction, an event that preempts a transaction commits its updates to a private commit buffer on completion, rather than committing directly to memory.

Before committing buffered event updates, Buffi performs *conflict checking*, which ensures that the event did not read data that the transaction wrote. If an event that executed during the transaction read from a memory location that a task in the committing transaction wrote, the event would not see the value updated by the transaction because the update would be buffered. Reading this stale value is inconsistent with the event's serialization after the task, so Buffi discards the events' buffered updates.

3.5.3 Buffi Implementation

Buffi's implementation is similar to Coati's, but is generally more complex. The major differences arise from maintaining additional state to allow for concurrent access to task-shared variables by events and transactions.

Buffi Events. Buffi statically allocates an *event context object* for each programmer-defined event function. The event context object holds a pointer to the start of the event and its commit state. To trigger an event in response to an interrupt, the programmer inserts an `event_handler` call in the ISR after any device-specific code required to clear the interrupt that was triggered. `event_handler` sets Buffi's `return_context` pointer to point to the interrupted task's context, and transitions to the event by setting the `current_context` pointer to point to the triggered event's context. The programmer places an `event_return` statement at the end of the event, which returns control to the stored `return_context`.

Buffi Memory Accesses. Buffi maintains three statically allocated buffers: the *task buffer*, the *event buffer* and the *transaction buffer* to ensure isolation between concurrent updates to task-shared variables. In Buffi, a task in a transaction writes its buffered updates into the transaction's buffer when the task commits. While a transaction executes, a single event buffer persists across events, allowing events to see updates written by previous events. In tasks and events outside an active transaction, Buffi accesses task-shared variables the same way as Coati (as described in Section 3.4.2). If a read occurs in a task within a transaction, the runtime first searches through the task's buffer, then the transaction's buffer, before reading the value from main memory. In Coati, there is no transaction buffer; the read immediately returns the value from memory. Buffi also tracks the set of writes performed by a transaction and the set of reads performed by events for use in conflict detection. A write in a Buffi transaction updates its write set, adding the address of the written location. A read in a Buffi event first checks if there is an active transaction, and then updates the event's read set. Recall that Coati need not track write or read sets.

Buffi Commit. For tasks and events that occur when no transaction is active, commit follows the procedure described in Section 3.4.3. To commit a transaction, the transaction must first commit updates made by the last task in the transaction to its transaction buffer. Next, Buffi compares the set of addresses that are updated in the transaction buffer to the set of addresses that were read by any event. If the two sets of addresses overlap, Buffi detects a conflict and discards the event buffer. Next, Buffi writes the updated values in the transaction buffer back to memory. Last, if there was no conflict between an event and the transaction, Buffi commits the event buffer, and continues to the next task after the transaction. Recall that Coati simply commits the updates from the last task in the transaction, and then unconditionally processes the event queue.

3.5.4 Buffer Design

To study the effect of buffer design on Buffi's memory access latency, we implemented the transaction buffer both as a linear buffer and alternatively as a fixed size, chaining hash tables. The hash table design results in speedup when the transaction commit list is long because linear search in a long list is slower than hash lookup. Table 3.2 shows the average and maximum number of entries committed at the end of each task and transaction in several benchmark applications (described in Section 3.6). The data show that a simple, linear buffer often works well because the average number of entries committed at the end of a task tends to be small. The data also show that a hash table is the best option for RSA and CF (full evaluation details are in Section 3.6).

Table 3.2: Commit Statistics. The average and maximum number of entries committed at the end of each task and transaction in benchmark applications.

App	BC	AR	RSA	CEM	CF	BF
Task Avg	3	2.4	4.8	9.4	1.6	–
Task Max	3	6	18	194	135	–
Tx Avg	4.3	6.2	97.7	29.4	147	–
Tx Max	5	10	98	32	149	–

3.6 Evaluation

We evaluated Coati using applications from prior work on a real energy-harvesting device and directly compared to a state-of-the-art intermittent computing system. Since no prior systems correctly support concurrent interrupts in an intermittent execution, we compare Coati's split-phase transactions to three additional concurrency control strategies. Our evaluation shows that Coati avoids failures permitted by existing systems and does so with low runtime overhead and reduced programming effort, permitting responsive event-driven applications.

3.6.1 Benchmarks and Methodology

We evaluated Coati using the Capybara hardware platform [57]. We used a collection of full applications from prior work [174], modified to use event-driven I/O. The applications collect input, process data and communicate results. To make experiments repeatable, we emulate peripherals with logged data. Like TinyOS [159], we assume that events are short and that they are triggered relatively infrequently. To emulate event arrival, we trigger events via a GPIO pin driven by a continuously powered Arduino [18] for 20ms pulses with an interarrival time drawn from a Poisson distribution with $\lambda = 100ms$ [259]. We use the same event arrival emulation parameters for all benchmarks. We measured runtime using a logic analyzer to capture GPIO pulses at the start and end of each run. We use an attenuated bench supply as a harvested-energy source providing under 10mA, like prior work [57].

We evaluate *six* configurations: Ideal (which does not complete on harvested energy), Alpaca (which breaks with interrupts on harvested energy), Coati, Buffi, and two additional comparisons Atomic, and Hand-Op. We compare against Alpaca because, at the time of writing, it is the fastest task-based intermittent computing model that supports arbitrary computation. Alpaca avoids the channel management overhead of Chain [55], and allows loops in the task graph unlike Mayfly [111]. To provide idempotent task re-execution in the case of power failures, Alpaca buffers only the task-shared variables involved in write-after-read dependences. At the end of each task, Alpaca commits the buffered variables to memory. Ideal runs the same application code as Alpaca, but the runtime has been modified to remove all buffering and commit code. Ideal *fails* on harvested energy, but on continuous power it represents a task-based system with the minimum possible overhead. Atomic and Hand-Op use fully buffered tasks, like Buffi, but neither uses transactions. Atomic masks interrupts during critical regions, representing a naive solution to multi-task atomicity. Hand-Op uses hand-optimized code to synchronize tasks and events, demonstrating the programming effort required without Coati.

We modified the applications to use the Coati API to handle events, preserving their task decomposition [174]. **BC:** Bitcount (BC) counts bits set in an array using various algorithms. Its event changes an array index and a transaction ensures each algorithm sees a consistent array. Execution is correct if each algorithm reports the same count. **AR:** Activity Recognition (AR) is a simple machine learning model that classifies data from a three axis accelerometer as moving or stationary. We emulate an accelerometer with our interrupt providing random data. A transaction ensures that the sample window and count remain consistent. **RSA:** RSA Encryption (RSA) uses a fixed, 64-bit key to encrypt a string updated by an event. A transaction prevents string updates during encryption. **BF:** Blowfish (BF) uses a block cipher to encrypt a string updated by an event, using transactions to prevent updates during encryption. We omit BF for Buffi because the device runs out of memory. **CEM:** Cold-chain Equipment Monitoring (CEM) LZW compresses a stream of temperature data generated by an event. The code synchronizes a double buffered sample buffer and ready flag indicating data are available. **CF:** Cuckoo Filtering (CF) stores and queries for values in cuckoo filter. Events insert data, while tasks insert and query the filter. A transaction prevents concurrent, conflicting updates from being lost.

3.6.2 Correctness

We demonstrate that prior task-based intermittent systems do not correctly support interrupts. For each benchmark, we attempted to add synchronization manually to the Alpaca implementation to support interrupts that concurrently modify task-shared state. We used a combination of careful flag synchronization and short atomic (i.e., interrupts disabled) blocks. Alpaca tasks behaved as normal but we prevented Alpaca from privatizing data in ISR code because allowing it causes Alpaca's atomic commit to fail. The ISR directly updates memory.

Table 3.3 shows that Alpaca fails for all benchmarks except BC. Columns 6–8 show min/mean/max time to failure. The time to failure varies because failure is dependent on specific experimental event timings and harvested-energy recharge time. We

Table 3.3: Correctness. Coati prevents incorrect behavior during intermittent execution. Using Alpaca alone, most applications crash or hang. ✓ indicates correct execution, ✗ is incorrect. Mean time to failure (MTTF) varies with event timing, and application behavior, not measurement error.

	Intermittent Exec. Correct?					**Alpaca MTTF (s)**		
App.	**Atm.**	**H-Op**	**Bff.**	**Coati**	**Alpaca**	**Min**	**Mean**	**Max**
BC	✓	✓	✓	✓	✓	n/a	n/a	n/a
AR	✓	✓	✓	✓	✗	0.02	1.5	3.5
RSA	✓	✓	✓	✓	✗	4.6	26.9	45.4
CEM	✓	✓	✓	✓	✗	0.6	0.7	1.4
CF	✓	✓	✓	✓	✗	1.8	18.1	68.8
BF	✓	✓	–	✓	✗	2.8	3.8	5.0

investigated each failure and verified its root cause was the interaction of intermittent operation and interrupts. The "False Flag" problem was most common: privatization hides updates to synchronization bits set in the ISR or a task. Consequently, AR, RSA, CEM and BF all overwrite updates made by the event. The error causes CEM to stall indefinitely and the lost update causes visible corruption of AR's output. In CF, privatization breaks synchronization, leaving event counting statistics inconsistent. BC does not fail even using Alpaca because the program is simple, the event is very short, and the event timing does not lead to a failure.

3.6.3 Programming Effort

We compared the programming effort required to correctly synchronize the benchmarks with Coati and by hand (Hand-Op). We found that transactions in Coati simplify reasoning about correct synchronization in an intermittent execution. Table 3.4 quantifies the additional code required to manage synchronization for each benchmark with Coati (C) and by hand (H). We counted the number of variables, tasks and task transitions that had to be added or modified. In Coati, the lines of code were primarily used to start and end transactions. No more than 4 new lines of code had to be added to correctly split the event into a top and bottom. Hand-Op requires additional tasks and carefully written transitions to avoid the "False Flag" problem described in Section 3.2. CEM and BF nominally required only a few more lines of code to manage double buffers of data by hand, but the accesses to the new variables had to be carefully placed.

Table 3.4: Programming Effort. Coati (C) reduces the effort to write correctly synchronized code. Synchronizing the code by hand(H) required up to 10x more lines of code to manage extra variables, tasks and transitions.

App:	**BC**		**AR**		**RSA**		**CEM**		**CF**		**BF**	
Config:	**C**	**H**	**C**	**H**	**C**	**H**	**C**	**H**	**C**	**H**	**C**	**H**
lines	15	65	7	70	8	40	21	24	4	32	5	12
variables	0	3	0	3	1	3	2	3	0	1	1	1
transitions	7	14	3	9	2	0	1	1	1	3	0	0
tasks	0	2	0	4	0	0	0	0	0	2	0	0

3.6.4 Events Captured

We evaluated Coati's ability to effectively capture events in an intermittent execution. Coati avoids losing events due to a power failure while synchronization or a disabled interrupt blocks an event.

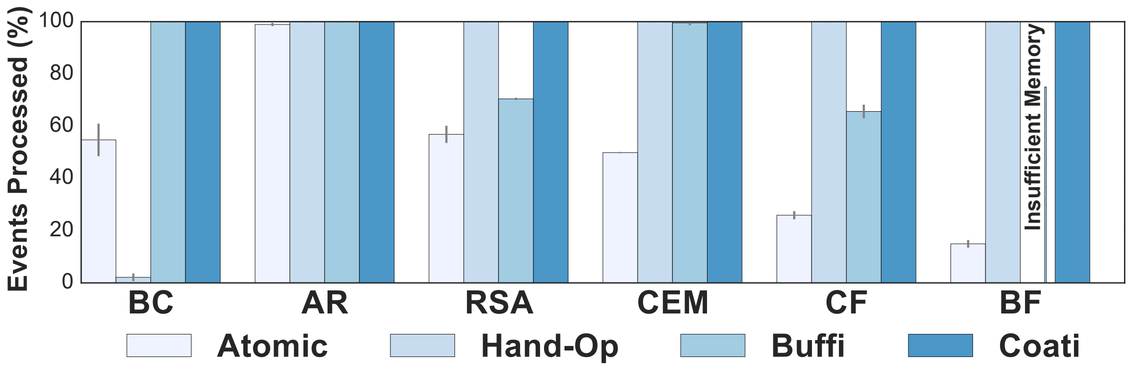


Figure 3.9: Events Captured. Under harvested energy, Coati consistently processed 100% of the events that occurred while the device was powered on.

Figure 5.12 shows the fraction of events that are processed (executed and committed to memory) compared to the total number observed while the device is intermittently powered-on. The key observation is Coati does not disable interrupts and allows the application to process all events that arrive. Buffi discards updates from events that conflict with transactions, so benchmarks with long transactions such as CF and RSA process fewer events. Disabling interrupts (Atomic), causes the application to lose events without running the ISR code. The effect is exacerbated under intermittent execution because pending interrupt signals stored in volatile memory are cleared on reboot. Hand-Op processes a high fraction of events in all benchmarks except BC. Hand-Op uses a try-lock mechanism in BC to prevent the event from writing to the shared index during a critical region, and the event is rarely able to acquire the lock.

3.6.5 Performance

We evaluated Coati's performance on harvested energy and continuous power showing that it has practical overheads compared to an ideal system. On continuous power, Coati is competitive with an "Ideal" Alpaca system that avoids all buffering and commit overheads, but does not run correctly on harvested energy. On harvested energy, our experiments show that all Coati configurations have similar performance.

Figure 3.10 shows the runtime on continuous power for each application and configuration and Figure 3.11 shows the percentage of the runtime spent in different parts of the computation. The data show that progress in AR and CEM are bound

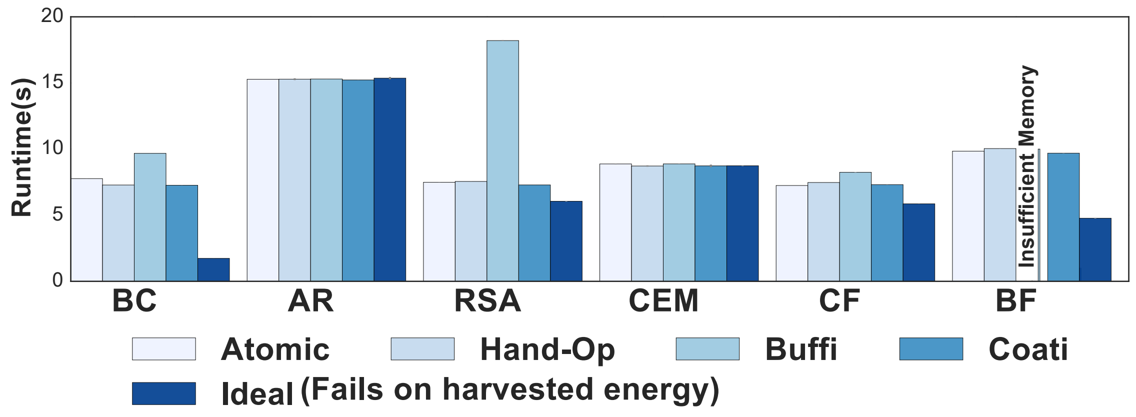


Figure 3.10: Runtime on continuous power. For applications that are bound by event arrival frequency, Coati performs as well as the ideal baseline.

by the frequency of events arriving, not by the performance of computing. As a result, Coati performs as well as the ideal case. BF and RSA perform more memory accesses per event, and Coati's overheads slow it relative to the idealized baseline.

BC computes continuously, regardless of the arrival rate of interrupts and Coati's overhead on accesses to shared memory degrades its performance relative to the idealized baseline. Buffi's complicated commit protocol for tasks in transactions contributes to a 2.5x slowdown of Buffi over Coati for RSA.

Figure 3.12 shows the end to end runtime for each of the applications while the device is powered by harvested energy. The runtimes include the time to recharge and to reinitialize the device on each reboot, which account for approximately 85% of the total runtime (the device is operational for about 85ms before spending about 900ms recharging). The Ideal and Alpaca runtimes are omitted because all of the applications failed to complete correctly. The fastest configuration varies because the overhead incurred by each of the configurations is determined by specific application characteristics. Overall Coati provides performance that is better or within a standard deviation of Hand-Op on all benchmarks without incurring the additional programming overhead of Hand-Op or losing events like Atomic.

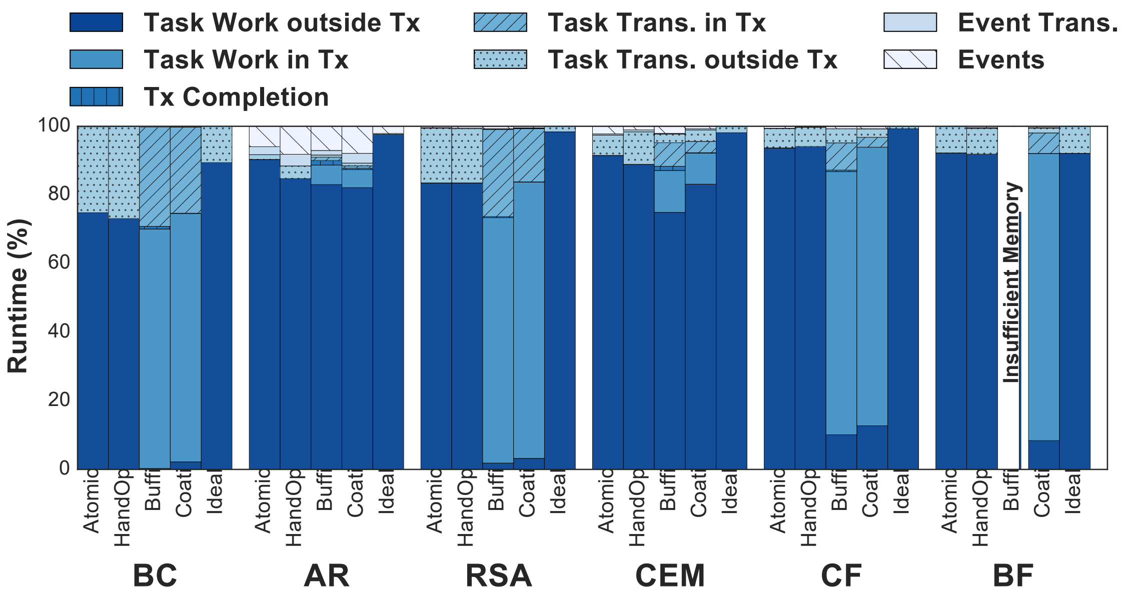


Figure 3.11: Breakdown of runtime usage. The runtime usage varies across the benchmark applications. The intermittent failure-safe systems all incur additional buffering and commit overhead compared to the ideal baseline.

3.7 Related Work

There are several areas of work related to Coati beyond intermittent computing, as discussed in Chapter 2. In particular, Coati touches on work defining synchronization in embedded systems, research on non-volatile memory systems, and transactional memory. Coati applies concepts from prior work to demonstrate that interrupts violate the underlying assumptions of privatization for intermittent tasks.

Concurrency in Embedded Systems

There is a long history of concurrency control research [76, 151, 152]. Prior work [251] provides a survey. The most related efforts on synchronization for embedded systems are TinyOS [158, 159] and nesC [92]. These are frequently-used and provide atomic statements to serialize synchronous tasks and asynchronous events. Other work follows suit [79, 157, 160].

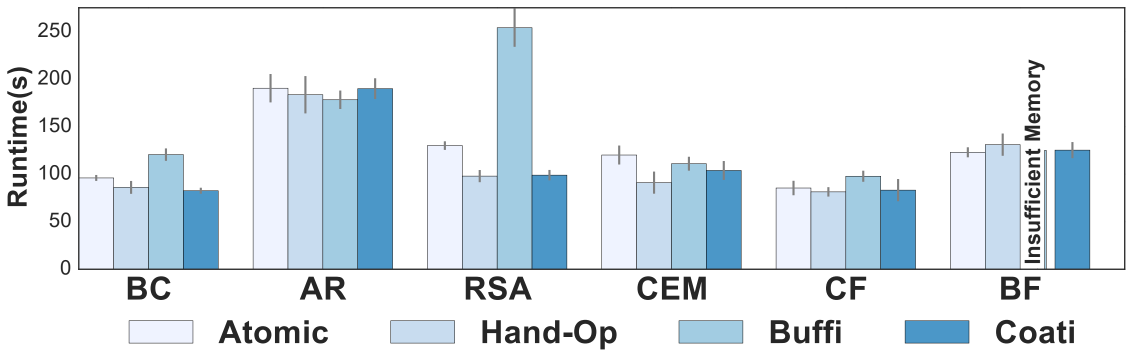


Figure 3.12: Runtime on harvested energy. Coati's performance is comparable to Hand-Op's without the additional programming overhead.

Non-Volatile Memory

The emergence of byte-addressable, non-volatile memory has led to the development of strategies for improving the performance of fault-tolerant, persistent data structures. Persistency models define allowable reorderings of persists to non-volatile memory [100, 145, 206, 207]. Hardware [131, 145, 197, 294] and software [13, 51, 144, 271] support for multi-threaded application programming with persistent memory have been explored. Like these prior works, Coati provides crash consistency for concurrent updates to persistent memory. However, prior work targets large scale systems while Coati ensures data consistency and forward progress under extreme resource constraints.

Transactional Memory

Coati's synchronization model — especially its multi-task transactions — takes a direct cue from a long history of work in transactional memory systems. Transactional memory systems started as mechanisms for manipulating small multi-byte data structures atomically [105, 106, 237], but have grown into a broad research area with support in hardware [107, 187, 227], support for unbounded transactions [15, 37], and support for exotic serialization models [38, 219, 244]. While similar in principle to Coati, the purpose of most prior work on transactions was to synchronize systems that use multiple concurrent threads to perform parallel computations. Between Coati and these prior efforts there are many common ideas: update buffering and commit, conflict detection, atomicity, serialization, and speculation and rollback. Coati draws inspiration from work on transactions, recruiting mechanisms to the specific purpose of synchronizing event-driven interrupts with task-based intermittently executing programs on tiny, energy-harvesting devices.

3.8 Conclusion

This work is the first to study interrupt-driven concurrency in intermittent systems, showing that a naive attempt to combine the two will cause programs to misbehave. Concurrency control is critical to supporting peripherals in batteryless systems because concurrent, peripheral triggered interrupts allow a program to quickly gather data from peripherals without wasting energy polling. This chapter details two types of bugs that occur because the interface to shared data between interrupts and the MCU was not defined in prior work. The "Interrupt Interrupted" bug results in WAR violations and the "False Flag" problem stymies programmers' attempts to synchronize writes made by ISRs and tasks in an intermittent execution. To overcome these problems, this chapter presents Coati, the first programming model that correctly provides concurrency control in intermittent systems.

Coati uses a simple programmer interface and runtime support to provide a TinyOS-like programming model for concurrency with minimal overhead. Coati provides synchronous computational tasks and asynchronous interrupt-driven events, all robust to intermittent operation. In developing Coati, this work also defines transactions in an intermittent system that allow sequences of intermittent tasks to be atomic with respect to events. We explore two different serialization mechanisms for events, tasks, and transactions. Buffi maximizes flexibility by fully buffering updates to shared memory in tasks and events, while Coati uses split-phase interrupts to reduce the memory and runtime overheads associated with concurrency control.

Our evaluation tested interrupt driven workloads on real hardware to compare Coati and Buffi to Alpaca, a state-of-the-art task-based intermittent system, as well as several other approaches for providing concurrency. First and foremost, the evaluation demonstrates that prior work, represented by Alpaca, fails in the presences of interrupts– Alpaca completed only one benchmark successfully. The results go on to show that by restricting access to data shared by events and tasks, Coati achieves concurrency control with low runtime and memory overheads as well as little programmer effort and limited event loss. On average, Coati is faster or within 10% of the runtime of hand-optimized code while requiring far fewer arcane code changes. In summary, this work provides a starting point for future concurrency models in intermittent systems and defines several of the shared-data bugs that system developers must consider.

Correctly managing access to shared data allows peripheral triggered interrupts to coexist with the main thread of execution, the challenge however, is that data is not the only state shared between peripherals and the MCU. In the next chapter, we show that the power consumption of a single peripheral can affect the hardware state of the entire device *and* the correctness of intermittent programming models.

Chapter 4

Debugging and Avoiding Peripheral Energy Bugs

Despite the importance of peripherals, most prior work in state restoration for intermittent execution focused strictly on software state. However, capturing only software state (e.g. registers, stack, global variables) before a power failure is insufficient. To ensure correctness and efficiency – the hardware state (e.g. external sensor configurations) must be handled carefully as well.

As discussed in Chapter 2, hardware *peripherals*, such as sensors, radios and hardware accelerators, are central to applications on batteryless devices, but constant power failures complicate their usage. Two key pieces of system support allow peripherals to be readily integrated into an intermittent execution. First, peripheral restoration systems ease the burden of using peripherals in an intermittent execution by automatically reconfiguring active peripherals after a power failure [33, 41, 220]. Second, support for atomic blocks prevent power failures from disrupting peripheral accesses with unintended pauses [148, 176, 253]. However, writing code that correctly and efficiently manages peripherals in an intermittent execution is still error prone. Peripherals account for a large portion of a batteryless device's energy budget and seemingly slight changes in peripheral operating modes can cause large increases in total system power. For instance, the power consumed by a low end IMU [248] can more than double with a change in the sampling frequency, going from 13% of the total batteryless device power to nearly 30% [57]. The extra power consumption that results from peripheral mismanagement (i.e. incorrectly configuring a peripheral) results in two classes of *intermittent peripheral energy bugs.* The first occurs when a programmer wastes energy by incorrectly disabling a peripheral between uses and causes a slowdown. The second causes the device to live-lock by unintentionally running an atomic block with peripherals configured such that the code consumes more energy than the device can provide. Both present substantial challenges to application developers building pervasive sensing and computing applications.

An energy-harvesting device spends time collecting energy from its environment, so its end-to-end *performance* suffers if it uses energy inefficiently [71]. A challenge for programmers then, is to reduce extraneous power to improve performance, however, we show that attempts to reduce peripheral power lead to *peripheral energy burning bugs*, BurnBugs. A simple strategy for reducing peripheral power is to deactivate a peripheral if it will not be used for a long time, e.g., before a long numerical computation, but activating and deactiviting a peripheral costs time and energy. As a result, this *toggling* strategy requires the programmer to reason about fine-grained peripheral management and instrument code to toggle, which is difficult. Over-instrumenting code leads to unnecessary toggling costs, despite scant benefits; under-instrumenting code leaves a peripheral in a state that forfeits the opportunity to save energy. A program has a BurnBug if its code is either over- or under-instrumented for peripheral toggling. Section 4.6.4 shows that BurnBugs cause a 30% overhead in a full application case study requiring conditional peripheral management.

As discussed in Chaper 2, some operations must run in an atomic block that executes completely without experiencing a power failure [176, 253]. Application developers must ensure that no atomic block consumes more energy than a device can buffer. Such an atomic block will repeatedly fail and partially re-execute, but, exceeding the energy supply, will never complete, effectively "bricking", i.e. rendering useless, the device. We refer to the causes of such repeated failures that are not captured by existing testing procedures as *device-bricking energy bugs*, BrickBugs.

Prior work [55, 111, 176, 224, 253] recognized that an atomic block may consume different amounts of energy depending on the path through the block and requires developers to test each block to confirm that in the worst case each will finish. However, prior work overlooks the energy cost of peripherals when testing atomic blocks and fails to specify how a programmer should account for changes in peripheral operating mode *outside* of the atomic block. If a peripheral's operating mode changes unexpectedly (e.g. inside a device driver or an interrupt handler), the block can consume much more energy than during testing and exhaust the device's energy budget. Such a bug is a permanent failure because the atomic block will indefinitely fail, restore its peripherals to the errant operating modes, use too much energy, and fail again. In this work, we characterize these BrickBugs which stem from the interaction of atomic blocks and peripheral restoration code. Section 4.6.3 shows that BrickBugs cause failures on real devices and are easily introduced when combining atomic blocks and peripheral state restoration.

To enable complex, peripheral-centric applications for batteryless, energy-harvesting devices, we present Pudu[1], a compiler analysis and runtime system to statically prevent and dynamically avoid peripheral energy bugs. We evaluate Pudu's compiler analysis, Pudu-Static, on a set of buggy applications and demonstrate on energy-harvesting hardware that Pudu-Static finds bugs that cause intermittent applications to fail. We also show that Pudu's dynamic runtime, Pudu Auto-Toggle, opportunistically prevents BrickBugs and alleviates BurnBugs to improve application runtime.

4.1 Reducing Peripheral Costs

In an embedded device, programmers are responsible for writing the code on the primary MCU that coordinates an array of peripherals, all with different communication protocols, and power requirements. Further, in an energy constrained device, programmers must consider efficiency, and strategically enable/disable peripherals to reduce excess power. However, the benefit of fine-grained peripheral management varies substantially with changes in peripheral operating mode and application characteristics.

4.1.1 Toggling Cost vs. Benefit

A programmer should disable a peripheral only if the energy saved exceeds the *toggling cost*, the energy cost to place the peripheral in a low-power sleep state and

[1]Pudus are the world's smallest deer. Like batteryless devices, they operate on limited resources because their small stature impedes gathering food. Pudu stands for **P**eripheral **U**pdate **D**eemed **U**nsafe.

restore it before the next use. A single platform may have many peripherals each with a different operating power and toggling energy. Figure 4.1 shows measurements that characterize the operating power and toggling overhead design space for peripheral sensor configurations *on a single device* [57]. The peripherals include proximity and color sensors as well as an accelerometer and a gyroscope running at several sampling frequencies. The plot shows the difference in power between active and sleep mode for each peripheral configuration on the y-axis (i.e., the power benefit of toggling) versus the energy cost of toggling. Toggling energy encompasses both the energy spent to [de]activate a peripheral and the power spent as the MCU waits for the peripheral to finish initialization. The wide variation in operating power and toggling energy presents an efficiency challenge to programmers. Disabling peripherals can substantially reduce power consumption [8, 22, 57, 193], but toggling can impose a high energy overhead.

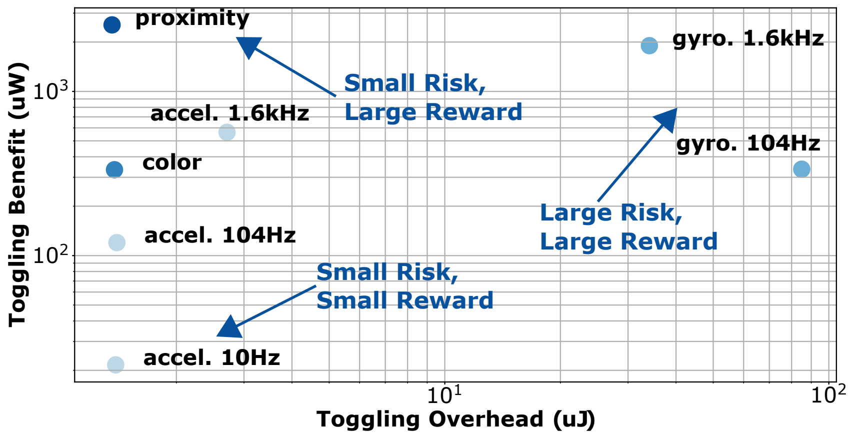


Figure 4.1: Peripheral Design Space. This graph compares the toggling cost and benefit for different peripheral sensors and (where applicable) different sampling rates on a single device.

4.1.2 Application-Level Effects

The key problem for batteryless system developers is the large performance impact peripheral toggling (or not) can have on applications. The benefit (cost) varies across applications, input sizes and peripheral access frequency. System designers need to be able to eliminate inefficiency due to peripheral management to target actual application and system bottlenecks.

Figure 4.2 shows the energy savings from toggling a peripheral across a set of linear algebra applications that operate on different widths of data collected from a gyroscope in two operating modes. Bars above zero indicate energy savings from toggling, bars below indicate an energy cost. More details of the benchmarks will be discussed in Section 4.6.

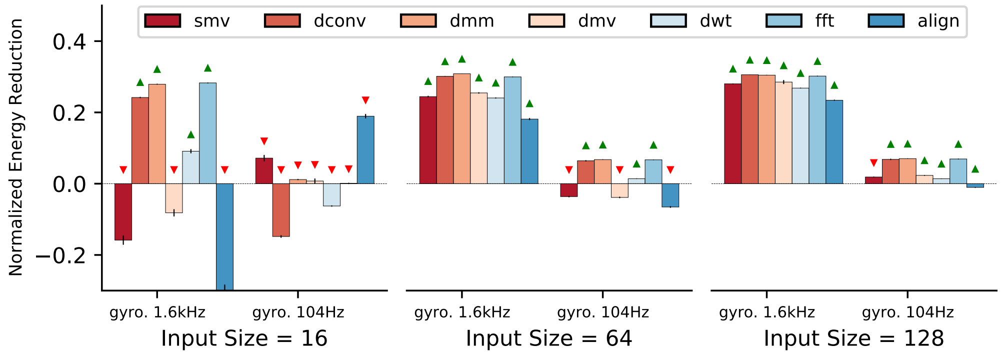


Figure 4.2: Toggled vs No-Toggle. The graphs show the normalized energy reduction of toggled applications.

The data demonstrate the challenges to a programmer of deciding when to toggle peripherals. For applications with large compute kernels that run between peripheral accesses (e.g. dconv, input > 16) toggling is the obvious choice, but in other applications the choice is not. Further, the end-to-end benefit of toggling a peripheral varies with the ratio of application energy spent sensing versus computing; toggling only reduces the energy of computing. Figure 4.3 shows energy savings for two applications, across sensing energy levels. Sensing energy changes by varying the number of data values the application replaces after every iteration (e.g. “min” reads a single value, “full” rereads the entire input block). The energy benefit declines as sensing energy increases because optimizing compute has less effect on the entire application.

In summary, managing peripheral power in energy harvesting devices is challenging because the scale of the problem changes with each application’s exact characteristics, and there is no universally beneficial solution. Batteryless device system developers must be aware of the variable impact peripherals can have on application performance because as the next section demonstrates, subtle changes in code can introduce bugs.

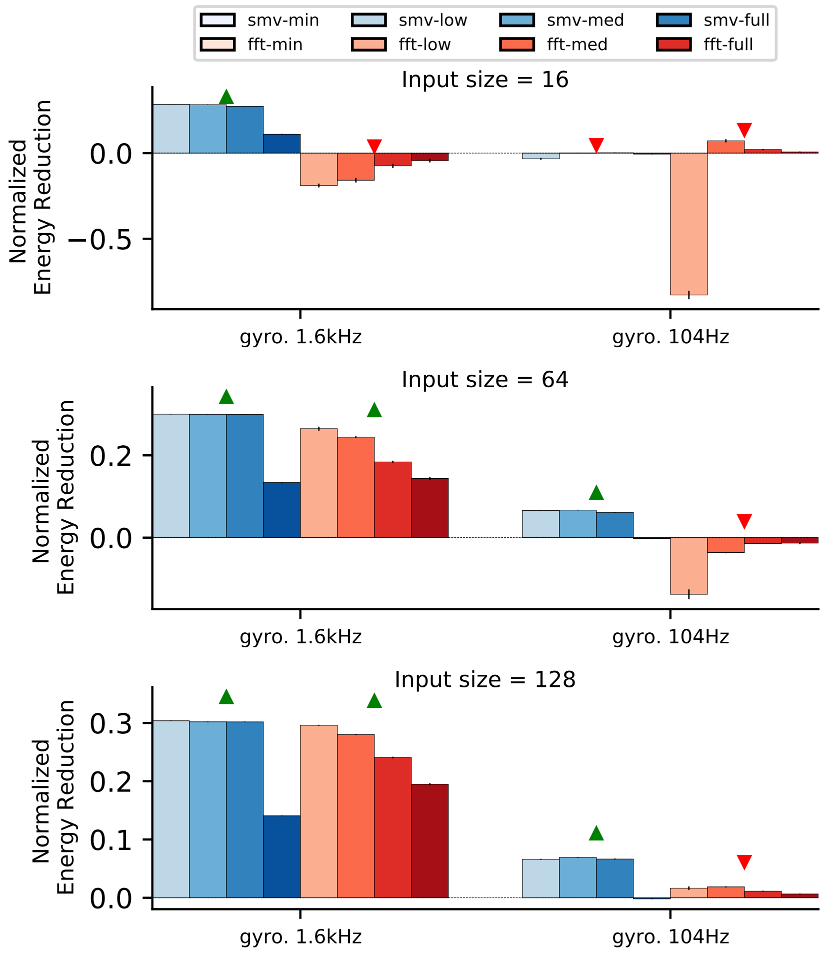


Figure 4.3: Sensing Energy. As sensing energy decreases there is more energy savings available to capture.

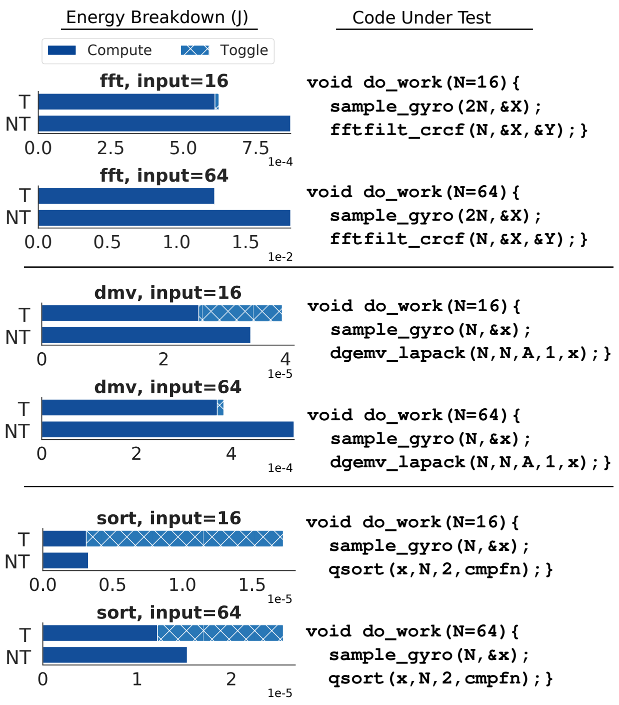
4.2 Problem: Intermittent Peripheral Energy Bugs

There are two categories of energy bugs that stem from peripheral mismanagement in an intermittent system. The first class of bugs are *peripheral energy burning bugs*, which either toggle or fail to toggle a peripheral for some span of an execution when doing the opposite would save energy. The second class of bugs are *device-bricking peripheral energy bugs*, which execute part of a program with a peripheral configuration that consumes too much energy and prevents further progress. In this section, we demonstrate that peripheral energy bugs are simple to introduce and difficult to diagnose in intermittent systems.

4.2.1 Peripheral Energy Burning Bugs

For a programmer trying to improve an application's energy efficiency, toggling peripherals into low-power sleep states when not in use is an ostensibly simple strategy. In reality, the toggling decision is not obvious. The benefit depends on the run time of the code between peripheral uses (which may be input-dependent) and the peripheral's power consumption (which varies depending on its mode as shown in Figure 4.1). If code that runs between peripheral accesses is not long enough to "break even" and amortize the cost of toggling the peripheral, then the program's energy will *increase* due to toggling. Figure 4.4 illustrates the problem with snippets from three different programs, a Fast Fourier Transform (FFT), a Dense Matrix Vector kernel (DMV), and a numeric sort kernel (SORT). Each application collects and processes a full buffer of sensor data in a function, `do_work(int size)`.

The programmer is forced to decide whether or not to disable the peripheral device across each of the compute kernel invocations. Figure 4.4 plots the energy spent to toggle the gyroscope [248] and perform the computation, T, compared to the energy spent in the No-Toggle case (NT) which leaves the gyroscope enabled and incurs no toggle energy. The compute energy, the solid blue part of the bars, is always smaller when the peripheral is toggled off, but the fixed cost to toggle, the striped part, can cause the total energy in Toggle to exceed that of No-Toggle. For both FFT sizes, toggling the gyroscope leads to a significant reduction in compute energy because the toggling cost is negligible relative to the compute cost. The striped toggling bar is so small relative to the FFT compute energy that it is barely visible. DMV with an input size of 64 sees a benefit from toggling, but with an input size of 16, the computation does not run for a long enough time to amortize the toggling overhead. SORT sees no benefit from toggling the gyro regardless of the input size in this experiment. Moreover, unlike FFT and DMV, the run time of SORT is highly input value dependent, not only size dependent, further complicating the programmer's decision. Despite the simplicity in these examples, there is no obviously optimal strategy for managing peripherals to minimize energy consumption, which allows for slowdowns due to BurnBugs if the programmer chooses incorrectly.



```
void do_work(N=16){
  sample_gyro(2N,&X);
  fftfilt_crcf(N,&X,&Y);}
```

```
void do_work(N=64){
  sample_gyro(2N,&X);
  fftfilt_crcf(N,&X,&Y);}
```

```
void do_work(N=16){
  sample_gyro(N,&x);
  dgemv_lapack(N,N,A,1,x);}
```

```
void do_work(N=64){
  sample_gyro(N,&x);
  dgemv_lapack(N,N,A,1,x);}
```

```
void do_work(N=16){
  sample_gyro(N,&x);
  qsort(x,N,2,cmpfn);}
```

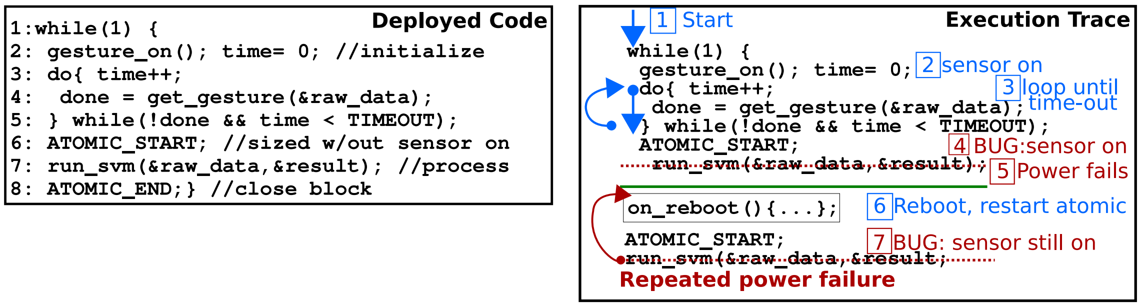
```
void do_work(N=64){
  sample_gyro(N,&x);
  qsort(x,N,2,cmpfn);}
```

Figure 4.4: Peripheral Toggling Cost. "T" indicates a program that toggles off the gyroscope around the computation, "NT" leaves the gyro enabled. The relative cost to toggle a peripheral varies depending on the library code called and arguments passed. Minute changes in code can result in large changes in the energy breakdown.

4.2.2 Device-Bricking Energy Bugs

In batteryless systems with small energy buffers, leaving a peripheral in a high-power state can lead to an unrecoverable system failure. BrickBugs occur when the peripheral operating modes change, unbeknownst to the programmer. Given that embedded systems typically abstract peripheral operation behind device driver APIs, the scenario is not unlikely. Peripheral device drivers are well known sources of bugs in desktop environments [26, 28], causing device modes the programmer did not expect. The problem is exacerbated in the low-end sensing domain where many peripheral drivers come from open-source software repositories (e.g. Arduino, Adafruit, Sparkfun). Bugs and unexpected operating mode changes are common in driver code [19].

Consider the following possible user scenario based on a real complaint about an open-source driver [87]. A programmer wants to build a touch-free remote application for a batteryless device. She selects a (relatively) low power gesture detector with available driver code and links in system support for frequent power failures, including peripheral restoration and atomic blocks. She uses a support-vector machine (SVM) to atomically process the raw gesture samples. Per the state of the art atomic block definition [55, 176] , she tests the SVM atomic block in isolation– by charging the device's energy buffer, and starting execution at the beginning of the atomic block. As the deadline looms, the programmer notices that the driver code gets stuck if someone leaves their hand over the detector instead of completing a swipe [87]. She rewrites the detector code, placing the driver API function that collects gesture data in a loop with a time-out, as shown in Figure 4.5. On continuous power, the application behaves correctly. Per state-of-the-art requirements, she does not re-test the application on harvested energy since she did not modify any atomic blocks, so she ships the code.


Deployed Code

```
while(1) {
  gesture_on(); time= 0; //initialize
  do{ time++;
    done = get_gesture(&raw_data);
  } while(!done && time < TIMEOUT);
  ATOMIC_START; //sized w/out sensor on
  run_svm(&raw_data,&result); //process
  ATOMIC_END;} //close block
```

Execution Trace

```
while(1) {
 gesture_on(); time= 0;
 do{ time++;
  done = get_gesture(&raw_data);
 } while(!done && time < TIMEOUT);
 ATOMIC_START;
  ~~run_svm(&raw_data,&result);~~
on_reboot(){...};
ATOMIC_START;
~~run_svm(&raw_data,&result);~~
```

Figure 4.5: Gesture Detector Bug. `get_gesture` quietly disables the sensor if valid data are captured, but if the loop exits via the time-out, the sensor will be active in the atomic block and exhaust the energy buffer.

The deployed application appears to work until the time-out path is triggered, at which point the application bricks– it is stuck in an endless loop trying to complete the SVM atomic block and never has enough energy. Figure 4.5 shows the reason for the failure. Internally, the driver API call disabled the gesture detector only when a valid gesture was detected. The new time-out path skips the disable and leaves the gesture detector active and drawing power while the SVM atomic block runs. The energy buffer is too small to support the gesture detector during the SVM's atomic block, so the app fails to make progress. Worst of all, the peripheral restoration system turns the gesture detector back on at every reboot because that is the state it was in when the SVM atomic block first ran.

4.2.3 Energy Bug Prevention Techniques

Energy bugs were first characterized in smartphones [129, 130, 201], and the ensuing works proposed fine-grained energy profiling tools [202], to track and automatically correct certain energy bugs [172]. Runtime tools designed for smartphones will not detect peripheral energy bugs in intermittent systems. Such tools continuously power the device, preventing the failures that trigger BrickBugs, and they are not designed to detect the sub-milliamp current changes that indicate BurnBugs. The energy-interference-free debugger [54] is sensitive enough to profile intermittent systems, but its software interface only provides visibility into a program's *s*oftware state, not its hardware state. Compile-time tools for smartphones use techniques similar to Pudu to address bugs stemming from "sleep-disorders" [203] and resource leaks [103] that reduced battery lifetime. These compiler tools use smartphone specific knowledge to identify bugs (e.g. patterns within Android APIs).

Language level techniques for preventing energy bugs in mobile systems have also been explored. Most notably, the Energy-Types (ET) and Ent languages use type checking to prevent energy bugs in which an energy-hungry code fragment is accessed when the device is in a low-energy state [46, 52]. ET/Ent's type checking, however, is designed for battery powered systems where the energy level changes slowly, but a batteryless system's energy-availability, i.e. capacitor voltage, changes rapidly as part of an intermittent execution. Further peripheral energy bugs appear even when using the lowest power peripherals on a device– type checking based on the hardware state will not prevent them.

4.3 Pudu Overview

Pudu is a solution to peripheral energy bugs in batteryless systems. Pudu combines a compiler analysis that helps find and fix BrickBugs with a runtime system that automatically avoids energy-wasting BurnBugs. The flow to use Pudu is shown in Figure 4.6. First, Pudu-Static takes in application code and lightly annotated peripheral driver libraries to track changes in peripheral operating modes, producing bug reports as needed. The programmer uses the reports to iteratively find and fix BrickBugs. The BrickBug-free program is passed to Pudu Auto-Toggle with break-even profiles for each peripheral. Pudu Auto-Toggle links the application and driver code with the Pudu runtime to produce an executable free of BurnBugs.

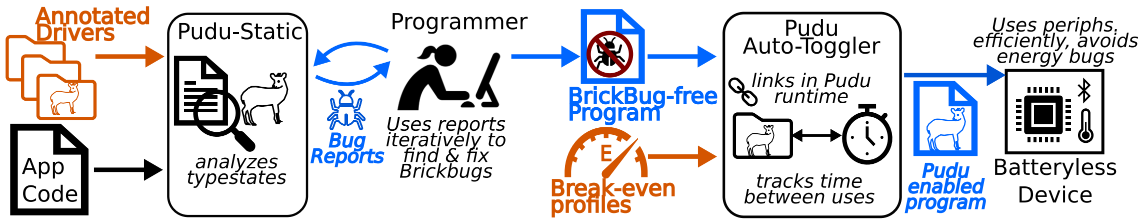


Figure 4.6: Pudu Overview. Pudu uses annotated driver code, peripheral break-even profiles and source code to produce BrickBug reports and instrumented app code without BurnBugs.

4.3.1 Pudu-Static

Pudu-Static is a compiler analysis that searches program code for potential BrickBugs and produces a report for the programmer describing each. For a BrickBug to occur, two conditions must be met by a region of code. First, the code must include an atomic block. Second, the code must allow executing the atomic block in multiple peripheral *typestates*, one of which was unintended by the programmer. Each peripheral's typestate is equivalent to its operating mode. At the device level, we refer to the set of all peripherals' operating modes as the device's *peripheral typestate*. BrickBugs occur only in atomic blocks, because JIT checkpointing preserves state and typestate without re-execution. Multiple peripheral typestates are required to trigger a BrickBug because if only one peripheral typestate is possible, we may assume the block was tested with it.

Consider the BrickBug experienced by the touch-free remote in Section 4.2.2. The bug occurred because the SVM atomic block could not finish with the gesture sensor enabled, but the block was only tested with the gesture sensor off. Pudu uses a heuristic to find potential BrickBugs for an atomic region: if the region may execute in more than one peripheral typestate, the analysis flags that region as a possible BrickBug. Pudu-Static then reports the atomic block and the possible peripheral states at its entry point. For each bug report, Pudu-Static reports where in the code each such peripheral state change may happen. In the touch-free remote, Pudu-Static would flag the atomic block on line 6 of Figure 4.5 because it may run with the gesture sensor either enabled or disabled. Pudu-Static would then report the lines in the driver code (not pictured) where the enable/disable mode changes occurred.

The programmer can iteratively use the information in the report to change the application logic and avoid the offending typestate change along paths leading to the reported atomic region.

4.3.2 Pudu Auto-Toggle

Once the program is free of BrickBugs, Pudu Auto-Toggle is linked in to remove BurnBugs that occur if peripherals are toggled too frequently or infrequently. We use a dynamic analysis to remove BurnBugs, because input dependent control flow can radically change the time between peripheral accesses, which makes static analysis inaccurate. Instead, Pudu Auto-Toggle determines at runtime if toggling is beneficial without programmer intervention. Pudu Auto-Toggle uses pervasively available embedded timing hardware in the MCU to track the time between a peripheral's uses, comparing that time to the break-even time of that peripheral. If the time between peripheral actions tends to exceed the break-even time, then toggling the peripheral is likely to be beneficial, and Pudu Auto-Toggle opts to toggle off, otherwise, the peripheral is left active. Pudu Auto-Toggle must run *after* Pudu-Static in the Pudu flow because it introduces new peripheral typestates that would be flagged as BrickBugs. The typestate changes produced by Pudu Auto-Toggle, though, yield to lower power in any atomic blocks long enough to cause BrickBugs. Further, Pudu Auto-Toggle's power-failure awareness corrects the unlikely scenario where it a BrickBug is temporarily introduced by toggling.

4.4 Pudu Design & Implementation

Pudu's design and implementation are decomposed into its programmer-visible features, its compiler analysis, and its runtime library. We also present a profiling approach for finding peripherals' break-even times.

4.4.1 Programmer-visible Pudu Features

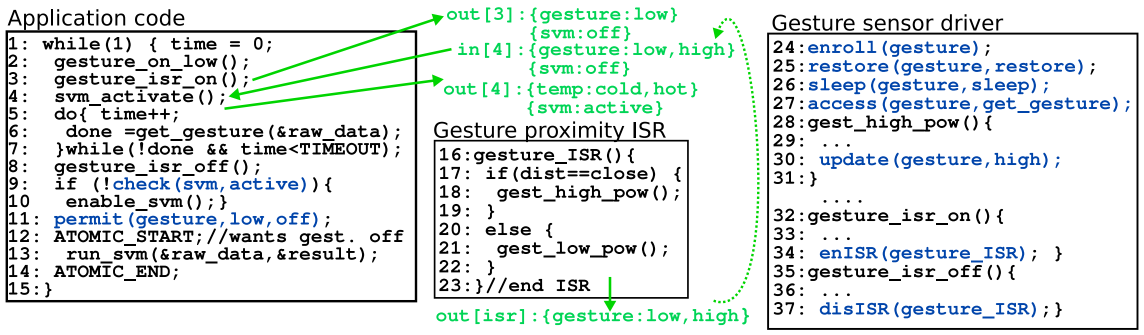
Pudu requires very little extra programming effort, except for a small burden of annotation in device drivers. The programmer visible features are separated into those commonly used by the device driver programmer and those used by the application programmer. The separation is necessary because some of Pudu's annotations require an understanding of the peripheral operation. We expect the driver writer to have this understanding, but not necessarily the application programmer. The application programmer simply accesses all peripherals via the API exposed by peripheral device drivers. We implemented the annotations as a collection of C preprocessor macros and internal functions detected by our compiler. The full Pudu API is defined in Table 4.1.

Table 4.1: A summary of the Pudu API. `per` is a peripheral. `state` is a peripheral typestate value. `func` is a function. `ISR` is an interrupt triggered by an asynchronous event.

Typestate Tracking	**Function Registration**	**Checkers**
`enroll(per)`	`restore(per,func)`	`check(per,state)`
`update(per,state)`	`sleep(per,func)`	`permit(per,state,...)`
`enISR(per,ISR)`	`access(per,func)`	
`disISR(per,ISR)`		

Figure 4.7 demonstrates the API usage by annotating the gesture sensing application described in Figure 4.5 and excerpts from its associated driver. Beyond the behavior described in Section 4.2.2, the application includes an ISR to respond to the gesture sensor's proximity interrupt. The interrupt fires in response to a sensed object's movement– switching to high power to capture the movement if the object is close, and switching to a low power wait state if it has moved away. The driver writer first registers peripherals and functions that will be tracked by Pudu (code lines 24-27 in Figure 4.7). The `enroll` primitive creates and initializes typestate tracking for the provided peripheral. The `sleep` and `restore` annotations designate driver API functions that put the peripheral into low-power sleep mode, and restore it to an active mode. The `access` annotation denotes a function that reads from a peripheral, but does not change its typestate.

The driver must define the set of a peripheral's modes, and end code that transitions from one mode to another with a Pudu `update(per,state)` operation, which updates the typestate, for peripheral, `per`, to a new typestate, `state`. In code line 30, the driver writer indicates a change in the gesture sensor's power level to Pudu. Finally, if driver code triggers asynchronous events, e.g. enables an interrupt and associated interrupt service routine (ISR), the code must include `enISR` and `disISR` annotations. The "ISR" annotations associate a peripheral with an asynchronous event, ISR, and inform the typestate analysis where in the code the event may fire. In Figure 4.7, the driver writer indicates in code lines 34 and 37 when the interrupt is enabled/disabled. This level of driver annotations mimics the state tracking used in peripheral restoration systems [41].



```
1: while(1) { time = 0;
2:   gesture_on_low();
3:   gesture_isr_on();
4:   svm_activate();
5:   do{ time++;
6:    done =get_gesture(&raw_data);
7:   }while(!done && time<TIMEOUT);
8:   gesture_isr_off();
9:   if (!check(svm,active)){
10    enable_svm();}
11: permit(gesture,low,off);
12: ATOMIC_START;//wants gest. off
13:  run_svm(&raw_data,&result);
14: ATOMIC_END;
15:}
```

```
16:gesture_ISR(){
17: if(dist==close) {
18:  gest_high_pow();
19: }
20: else {
21:  gest_low_pow();
22: }
23:}//end ISR
```

```
24:enroll(gesture);
25:restore(gesture,restore);
26:sleep(gesture,sleep);
27:access(gesture,get_gesture);
28:gest_high_pow(){
29:  ...
30:  update(gesture,high);
31:}
    ....
32:gesture_isr_on(){
33:  ...
34:  enISR(gesture_ISR); }
35:gesture_isr_off(){
36:  ...
37:  disISR(gesture_ISR);}
```

Figure 4.7: Example code annotated for Pudu. An improved gesture detection application code example uses Pudu's checking annotations and relies on a Pudu annotated driver (shown in blue). The effects of the gesture ISR on the output of the Pudu pass are shown in green.

To use Pudu, the application programmer need only provide application code using a properly annotated driver. However, during development and debugging, the programmer can use Pudu as a dynamic or static typestate specification checker. The dynamic `check` annotation returns "True" at runtime if the typestate of peripheral `per` matches `state`. The static `permit` annotation specifies a set of allowed typestates at a program point. Pudu produces a compile-time warning if unspecified typestates are possible at a `permit`. Further, the allowed typestates will not be flagged as bugs during BrickBug detection. For example, the application programmer in Figure 4.7 added a `check` annotation to configure the SVM hardware and a `permit` annotation to limit the gesture sensor to either the "off" or "low" states.

4.4.2 Pudu-Static: Peripheral Typestate Analysis

The Pudu compiler is a custom LLVM pass [153] that analyzes the annotated program and driver source code. The analysis translates `update` annotations into assignments to a runtime variable that represent the peripheral's typestate. The compiler performs a context-sensitive reaching-definition analysis of the typestate variables to compute the set of possible peripheral typestates at each code point [97]. Pudu's analysis reports, for each point in the code, what typestate each peripheral may be in when the code executes. For instance, the complete typestate at code line 3 of Figure 4.7 is {`gesture:low,svm:off`}. Pudu assumes an initialization function restores a peripheral's operating mode on restart at a checkpoint [31, 126, 176] or atomic block. Algorithm 1 shows a high level description of the analysis pass as it is used for BrickBug detection.

Algorithm 1 Pudu-Static Analysis Pass.

```
1: function ReportBugs(Program $P$)
2:     AllStates $T \leftarrow Per[\varnothing]$            ▷ Initialize the set of registered peripherals
3:     MarkDrivers($P$)            ▷ Denote functions that modify peripheral state as drivers
4:     SplitBlocks($P$)            ▷ Splits basic blocks at peripheral state changes
5:     Analyze($P.ISR$,$T$,$\varnothing$,$\varnothing$)            ▷ Analyze state changes in ISRs
6:     Analyze($P.main$, $T$,[$Off$],$\varnothing$) ▷ Analyze state changes in the program; all peripherals start in "Off" state
7:     $specs$ ←CheckSpecs($P$,$T$)            ▷ Compare programmer spec. to reaching states.
8:     $reports$ ←CheckAtomics($P$,$T$)            ▷ Report if > 1 typestate at atomic block
9:     $filteredReports$ ←FilterReports(reports)            ▷ Filters non-critical bugs
10: end function
```

The analysis begins by identifying "Driver" functions that contain assignments to peripheral typestates or call functions that modify peripheral typestates, e.g. `gest_high_pow`. Next, all basic blocks are split at function calls and peripheral typestate assignments to form separate basic blocks. This change allows the analysis to determine the effect of asynchronous interrupts regardless of where typestate assignments fall in a block, e.g. relative to calls that disable asynchronous peripheral operations. For example, breaking code lines 2-4 into separate blocks clarifies that the ISR cannot affect the peripheral typestate until code line 4. On line 5, the analysis updates T, the data structure storing the peripheral typestates for each basic block in the program, to include information about the reaching typestates on exit from each ISR. For instance, after analyzing `gesture_ISR`, T would report that the ISR changes the gesture sensor state to either "low" or "high". Pudu uses the ISR reaching typestate information when Analyze runs on `main`, at line 6. The input typestate of all peripherals is set to "Off" when Analyze is run on the `main` function to simulate the initial condition of all peripherals on first boot.

The typestate information for the code reachable from `main` is passed to CheckSpec to confirm the programmer's `permit` annotations and to CheckAtomics to produce bug reports. A bug is reported if any peripheral has multiple reaching typestates, unspecified by `permit`, at the beginning of an atomic block. However, some atomic blocks are unlikely to cause BrickBugs because their energy requirements are small and the runtime is short. We found that short atomic blocks fall into two categories:

those that form part of the hardware access layer (HAL) for peripheral driver calls, and those used to protect simple input-output (I/O), like calls to `printf`. FilterReports implements the HAL-I/O filter to optionally drop these excess bug reports. The filter only hides blocks that contain a *single* function call, e.g. a single, low level driver call, to reduce to likelihood of false negatives.

Algorithm 2 Analyze function definition. Analyze uses an iterative worklist algorithm to perform a reaching definition analysis on peripheral typestates.

```
1: function ANALYZE(Function $F$, AllStates $T$, States $In$, States $Out$)
2:     $list \leftarrow$ RETURNBASICBLOCKS($F$)                ▷ Place all blocks in worklist to start
3:     while $list \neq \varnothing$ do
4:         $BB \leftarrow list.pop()$
5:         MEETUNION($BB, In, T$)                ▷ Take the union of the output of predecessor blocks
6:         MEETUNIONISRS($BB, In, T$)                ▷ Take the union of the output of active ISRs
7:         for $I \in BB$ do                ▷ For each instruction in block
8:             if ISDRIVERCALL($I$) then
9:                 $f \leftarrow$ GETCALLEDFUNCTION($I$)
10:                ANALYZE($f, T, In, Out$)                ▷ Necessary for context sensitive analysis
11:            else
12:                TRANSFERFUNC($I, In, Out$)                ▷ $I$ kills prior definitions and generates a new one
13:                $In \leftarrow Out$                ▷ Copy new state for next instruction
14:            end if
15:        end for
16:        if $T[BB] \neq Out$ then                ▷ If BB's state changed, add successors to worklist to continue iterating
17:            $T[BB] \leftarrow Out$
18:            $list.push$(GETSUCCESSORS($BB$))
19:        end if
20:    end while
21: end function
```

Algorithm 2 shows the details of the iterative dataflow analysis that we use to find the typestates of each peripheral that are possible at each code point [10]. The green text in Figure 4.7 shows part of the progression of the analysis on the code example. The algorithm works like a reaching definition analysis, except that it analyzes only peripheral state definitions and uses, not general memory accesses. At a high level, the algorithm takes the input typestate (`in[]`) at each basic block (`BB`), and drops old definitions of peripheral state if new ones occur within the block to produce the block's output typestate `out[]`. Table 4.2 defines the exact dataflow framework we use. The dataflow domain is peripheral typestate changes made in a basic block ($P_{cfg}(B)$) that map the state of a peripheral ((p, s)) to a new state ((p, x)). If a block (`BB`) changes a peripheral's state, it *generates* a new state definition and *kills* all preceding peripheral state definitions.

Table 4.2: Pudu dataflow framework. Pudu's framework closely resembles that of the reaching definitions problem. Minor changes include the starting (boundary) conditions and the addition of active ISRs ($isrs(B)$) to a block's predecessors.

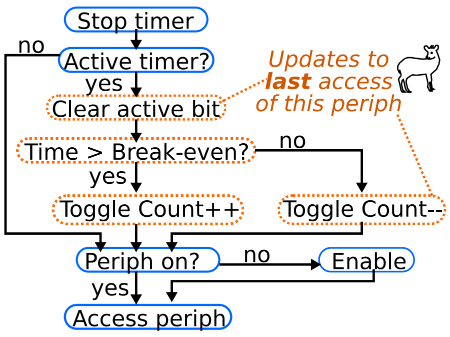
Domain	$P_{cfg}(B) : (p, s) \leftarrow (p, x)$
Direction	Forward
Generate	$gen[B] = \{P_{cfg}\}$
Kill	$kill[B] = (p, *) - P_{cfg}(B)$
Transfer Function($f_B(x)$)	$gen_B \cup (x - kill_B)$
Meet Operation ($\wedge$)	$\cup$
Equations	$OUT[B] = f_B(IN[B])$ $IN[B] = \wedge OUT[pred(B) \cup isrs(B)]$

The first major difference between our algorithm and a standard reaching definition algorithm is the inclusion of a second meet operation (line 6) specifically to fold in typestate changes made in ISRs. MeetUnion takes the union of possible typestates produced by `BB`'s predecessors to update the input to `BB`. The MeetUnionISR takes the union of reaching typestate for any ISRs that may be active at `BB`. MeetUnionISR checks if an interrupt may fire by checking for calls to `enISR` along any path between the current basic block and the start of the function. Then MeetUnionISR confirms that there is not a call to `disISR` that dominates the current basic block and nullifies the `enISR`. The effect of MeetUnionISR is visible in the input and output typestates at code line 4. The output of code line 4's predecessor (code line 3) is merged with the output of the ISR (`gesture:low,high`) to form `in[4]`, because the ISR is enabled at code line 3.

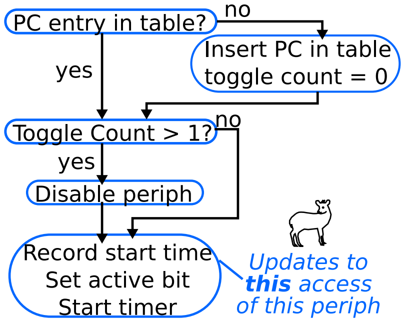
The second difference is the recursive call to Analyze on line 8 if the pass encounters a Driver call, allowing the pass to find `update` calls embedded in drivers. The call to `gest_high_pow` on code line 18 would activate the recursive call. Since our reaching definition analysis is conservative and the code bases for the programs we analyze are small, the pass can perform context-sensitive analysis quickly. The pass reliably completes in under 1 second for all our test cases.

4.4.3 Pudu Auto-Toggle: Peripheral Toggling

Pudu Auto-Toggle is a runtime system that tracks the time between peripheral uses to automatically toggle peripherals (or not) to minimize peripheral energy costs. The major features of the Pudu Auto-Toggle software are the runtime instrumentation, the toggling decision policy, and the Tracking Table, a hash table for storing the relevant metadata.



(a) **Before Access.**



(b) **After Access.**

Figure 4.8: Pudu Auto-Toggle Instrumentation. Before an `access`, Pudu updates the toggling decision for the *last* access to the peripheral in use. After the access, Pudu acts on the toggling decision for the current access.

Pudu Auto-Toggle inserts instrumentation before each instance of an `access` function, as shown in Figure 4.8a. The instrumentation first stops the timer and checks the Tracking Table for the last time the peripheral was accessed, i.e. an entry with a matching peripheral and an "on" active bit. If a matching, active access is found, Pudu updates the active access' toggle predictor and clears its active bit in the Tracking Table. Pudu then toggles the active peripheral back on, if necessary, before proceeding with the `access`. After the `access`, Pudu Auto-Toggle's instrumentation searches for an entry in the Tracking Table that matches the current program counter value (PC), see Figure 4.8b. If no entry is found, Pudu inserts a new entry using the current PC as an identifier and resolves collisions with linear probing. Once an entry is found or created, Pudu makes a toggling decision, sets the active bit and restarts the timer.

Pudu Auto-Toggle also uses instrumentation on reboot and power down to provide the appearance of a continuously ticking clock and gather power failure data. Pudu records the timer status with each JIT checkpoint, then on reboot, restores the timer and increments each active peripheral access' failure counter (not pictured in Figure 4.8. Unlike efforts that keep track of wall-clock time through power failures [63], Pudu only tracks time while a device is on.

Toggling Decisions

Pudu's toggling decision mechanism is characterized by three key features: predictor function, decision granularity, and the effect of power failures on the decision. At each `access`, Pudu updates a 2-bit saturating counter to predict whether or not toggling is beneficial. Pudu compares the time since the last use of the peripheral, in any mode, to the break-even point for the peripheral in its current operating mode. If the time is greater, Pudu increments the counter; otherwise Pudu decrements the counter. In our prototype, a counter value greater than 1 indicates a peripheral should toggle off after a peripheral operation. The small saturating counter design prevents a single transient event from changing Pudu's decision and still allows the decision to change throughout the program.

Pudu Auto-Toggle makes toggling decisions for each point in the code where a peripheral is accessed, as determined by the program counter. Our Pudu Auto-Toggle prototype does not track the full calling context of peripheral accesses. This limitation was not a problem because in most of the programs we analyzed, the peripheral accesses are called from `main`. For each PC entry, Pudu tracks toggling decisions *separately* for up to eight operating modes because modes can have drastically different cost/benefit trade-offs as shown in Section 3.1.

Pudu Auto-Toggle updates toggling decisions on every reboot to consider power failures. If three or more power failures occur since a peripheral's last use, Pudu updates the toggling decision of the last peripheral `access` to toggle the peripheral off. The effect of forcing the toggling decision to off, is any problematic, or repeatedly unused, peripherals will be left inactive on reboot, opportunistically preventing BrickBugs. While a peripheral is associated with an active asynchronous event (i.e., a sensor set to trigger an interrupt), Pudu sets the time and power failures between peripheral uses to zero, and never toggles the peripheral. Pudu Auto-Toggle's policy uses power failures and break-even time comparisons to simultaneously address BrickBugs and BurnBugs.

Hardware Resource Usage

Pudu Auto-Toggle uses a timer on the MCU to count "ticks" between peripheral accesses. Our prototype uses a slow clock (32 kHz) for negligible power [264]. Embedded systems are often extremely resource limited, so we designed Pudu Auto-

Toggle to share a clock with other code in the system. For a small memory overhead, the system maintains the Tracking Table with one eight-byte entry per peripheral operation, and 16 entries. The Pudu Auto-Toggle instrumentation library consumes 2 KB of code memory, and the total overhead varies with the number of peripheral accesses. Compiled for the MSP430, the overhead per peripheral was 160 bytes because we inline the runtime calls. With low power, timer, and memory overheads, Pudu is simple to incorporate into a resource-constrained system.

4.4.4 Break-Even Time Profiling

Pudu Auto-Toggle requires a table of break-even times that includes an entry for each peripheral in all of its operating modes. Datasheets for a given peripheral rarely provide information about the energy cost to transition from one mode to another, and typically report current for a small selection of a device's various modes. As a result, determining break-even times requires directly measuring the target device. To determine the break-even time for a peripheral in some arbitrary mode, our profiler takes a brute-force approach. The profiler runs the MCU in a busy loop to capture system-wide power and then measures the iteration energy with the peripheral active for a range of compute times. Next, the profiler measures the toggled version, measuring the energy to run, first disabling the peripheral, then computing, then re-enabling the peripheral. We accumulate a set of measurements for different time values and empirically find the smallest time when the toggled loop consumes less energy than the loop with the peripheral active, the break-even point.

4.5 Methodology

The goal of our evaluation is to demonstrate the prevalence and significance of the code patterns that lead to intermittent peripheral energy bugs, and then evaluate Pudu's ability to prevent them. Our methodology relies on testing on real energy harvesting devices to demonstrate that Pudu provides correctness and performance where state-of-the-art runtime systems fail.

4.5.1 Hardware Setup

All experiments were performed on the Capybara [57] energy-harvesting platform with its MSP430FR5994 MCU operating at 8MHz. We chose to use the Capybara platform because it is already equipped with several peripherals (e.g. IMU, gesture detector, BLE radio) as shown in its open-source schematic [5]. Capybara's power system is similar to other batteryless devices that use an output booster to maximize the energy extracted from large capacitors [167, 193, 281], but only if the reconfigurable energy-buffer is disabled. We shorted Capybara's separate energy buffers together to represent more common batteryless devices that use a unified energy buffer [63, 166, 193, 210, 229, 284]. To charge the energy buffer, we emulate a stable solar energy source by feeding the 3.3 V output of the Texas Instruments ez-FET [266] programmer into a 2.2 V voltage regulator in series with a 1 kΩ resistor. We measured the voltage drop across the 1 kΩ resistor and confirmed that no more than 60 μA of current are ever supplied, which forces the Capybara into frequent power failures. To tolerate power failures, we implemented Samoyed for JIT checkpointing and atomic blocks [176], and KARMA to restore peripheral state [41].

Sensors & Break-Even Points.

We profiled six peripheral operating modes on the Capybara platform to find their break-even points, following the procedure outlined in Section 4.4.4. We measured current using a differential amplifier configured for high-side current sensing [257] sampling at 625kHz [228]. A digital logic analyzer captured the start/end GPIO pulses from the MCU to measure time. We characterized the on-board peripherals in different operating modes. The gyroscope and accelerometer, which are packaged together in a single chip [248], were tested separately at sampling rates of 1.6KHz and 104Hz. The proximity and color sensors, which are in the same package as the gesture sensor [22], were both tested in active mode. The gyroscope, accelerometer and color sensor are similar in power consumption to other lower power sensors used in batteryless systems [14, 40, 114, 239]. The proximity sensor is representative of more expensive peripherals that like radios or cameras [63, 167, 193, 284] and serves as a stand-in for the gesture sensor referenced in Section 4.2.2. Table 4.3 shows the resulting profile used in our evaluation. Peripheral break-even times vary inversely with peripheral power from 0.77ms (proximity) to 85.44ms (gyro, 104Hz).

The maximum benefit of toggling a peripheral ranges from 3% to 40% of total system power, varying directly with peripheral power. We include the current savings from toggling and the minimum time to toggle from sleep to active mode according to the peripheral datasheet to allow for comparisons to the peripherals included in our energy bug survey.

Table 4.3: Peripheral profiling results. The break-even times on a Capybara sensor span two orders of magnitude. The Toggle-Time is the minimum time to shift from sleep to active according to the datasheet.

Op. Mode:	**Accel,1.6KHz**	**Accel,104Hz**	**Gyro,1.6KHz**	**Gyro,104Hz**	**Color**	**Prox.**
Break-Even(ms)	6.7	7.0	9.5	85.4	9.4	0.8
Power Savings	13%	3%	29%	13%	7%	43%
Toggle-Time(ms)	1.2	0.3	2.4	19	2.4	0.3
Current savings (μA)	235	50	790	140	140	1060

4.5.2 Peripheral-Energy Bug Analysis

We begin our evaluation of peripheral energy bugs with a literature survey that demonstrates the prevalence of patterns that lead to BrickBugs and BurnBugs in existing batteryless systems. We go on to demonstrate BrickBugs, using three applications containing BrickBugs that interact with low-power peripherals and perform complex computations similar to prior work in batteryless systems [41, 55, 57, 98, 176, 193].

Bug Survey

We conducted a survey of open-source code bases that use peripherals from batteryless, energy-harvesting systems literature. To characterize the applications, we recorded lines of application code, (omitting peripheral driver code), and the number of peripherals. In each code base, we examined the peripheral usage in the context of the target hardware platform to determine if a BurnBug was likely due to under or over toggling. Using each peripheral's datasheet, we gathered the minimum time to toggle the peripheral and the corresponding power savings based on the way the application configured the peripheral. For peripherals with substantial toggling power savings and toggling times less than 20 ms, we report likely under-toggling BurnBugs if an application carries out a complicated operation without disabling the peripheral. Detailed examples are provided in Section 4.6.1. Under-toggling BurnBugs are identified in applications with toggling costs that are high relative to their benefit.

To find potential BrickBugs, we identified instances where regions of code that must be atomic could run in multiple typestates. We limit our reported BrickBug patterns to those where the affected atomic blocks perform a non-trivial operation, e.g. computing a large matrix-matrix multiplication, not updating a single running average. In many of the code bases we examine, the underlying runtime will prevent permanent bricking behavior. However, the existence of patterns that lead to permanent failures serves as a warning to batteryless system developers and motivates Pudu as batteryless systems target more complex sensing and computing applications.

Bug Testing

The core bug kernels we test are derived from applications in the batteryless-computing literature [57, 176]. We extract code segments that control peripherals and make them resilient to power failures by following the procedures from prior work to add atomic blocks and peripheral state restoration. Each buggy program is run on a Capybara using the smallest capacitor size that supports the largest atomic block in each program, and each leads to failures on harvested energy using state-of-the-art runtime support. To correct the bugs, we run Pudu-Static on each application and iteratively remove bugs; we use the corrected applications to explore false positives produced by our analysis.

To demonstrate Pudu-Static's usefulness across peripheral restoration policies, we implemented each bug using the RESTOP [220], Sytare [33], and KARMA [41] peripheral restoration policies. We also test the Pudu Auto-Toggle variant, which tracks and restores peripheral state similarly to KARMA, but uses Pudu's auto-toggling strategy (Section 4.4.3) instead of KARMA's lazy initialization. We implemented *asynchronous* versions, when applicable, with interrupt-triggered peripheral operations to test Pudu's ability to track changes through ISRs. The asynchronous variants assume that any function that disables a peripheral disables that peripheral's interrupt.

4.5.3 Performance Benchmarks

We evaluate Pudu Auto-Toggle using compute kernels developed for the MSP430 by prior work, obtained from the authors [99]. The kernels are representative of on-device signal processing and machine learning applications found in embedded sensing devices [98, 121, 168]. **smv:** multiplies a dense vector of samples with a sparse matrix. **dconv:** convolves a matrix with a filter. **dmm:** multiplies two matrices. **dmv:** multiplies a vector and a matrix. **dwt:** transforms a matrix of samples using a wavelet. **fft:** performs 2-D fft. **align:** Needleman-Wunsch sequence aligns a vector of samples and a delayed version of the vector. The individual kernels vary in time complexity and each accepts multiple input sizes. On continuous power their execution times range from under 10ms to greater than 40 seconds, with many falling below the break-even point of peripherals, benefiting from Pudu.

Each benchmark repeatedly runs a kernel with a sequence of input dimensions and re-samples a subset of the input data from a peripheral on each iteration. The sequence is 20 iterations on small input (16 elements), 5 iterations on medium (64), and 1 iteration on large (128). Changing the kernel input size changes the time between peripheral accesses, so to avoid artificially weighting our study towards short or long kernel executions, the sequence provides roughly equal execution time in each case. We ran the benchmark applications using Capybara's largest single bank, 38 mF of capacitance, which corresponds to a powered-on period of about 2 seconds and a recharge time of 30 seconds. The large capacitor mimics recent work that pursues compute intensive applications and support (relatively) high energy peripherals. For instance, a batteryless robot used 22 mF of capacitance [284] and recent work supports object detection reported via LoRa with a 33 mF bank [72].

Performance Comparison

Energy savings translate directly into performance improvements in energy-harvesting applications because these devices take time to harvest all of the energy they use [71]. Our evaluation, therefore, focuses on the performance impact of excess peripheral energy and reports runtime instead of energy savings (e.g. in joules) to demonstrate the direct benefit of peripheral energy savings. To capture the many factors that impact application performance, all runtime data in our experiments are end-to-end measurements. The measurements include time spent checkpointing, recharging, and re-initializing the device. We compare Pudu Auto-Toggle's runtime to two static peripheral management approaches: leaving the peripherals active throughout the application (No-Toggle), and toggling peripherals after every use (Toggle). No-Toggle is the default for all but expert programmers because as described in Section 3.1, correctly manipulating a peripheral requires intimate knowledge of the device hardware. Toggle mimics aggressive driver code– some peripheral drivers wrap peripheral setup and disable code within a peripheral access call. As we show in Section 4.6.1, Toggling after each peripheral access is not common practice for low power peripherals. Further, this technique does not consider whether the program will amortize the toggling cost.

4.6 Evaluation

We evaluate this work in several phases. The first is a survey of potential peripheral energy bugs that appear in code from the batteryless computing literature, showing the prevalence of code patterns that lead to energy bugs. The second is bug analysis and testing which shows, via experiments on real hardware, that prior systems encounter BrickBugs that Pudu can identify and help the programmer fix. The third is performance testing to demonstrate the practicality of Pudu Auto-Toggle on real energy-harvesting hardware running a range of workloads.

4.6.1 Prevalence of Intermittent Peripheral Energy Bugs

We looked at code repositories from 14 different publications [27, 55, 57, 61, 63, 64, 146, 148, 176, 177, 193, 252, 253, 284] , and found code patterns that are indicative of peripheral energy bugs in 7 of them. The results of the bug survey are shown in Table 4.4. We sort the bugs based on the software runtime, then the hardware platform and application before separating each buggy peripheral. Each potential bug is marked as BurnBug (🔥) caused by either under (▽) or over (△) toggling or a BrickBug (☠). The results demonstrate that the coding styles which lead to peripheral energy bugs are prevalent, and applications with characteristics similar to the bugs presented in this paper, exist in the wild.

In our survey, the BurnBugs primarily occur because complicated applications often use simple sensor management policies. Batteryless Intermittent Actuation

(**BIA**) and Satellite Data Capture (**SDC**) re-enable all peripherals on every reboot and they remain active through relatively long operations. InK's Fast-Fourier Transform (**FFT**) and Activity Recognition (**AR**) applications enable peripherals at the start of a thread and then leave them active, burning energy unnecessarily until the device powers down. In contrast, Person Detection (**PD**) and Light Sense 5 (**L5**) are both impacted by poor peripheral driver usage. These applications read from peripherals in tight loops by calling peripheral driver functions that disable the peripheral immediately after use, even though there is not time between uses to amortize the cost. The exact impact of the BurnBug will vary depending on the peripheral power savings and restart time. Higher power and lower restart times mean a bug is more likely in the event of "under" toggling and the reverse is true for "over" toggling.

The survey also revealed BrickBug patterns in applications across a wide range of application code sizes (as measured in lines of application code, LoC). Several of the potential BrickBugs stem from the behavior of the underlying task-based runtime, InK and Chain [55, 284]. On reboot, both run an application-specific initialization function before transferring control to a task, so we assume each task was tested individually after running the initialization. Any peripheral typestate other than what is is set by the initialization function is therefore a potential BrickBug. As a result, Activity Recognition and Fast-Fourier Transform experience potential BrickBugs because the accelerometer is enabled in a task and left powered on and may impact tasks on other "task-threads". In Correlated Sensing & Reporting (**CSR**), the magnetometer is left enabled, which cauess BrickBugs on subsequent tasks and is the source of a lingering BurnBug. Several of the BrickBugs, though, stem directly from application behavior. For instance, the Gesture-driven Remote Control (**GRC**) application leaves its gesture sensor enabled if too few valid samples are collected. The Greenhouse Monitoring (**GRM**) application correctly handles a moisture sensor, but incorrectly toggles an LED which changes the energy of subsequent, atomic, sensing operations. Finally, error handling code in Facemask Monitoring (**FMM**) fails to disable the pressure sensor, and can increase the energy cost of subsequent radio transmissions. These bugs demonstrate that the code patterns leading to BrickBugs are difficult to avoid in applications with several peripherals.

Table 4.4: Peripheral Energy Bug Survey. We characterize potential bugs in terms of the software runtime (**SW**), hardware platform (**HW**), application (**App**), lines of code (**LoC**), number of peripherals (**NP**), the peripheral involved as well as it's current savings from toggling (**I**) and toggling time (**TT**). ☠ indicates a potential BrickBug, 🔥indicates a BurnBug due to either under (▽) or over (△) toggling. *Pressure sensor current is likely higher, datasheet only reports power for 1 Hz data rate, but it is running at 10 Hz.

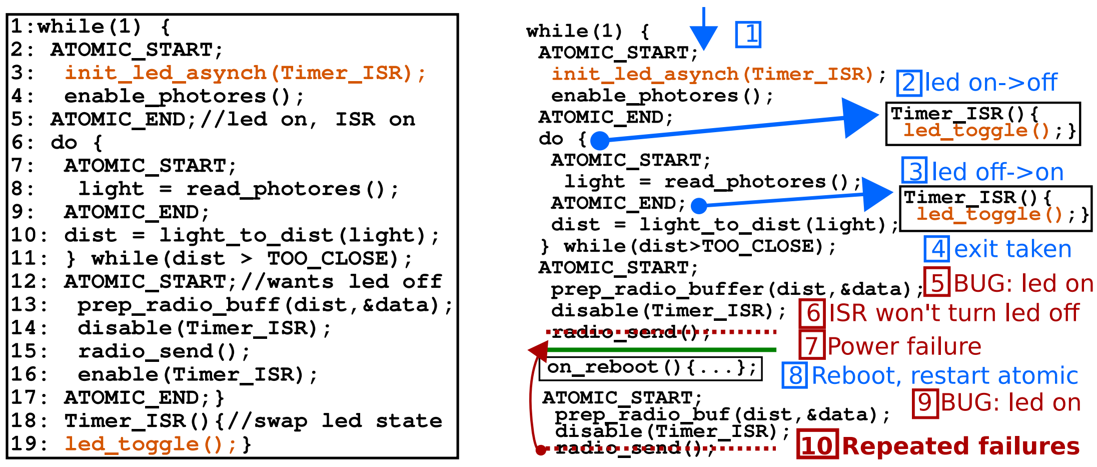
SW	HW	**App**	LoC	NP	Periph.	I(μA)	TT(ms)	Type	Description
Ink [284]	Launchpad [266]	**AR** [205]	839	2	Accel.	145	11	☠ 🔥▽	Accel. left on in nearest neighbors calculation, but not on every boot.
		FFT [204]	997	2	Accel.	145	11	☠ 🔥▽	Accel. left enabled during 128-point FFT, but not on every boot.
	InkBot [233]	**BIA** [234]	655		Gyro.	350	10	🔥▽	Gyro enabled at boot and left on during PID directed robot movements.
Chain [55]	Capybara [57]	**GRC** [134]	636	3	Gest.	830	0.6	☠	Gesture sensor left enabled if too few valid samples are collected, causes high power draw in subsequent sensing task.
		CSR [135]	524	3	Mag.	100	0.2	☠ 🔥▽	Magnetometer left on in sensing task before sending radio packet.
	EdbSat [166]	**SDC** [53]	779	4	IMU	70	19	🔥▽	IMU is enabled on every reboot, but is unused in most tasks including data compression and radio TX.
					Mag.	100	0.2	🔥▽	Magnetometer's costly initialization (includes 50 ms delay) always runs at power-up, but it's unused in most tasks.
TICS [148]	Launchpad [266]	**GHM** [147]	274	3	LED	2000	.002	☠	LED state is toggled, changing energy of subsequent sensing operation.
C++	Facebit [61]	**FMM** [60]	1226	4	Pres.	12*	0.5	☠	Pressure sensor left on during radio TX if error path is taken at line 60 of RespiratoryRate.cpp or line 66 of MaskStateDetection.cpp.
Samoyed [176]	Capybara [57]	**L5** [173]	206	1	Photo.	2	.32	🔥△	Photoresistor driver always toggles (costs 0.3 ms) despite only a 2 ms delay between photoresistor accesses and minimal current savings.
Tasks [168]	Camaroptera [193]	**PD** [72]	1435	2	Cam.	105	564	🔥△	Always toggles camera, which requires a long recalibration phase, and the shortest path between camera accesses is short in comparison.

4.6.2 BrickBugs Case Studies

Lightly condensed versions of the real code we tested are shown in Figures 4.9a and 4.10a for the Concurrent Changes and Multiple Peripherals applications. The third buggy application, Expensive Peripheral, is an expanded version of the bug described in Section 4.2. The buggy applications are similar to those examined in Section 4.6.1. The applications use no more than four peripherals, contain 250-400 lines of application code, and as detailed in Table 4.3, rely on peripherals with toggling savings similar to those in the literature.

Expensive Peripheral The first buggy application leaves an expensive peripheral, a gesture sensor [22] enabled during a critical atomic block. This application was derived from the wireless gesture-activated remote control code published in prior work [57], based on a *real* complaint about the gesture sensor driver [87]. As shown in Section 4.2, the application is bricked if the gesture sensor is on at the SVM's atomic block because the energy buffer is too small to power the gesture sensor and complete the SVM calculation.

Bug Fix: Pudu-Static reports that the gesture sensor at the SVM atomic block could be active due to the state change on line 2 of Figure 4.5, or disabled in `get_gesture` if a valid gesture is captured (not pictured). The report allows the programmer to deduce that there is an exit from the while loop that does not disable the gesture sensor. Unconditionally disabling the gesture sensor before entering the SVM's atomic block corrects the bug, i.e. disabling the gesture sensor between lines 5 and 6 of Figure 4.5.



```
1:while(1) {
2: ATOMIC_START;
3:   init_led_asynch(Timer_ISR);
4:   enable_photores();
5: ATOMIC_END;//led on, ISR on
6: do {
7:   ATOMIC_START;
8:    light = read_photores();
9:   ATOMIC_END;
10: dist = light_to_dist(light);
11: } while(dist > TOO_CLOSE);
12: ATOMIC_START;//wants led off
13:   prep_radio_buff(dist,&data);
14:   disable(Timer_ISR);
15:   radio_send();
16:   enable(Timer_ISR);
17: ATOMIC_END;}
18: Timer_ISR(){//swap led state
19: led_toggle();}
```

(a) **Original Code.**

```
while(1) {
 ATOMIC_START;
  init_led_asynch(Timer_ISR);
  enable_photores();
 ATOMIC_END;
 do {
  ATOMIC_START;
   light = read_photores();
  ATOMIC_END;
  dist = light_to_dist(light);
 } while(dist>TOO_CLOSE);
 ATOMIC_START;
  prep_radio_buffer(dist,&data);
  disable(Timer_ISR);
  ~~radio_send();~~
 on_reboot(){...};
 ATOMIC_START;
  prep_radio_buf(dist,&data);
  disable(Timer_ISR);
  ~~radio_send();~~
```

```
Timer_ISR(){
 led_toggle();}
```

(b) **Buggy Execution Trace.**

Figure 4.9: Concurrent Changes. Peripheral state changes, shown in orange, in asynchronous events can cause BrickBugs.

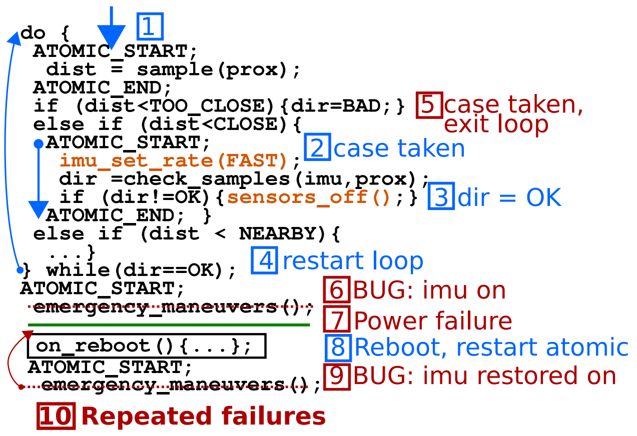
Concurrent Changes The distance sensing application in Figure 4.9 bricks due to peripheral operations in an asynchronous event. The application blinks a high power IR LED in a timer triggered ISR (line 19), then uses a low power photoresistor to measure the distance to a nearby object (line 8), and transmits a radio packet if it is too close (lines 12-17). A BrickBug occurs if the LED is on at the start of the radio's atomic block. The `disISR(Timer_ISR)` (line 14) call traps the LED on during the radio transmission, which increases the device's power draw and exhausts the energy buffer before the transmission is complete.

Bug Fix: Pudu-Static reports that the LED may be on or off at the start of the radio atomic block, with both typestate updates occurring in the `Timer_ISR`. The report indicates to the programmer that both the LED and the Timer ISR must be disabled *before the atomic block*, and only re-enabled after.

Multiple Peripherals We use an "upgraded" paper-airplane control system, Figure 4.10, to demonstrate how multiple peripherals complicate atomic block sizing. The original application used the proximity sensor to detect obstacles (line 3) and trigger emergency maneuvers (line 19) if necessary. The emergency maneuvers atomic block was sized assuming the proximity sensor is active. The system was upgraded with an inertial-measurement unit (IMU) to fold in trajectory data (line 9), and a `sensors_off` command that rapidly disables all sensors before exiting the loop (lines 10 and 16). A BrickBug occurs if the application uses the IMU, and on the next loop iteration, exits due to a proximity warning (line 5). Both the IMU and the proximity sensor will be active during the emergency maneuvers, which will exceed the energy budget.

```
1:do {//prox. starts on
2: ATOMIC_START;
3:  dist = sample(prox);
4: ATOMIC_END;
5: if (dist<TOO_CLOSE){dir=BAD;}
6: else if (dist<CLOSE) {
7:  ATOMIC_START;//imu & prox.
8:   imu_set_rate(FAST);
9:  dir = check_samples(imu,prox);
10:  if (dir!=OK){sensors_off();}
11: ATOMIC_END; }
12:else if (dist < NEARBY) {
13: ATOMIC_START;
14:  set_imu_rate(SLOW);
15:  dir = sample(imu);
16:  if (dir!=OK){sensors_off();}
17: ATOMIC_END;}
18:} while(dir==OK);
19:ATOMIC_START;//sized for prox
20: emergency_maneuvers();
21:ATOMIC_END;
```

(a) Original Code.



```
do {
 ATOMIC_START;
  dist = sample(prox);
 ATOMIC_END;
 if (dist<TOO_CLOSE){dir=BAD;}
 else if (dist<CLOSE){
  ATOMIC_START;
   imu_set_rate(FAST);
   dir =check_samples(imu,prox);
   if (dir!=OK){sensors_off();}
  ATOMIC_END; }
 else if (dist < NEARBY){
  ...}
} while(dir==OK);
ATOMIC_START;
 emergency_maneuvers();

 on_reboot(){...};
 ATOMIC_START;
  emergency_maneuvers();
```

(b) Buggy Execution Trace.

Figure 4.10: Multiple Peripherals. BrickBugs are likely if multiple peripherals are active simultaneously.

Bug Fix: Pudu-Static will report three possible typestates for the IMU at the emergency maneuvers atomic block, all stemming from inside the while loop (lines 8, 10, 14, 16). The programmer can conclude that there is an exit from the while loop that does not disable the IMU. Unconditionally running the `sensors_off` command before the emergency maneuvers fixes the bug.

4.6.3 Pudu is Effective Against BrickBugs

Table 4.5 summarizes our results, showing whether each of runtime system under test fails for each bug. Per the final column, Pudu-Static identified all of the bugs that cause existing runtime systems to fail.

Table 4.5: BrickBug Results. Pudu-Static correctly identifies all BrickBugs for both synchronous (S) and asynchronous (A) cases. ✗ indicates a failure. ∼ represents a success as a side-effect of the restoration policy. ✓ indicates successful completion or bug detection. ∗ [220], † [33], ‡ [41]

Description:	**Restop∗**	**Sytare†**	**KARMA‡**	**Pudu**	**Bug ID'd?**
Expensive peripheral (S)	✗	✗	∼	✓	✓
Multiple peripherals (S)	✗	✗	∼	✓	✓
Expensive peripheral (A)	✗	✗	✗	✗	✓
Multiple peripherals (A)	✗	✗	✗	✗	✓
Concurrent changes (A)	✗	✗	✗	✗	✓

Peripheral restoration systems incur BrickBugs. RESTOP and Sytare fail in all cases because, on reboot, they restore peripheral state to the failing state at the last checkpoint. While KARMA's lazy re-initialization policy was intended to improve performance by avoiding premature reconfiguration, the policy incidentally avoids synchronous BrickBug failures. Lazy re-initialization delays restoring peripheral state after a reboot, which, for synchronous BrickBugs (i.e. without interrupts), reduces energy consumption through the atomic region, preventing failure. This incidental benefit only holds for synchronous cases since KARMA cannot lazily initialize peripherals that may trigger asynchronously.

Pudu-Static Identifies BrickBugs. Beyond identifying BrickBugs, we show that Pudu's bug detection reports do so with few false positive reports. False positives occur if multiple typestates are possible at an atomic block, but none causes the device to consume more energy under normal operation than it can buffer . This definition is pessimistic– there is no guarantee that an arbitrary atomic block of application logic will always finish within the energy budget unless all possible peripheral typestates are tested. For instance, the only false positives retained

Table 4.6: Pudu-Static Bug Reports. Atomic blocks with multiple reaching peripheral typestates trigger Pudu-Static reports. False positives (F.P.) occur if the reported block never caused a BrickBug in testing. F.P's include HAL accesses to driver code, I/O calls, and application logic (App).

				F.P. Code Type			Filtered	
	Application:	**Total**	**F.P**	**HAL**	**I/O**	**App**	**Total**	**F.P.**
Buggy	**Expensive periph. (S)**	2	1	1	0	0	1	0
	Multiple periph. (S)	5	4	2	0	2	3	2
	Expensive periph. (A)	2	1	1	0	0	1	0
	Multiple periph. (A)	5	4	2	0	2	3	2
	Concurrent changes (A)	3	2	1	1	0	1	0
Fixed	**Expensive periph. (S)**	0	0	0	0	0	0	0
	Multiple periph. (S)	3	3	1	0	2	2	2
	Expensive periph. (A)	0	0	0	0	0	0	0
	Multiple periph. (A)	3	3	1	0	2	2	2
	Concurrent changes (A)	1	1	0	1	0	0	0

through the filter occurred in the Multiple Peripherals apps at the atomic blocks on lines 7-11 and 13-17. Both permit the IMU to run in *three* different modes. Table 4.6 elaborates on the number of reports produced for each buggy application and the corrected versions. After using the HAL-I/O filter to remove atomic blocks that are likely to be short, Pudu-Static produces few false positives for application (App) logic.

4.6.4 Pudu eliminates energy-waste due to BurnBugs

We evaluate Pudu Auto-Toggle using a suite of benchmark applications with variable time between peripheral uses and a micro-robot localization case study. The results show that Pudu Auto-Toggle adapts to changes in peripheral operating modes, and program phases better than static policies, saving energy and increasing the amount of useful work completed by the device.

Benchmark Study

We tested with two peripheral operating modes, the Gyro running at 1.6KHz and 104Hz, that represent distinct points in the trade-off between toggling cost and energy savings. Gyro 1.6KHz has a larger energy savings from toggling than Gyro 104Hz and a break-even time nearly an order of magnitude shorter. Figure 4.11 shows end-to-end execution time for each benchmark normalized to "No-Toggle", averaged over 10 trials.

For both operating modes, Pudu is faster on average (per the geometric mean) than either No-Toggle or Toggle (in the bar plots, lower is better). Operating at 1.6KHz, toggling the Gyro very often leads to an energy savings, and Pudu avoids BurnBugs due to under-toggling. When the Gyro is operating at 104Hz with a larger break-even time than at 1.6Khz (80ms vs 9ms), Pudu prevents more BurnBugs due to over-toggling. Specifically, **align**, **dmv**, **dwt** and **smv** all benefit from Pudu's measured toggling choices. Yet Pudu also tracks with Toggle on the largest benchmarks, **dmm** and **fft**, to capture the benefit of toggling during long spans of computing. For a typical application programmer, the difference between two operating modes of the same peripheral is not obvious, but the optimal toggling policy can be quite different. Pudu Auto-Toggle overcomes this problem and avoids the worst effects of BurnBugs.

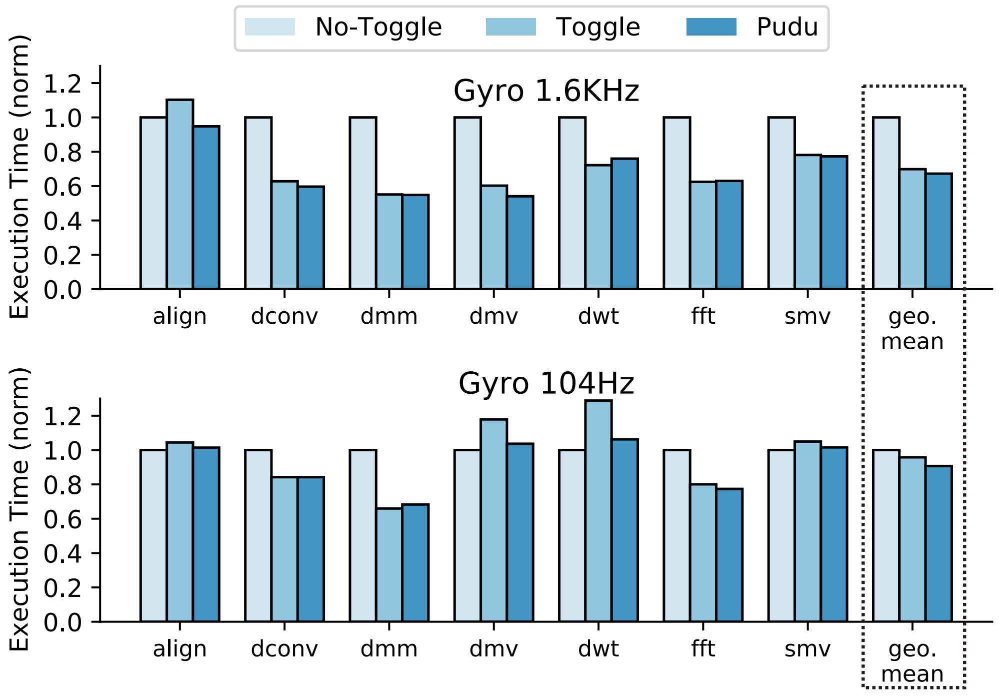


Figure 4.11: Benchmark Study. The graph compares execution time across different toggling strategies normalized to No-Toggle. Pudu Auto-Toggle adapts to changes in peripheral modes and kernel runtimes, avoiding slowdowns when static decisions are wrong.

4.6.5 Pudu Supports Complex Applications

We used the full Pudu methodology to remove peripheral energy bugs in an emulated micro-robot localization application running on an energy-harvesting Capybara sensor. The case-study shows the value of reducing energy consumption in energy-harvesting devices: reduced energy cost translates into faster run time [71].

Application Design

The application uses a particle filter [297] for robot location estimation in a stored map of illumination and floor angle data, using photoresistor and accelerometer readings as inputs. To reduce the cost of localization, the robot measures the floor slope using an accelerometer and only activates the particle filter if the slope is sufficiently steep. The application shifts from a compute intensive phase, calculating the particle filter, to a sense intensive phase, reading from the accelerometer, based on its external environment. The robot moves by activating a motion control circuit, updating its estimated location and repeating the process indefinitely. The conditional reliance on two sources of sensor data as inputs to a long-running computation makes peripheral management non-obvious, putting this application in scope for Pudu.

In lieu of a motor, our motion control circuit uses a MOSFET switch that sinks power through a resistor to capture the cost of actuation, without moving parts. The motion circuit consumes 50 μJ when activated, the same as a recent insect-scale robotics camera positioning application [123]. We configured Capybara to use its smallest capacitor bank (660 μF tantalum and 400 μF ceramic capacitors) [225] which corresponds to a powered-on period of about 150 ms and a recharge time of 2 seconds. The small bank allows the robot application to be more reactive than the large bank used in the benchmark test– with powered off periods less than 7% of the large bank.

Application Phases

The robot experiences three operating phases based on the floor slope that determine the code that follows running the motor in each iteration. **Very steep**: the robot's changes in position are likely to be large. To improve its accuracy, the application updates the particle filter and re-samples the particles after every movement. **Flat**:

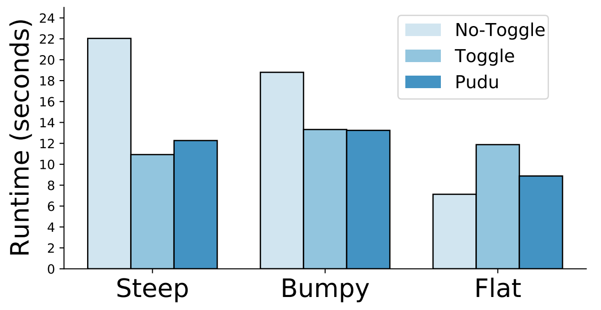
the particle filter is never triggered since the flat slope simplifies the robot's movements. Instead, one iteration in **flat** reads from the accelerometer and compares the floor slope to a threshold 100 times. **Bumpy**: to simulate an environment where the robot is forced to switch between compute and sense intensive phases, **bumpy** switches between iterations of accelerometer tracking and particle filter updates on every other iteration.

Pudu Simplifies Peripheral Management

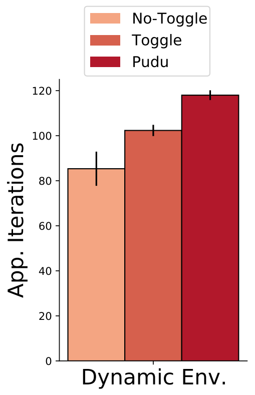
Since we used Pudu Auto-Toggle to prevent BurnBugs, we simply enable the peripherals once at the beginning of the application and run Pudu-Static to find BrickBugs. Pudu-Static confirmed that the peripheral typestates are not changed anywhere in the application, e.g. in driver code not readily visible to the programmer. The pass did not report any potential bugs among 11 atomic blocks (3 were peripheral accesses, 6 were I/O, 2 were application logic) for either of the peripherals. Pudu-Static reported the peripheral typestate at the atomic block for the motion activation (both peripherals active). We confirmed before deploying that the atomic block would finish in that typestate.

Pudu Adapts to Changing Environments.

Figure 4.12a shows the average end-to-end time to complete a single iterations of each phase tested in isolation, again, lower is better. We focus on iteration runtime because each iteration represents one movement of the robot; the faster the iterations are completed, the faster the robot's speed. Pudu achieves a lower geometric mean runtime across all three phases than No-Toggle or Toggle (11.2 seconds compared to Toggle's 12.0 seconds and No-Toggle's 14.3 seconds). Overall we find that Pudu adapts to different phases without programmer effort and avoids BurnBugs from either non-adaptive toggling solution. For instance, if the particle filter is updated (as in **bumpy** and **very steep**), Toggle outperforms No-Toggle because it is expensive to leave the accelerometer enabled. However, in the **flat** phase, heavy accelerometer use makes over-toggling expensive. In the fixed **very steep** and **flat** environments, the Toggle and No-Toggle policies, respectively, are optimal, so they each outperform Pudu Auto-Toggle because they do not have any runtime overhead. However the difference in **very steep** between Toggle and Pudu Auto-Toggle is small compared to the improvement over No-Toggle and comes at no cost to the programmer. Under **bumpy** conditions, Pudu Auto-Toggle outperforms both Toggle and No-Toggle because it learns to leave the accelerometer enabled while repeatedly reading it and to disable while running the particle filter. Still, our Pudu Auto-Toggle prototype underperforms on **bumpy** because it is slow to react to erratic mode changes. Future work should explore strategies like indexing loop exits in the Tracking Table to improve the reactivity.



(a) Runtime in Separate Phases Normalized to No-Toggle.



(b) Full App.

Figure 4.12: Robot Motion Case Study. Pudu avoids slowdowns from static toggling decisions in each app phase,(a), lower is better. Under dynamic conditions,(b), Pudu increases the work completed in a fixed time, higher is better.

To capture the full benefit of Pudu Auto-Toggle we test how the application performs when reacting to dynamic changes in the floor slope. The arrival of changes in the floor slope is simulated by raising a GPIO pin approximately every 150 seconds (the time between changes is drawn from a Poisson distribution). The pin is held high until the robot application responds to avoid missed interrupts due to intermittent execution [224]. On each change, the application steps to the next environment mode in a static list of randomly selected phases (steep, flat or bumpy). We count the number of workload iterations completed in 1000 seconds to demonstrate that improving energy efficiency increases the amount of work possible in a fixed time. Figure 4.12b shows the number of iterations completed, *higher is better*. We find that Pudu moved the robot 15% more times than Toggle and 38% more than No-Toggle because it can adapt to the changing environment at runtime. Pudu's improved peripheral management leads to better application performance with no programmer involvement.

4.7 Related Work

Pudu builds on prior work in intermittent systems and general program optimization. It is similar in spirit to work on energy optimization for embedded systems. Pudu is distinct from prior work because it identifies and corrects a new type of bug, intermittent peripheral energy bugs.

Energy Tracking in Mobile Operating Systems

Pudu is similar in spirit to prior work optimizing energy usage in mobile OS's, but it is distinct in its application. Quanto [86] observed that the power consumption of a given peripheral state is stable and can be tracked through driver code in a mobile phone. Pudu makes similar assumptions, but is designed for bare-metal systems. The Cinder mobile operating system [221] sought to automatically manage peripheral energy, but unlike Pudu, it requires invasive changes to the application. Pudu is related to efforts to improve performance under a limited energy budget by empowering the OS to divide energy among competing tasks [286, 287].

Energy Prediction and Optimization

Pudu uses state tracking similar to work in worst case time and energy estimation [75, 218, 273] and touches on some of the same challenges faced by low power RTOS systems [274, 275]. In contrast to Pudu, these works target systems where failure is not the common case. A closely related work, SysWCEC [273], uses a static analysis of peripheral activity to improve worst-case energy consumption estimation for real-time programs. Unlike Pudu's typestate analysis, SysWCEC's analysis is limited to binary peripheral activations. Numerous works have explored energy optimizations for highly energy constrained embedded systems that are orthogonal to the goals of Pudu. These include strategic sleep scheduling [177], optimal clocking [50], various compiler optimizations for the application code [116, 199], and system-wide energy optimizations for wireless sensor nodes as reviewed in [276].

Programming Languages & Optimization.

Pudu borrows terminology from work defining [11, 83, 90] and verifying typestates for objects [35, 66, 85]. Pudu Auto-Toggle's toggling decisions are inspired by the cost-benefit decisions in JIT optimizing compilers [20, 250] and feedback-driven optimizations for hardware [117, 161] and software [77, 272]. However Pudu Auto-Toggle's decision mechanism, comparing a timer to a static value, is much less resource intensive than these works.

4.8 Conclusion

Batteryless, energy-harvesting devices spend a large portion of their operating power on peripherals, but managing peripheral power in an intermittent execution is difficult and error-prone. This work is the first to study peripheral energy bugs that arise from peripheral mismanagement in intermittent systems. Specifically, we define BrickBugs and BurnBugs. BurnBugs cause slowdowns in batteryless devices and occur when programmers spend too much energy on peripherals– either due to over or under toggling. BrickBugs prevent a program from making forward progress by repeatedly restoring a failing peripheral state. We examine the prevalence of the code patterns that may cause these bugs and show that published applications for batteryless systems contain BurnBug and BrickBug susceptible code. Intermittent peripheral energy bugs are difficult for a programmer to remove on their own because the power consumption of a peripheral will vary depending on its precise operating mode.

Pudu is the first system designed to help programmers identify and overcome intermittent peripheral energy bugs by tracking peripheral typestate changes at compile time and runtime. Pudu is composed of Pudu-Static, which removes BrickBugs, and Pudu Auto-Toggle which removes BurnBugs. The Pudu flow begins by passing lightly annotated program and peripheral driver libraries to Pudu-Static which reports atomic blocks where more than one peripheral typestate is possible. The programmer iterates with the information provided by Pudu-Static to remove errant typestates or indicate that the atomic block has been tested with multiple typestates. Once free of BrickBugs, the code is instrumented with Pudu Auto-Toggle's runtime. By measuring the time between peripheral accesses, Pudu Auto-Toggle makes correct toggling decisions for each peripheral. Overall, Pudu reduces the burden of managing peripheral power in a batteryless system.

Our evaluation on real hardware demonstrates Pudu's value. We implement two state-of-the-art peripheral restoration systems and shows that Pudu-Static identifies bugs that cause applications on real hardware to fail, and does so with limited false positive reports. Likewise, we evaluated Pudu Auto-Toggle on a suite of complex computing and sensing benchmarks and showed that Pudu Auto-Toggle outperformed conventional peripheral toggling solutions. Per the geometric mean of the benchmarks, Pudu Auto-Toggle is faster than strategies that mimic novice and expert programmers across two different peripheral operating modes. Finally, we demonstrated that full Pudu flow using a micro-robotics case study. With little programmer effort, Pudu-Static verified the lack of BrickBugs in the program and Pudu Auto-Toggle improved the application performance. Pudu Auto-Toggle's ability to dynamically adapt to changes in the control flow behavior brought about by changes in the robot's environment allow it to move the robot 15% more times than an aggressive toggling strategy. The evaluation shows that peripheral mismanagement impacts application correctness and performance, but Pudu correctly manages peripheral state to curb excess power and energy consumption.

Pudu helps programmers model the behavior of a batteryless, energy-harvesting device by tracking changes in software that correspond to increases in peripheral power. This model helps programmers reason about application performance and forward progress, but it is not sufficient to capture the behavior of all batteryless systems. When using high power peripherals, more information about the exact energy/power requirements is needed because of shared state beyond data and power. As we detail in the next chapter, the state of the energy buffer is shared across all peripherals on a single device and affects what peripheral operations are safe at a given program point. To determine whether an atomic block will complete without a power failure, a system developer must consider *both* the energy and power requirements of a task.

Chapter 5

Charge Management for High Load Peripherals

As discussed in Chapter 4, intermittent systems use atomic blocks to define regions that must complete without an intervening power failure. In this chapter, we examine in detail how the voltage of the energy buffer changes as tasks are run. Figure 5.1 shows the energy buffer of a batteryless energy harvesting device as atomic tasks are run consecutively, i.e. as in systems like Alpaca which we studied in Chapter 3. As tasks execute across the top (represented by the colored blocks), they draw current and consume energy, decreasing the voltage further and further with each subsequent task. If voltage reaches the minimum threshold, V^{off}, the device powers off. After recharging, the orange task will fully re-execute. Past task-based systems mentioned in this thesis [55, 111, 168, 174] and atomic-block based languages [176] opportunistically execute tasks if the capacitor's voltage level is above V^{off}. However, Trying to execute a task with insufficient stored energy dooms the device to fail. The failed task wastes the energy spent on the operation before the power failure and imposes the cost of powering off, recharging, restarting, and re-execution. Further, as shown in Chapter 4, improperly sized atomic operations may lead to prolonged non-termination [56, 176].

Thus, systems have started to manage charge to avoid unexpected power failures using compilers [56, 175], hardware-aware runtimes [34, 281], or schedulers [120, 177, 188, 226, 284]. These works reason about energy to size tasks appropriately and execute them only when sufficient energy is available. To estimate task energy,

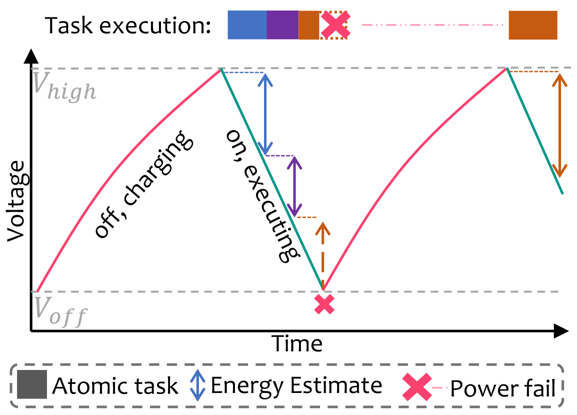


Figure 5.1: Capacitor voltage over time in an intermittent execution. Consuming energy lowers the voltage level; if it drops below V^{off}, the device powers off.

these systems use direct energy measurements, energy modeling, or programmer intervention [34, 231, 281]. Several use changes in capacitor voltage as a reasonable approximation of energy ($E_{cap} = \frac{1}{2}CV^2$). Whatever the estimation method, these systems implicitly depend on energy being the sole resource of interest for safe task execution. We find that reasoning about energy is insufficient; intermittent software systems must also consider *voltage*.

The voltage level of a device can change independently of energy consumption since a capacitor's voltage varies with current draw (or applied *load*). Even with oracular knowledge of task energy and stored energy, software may experience unexpected failures because of this load-dependent capacitor behavior. The key oversight is that a capacitor has an *equivalent series resistance* (ESR). In a load circuit, a capacitor behaves both capacitively and resistively, with ESR as its resistance. Because of the resistance, the capacitor experiences a drop in its voltage which"rebounds" to the original level once the load is removed, minus the energy used by the load.

While ESR is a well-known electrical engineering concept, no intermittent system has considered how these voltage changes impact software execution. Prior systems had low loads and low-ESR capacitors, resulting in negligible ESR-drops. However, batteryless, energy harvesting sensors must often be geometrically small, so they are increasingly adopting low-profile but energy-dense *supercapacitors* that have (relatively) high ESR [57, 64, 88, 193, 226, 281]. Furthermore, as we have discussed in this thesis, as batteryless applications become more sophisticated, they mix computing with use of sensors [22, 193], radios [112, 236] and compute hardware [166, 196], that have (relatively) high load. If either the load or ESR is high, the voltage drop due to ESR is substantial and cannot be safely discounted. Figure 5.2 illustrates this drop using a real trace of voltage over time for a load pulse applied to the Capybara power system. The energy consumed accounts for only the end-to-end drop, about 0.25 volts. The ESR-induced voltage drop (or *ESR drop*) spans a further 0.35 volts, a drop that is completely missed if a system only considers energy consumption. If this ESR drop causes the capacitor voltage to go below the system's minimum operating voltage, the system powers off *even when ample stored energy remains.*

Thus, ESR drop breaks the key assumption of charge management systems that if the device has enough energy to run a task, the task will not fail. Rather, a task safely executes only if there is sufficient energy *and* a high enough voltage level to sustain any ESR drops. These task failures can break correctness, cause poor performance, and lead to delayed response time.

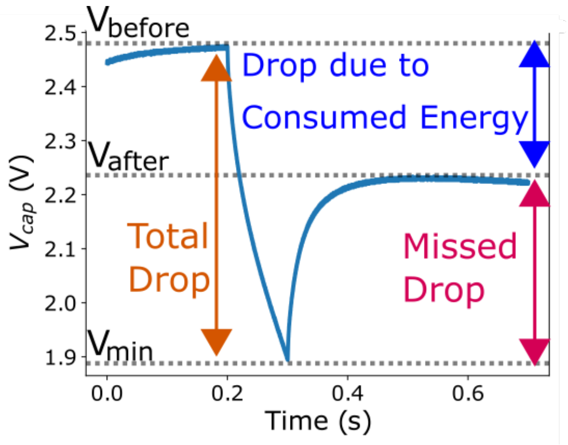


Figure 5.2: Voltage vs. time in high ESR supercapacitors. In batteryless devices with high ESR supercapacitors, considering only energy misses a key voltage drop that can lead to unexpected power failures.

For future batteryless systems to be correct, performant, and reliable, their designers must reason about the effects of voltage on software execution. However, considering low-level physical circuit properties such as capacitor ESR is burdensome for software developers. To enable developers to integrate voltage reasoning into charge management systems for energy harvesting devices, we present Culpeo. Culpeo is a hardware/software mechanism and architectural interface that provides the minimum safe voltage at which a task can execute without dropping below the operating minimum. We develop two Culpeo interface designs; the first is a compile-time profile guided analysis to reason about safe starting values, and the second is a runtime library to dynamically capture accurate starting voltage estimates. Crucially, the profile guided interface is designed to avoid profiling the load on the target device's power system, separating the concerns of the power system designer from the software developer. Instead, the power system's ESR characteristics are profiled independently of the load, and the load is characterized (e.g., on continuous power) independently of the power system. Similarly, the runtime library defines an interface that forms a clean boundary separating the exact power system dynamics from software schedulers and applications.

Our evaluation for Culpeo begins by prototyping both Culpeo designs and applying them to real energy harvesting systems and application workloads. We first show quantitatively that Culpeo produces safe starting voltages and demonstrate that disregarding voltage leads to wildly inaccurate estimates from state-of-the-art schedulers. We confirm the theoretical utility of including voltage in a scheduler by analytically correcting the feasbility analysis from a state-of-the-art scheduler. Finally, we integrate Culpeo's runtime interface into a charge management system, restoring its ESR-violated correctness guarantees in three event-driven applications. The end-to-end application experiments demonstrate that state-of-the-art charge management techniques violate application requirements without Culpeo's voltage reasoning.

5.1 Motivation

This work is motivated by the need for a hardware/software interface that exposes important power system characteristics of energy-harvesting hardware to the software layer. We first explain additional details of supercapacitor operation that were not covered in the energy harvesting power system description in Chapter 2, then elaborate on why energy-only charge management systems fail when using supercapacitors.

5.1.1 ESR Induced Voltage Drops

ESR makes a capacitor behave like a resistor (reducing current flow) as well as a capacitor (storing energy). When current is drawn from a capacitor, ESR induces a voltage drop, as in a resistor. This voltage drop does *not* actually consume (much) energy as the voltage rebounds to its original level as load decreases. If the drop causes the capacitor's voltage to sink beneath V^{off}, however, the system will power down regardless of the remaining stored energy. Ohm's Law illustrates the problem in mathematical terms: $V = IR$. We illustrate this problem in Figure 5.3. With a 10Ω ESR capacitor (in-scope of the domain) and a 50mA current draw similar to a LoRa radio, the voltage drop is 500mV. With a capacitor voltage range of 2.4V to 1.6V, this 500mV ESR drop is 62.5% of the device's operating range. This radio transmission may consume 50mA for a short duration, requiring far less energy than is stored in the capacitor (e.g., 5% of the stored energy). However, if the operation begins with a voltage lower than 64.5% of its operating voltage range (i.e., 2.12V), the ESR drop causes the system to shut down.

The key differences between ESR drop in batteryless devices and voltage fluctuation due to noise in a processor is the scale of the drop versus the core power and the symptoms that follow. Multi-core systems consuming tens of Watts of power may experience voltage fluctuations greater than 100mV [293], and if voltage drops too low, bit errors ensue. In contrast, batteryless systems consume milliwatts of power and yet may experience voltage drops greater than half a Volt. Further, if voltage drops too low the system is forced to shut down to recharge, which can violate application deadlines [177].

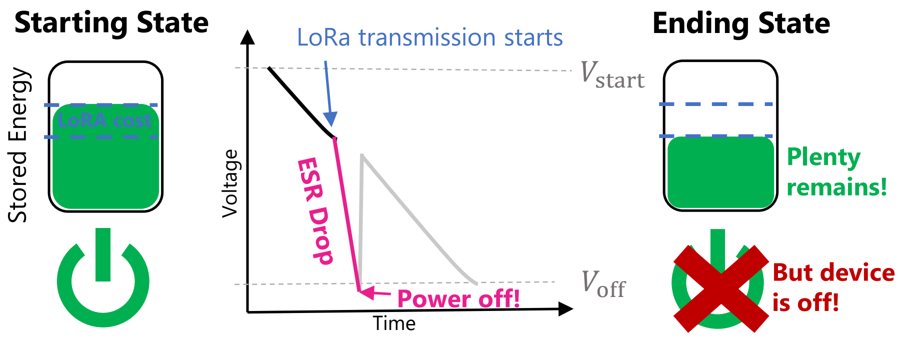


Figure 5.3: Voltage vs. Energy. Voltage drop due to ESR can cause the device to power down even when there is plenty of stored energy

Power-system designers commonly account for ESR drops due to quick, transient spikes in load-side current by adding small decoupling capacitors (around 10-100μF) close to the load-side components [21, 104, 181, 270, 282]. While adding a large amount of decoupling capacitance is the "go-to" circuit fix for load-dependent voltage drop, decoupling capacitance does not address the problem that Culpeo solves. Transient spikes draw their current from the decoupling capacitors instead of the high-ESR supercapacitors. However, our work targets *sustained* high current loads. Decoupling capacitors are typically too small to supply these sustained loads, which draw mainly from the supercapacitor. We quantitatively evaluated this effect by testing a wide range of decoupling capacitance (400uF to 6.4mF) with the Capybara [57] power system, running a 50mA-100ms load (similar to a LoRa packet) from a 33mF supercapacitor. Even with an abnormally high 6.4mF of decoupling capacitance, we still observed an ESR drop of 200mV, 20% of the device's operating range.

Simply adding a safety margin (e.g., provisioning extra energy) is also an inadequate solution. Provisioning unnecessary energy will make the entire system inefficient while still not guaranteeing correctness; a larger ESR drop that spills over the safety margin could still happen. Further, the programmer has little guidance on how much extra energy to provision for safety. The limited data in capacitor datasheets make handling ESR a guessing game for application developers, even if they are aware of the voltage drop effect [23, 81, 137, 189, 235]. While industry hardware designers perform expensive characterizations of ESR across frequency, temperature, humidity and lifetime, this information is not accessible to software designers [270]. We argue that a more practical approach is to provide software developers with an interface to reason about load-dependent ESR drops and we demonstrate that lacking such an interface, prior systems fail.

5.1.2 Disregarding Voltage Breaks Past Systems

Prior energy-harvesting systems only modeled incoming (recharging) and outgoing (computation) energy, without considering circuit-level characteristics like ESR. Considering only energy and disregarding ESR drop causes charge management systems like schedulers to fail frequently.

ESR drops violate the core assumption of schedulers for intermittent systems that if the buffer holds more energy than what upcoming task will consume, the task will successfully execute [48, 49, 96, 163, 177, 188]. As a concrete example, we consider the scheduler CatNap [177], which adapts RTOS's feasibility scheduling [289] for intermittently powered systems. CatNap looks at the energy consumed by high priority tasks and their deadlines, determining if it is possible to schedule tasks and recharges so that there is always energy to run the tasks at the appropriate time.

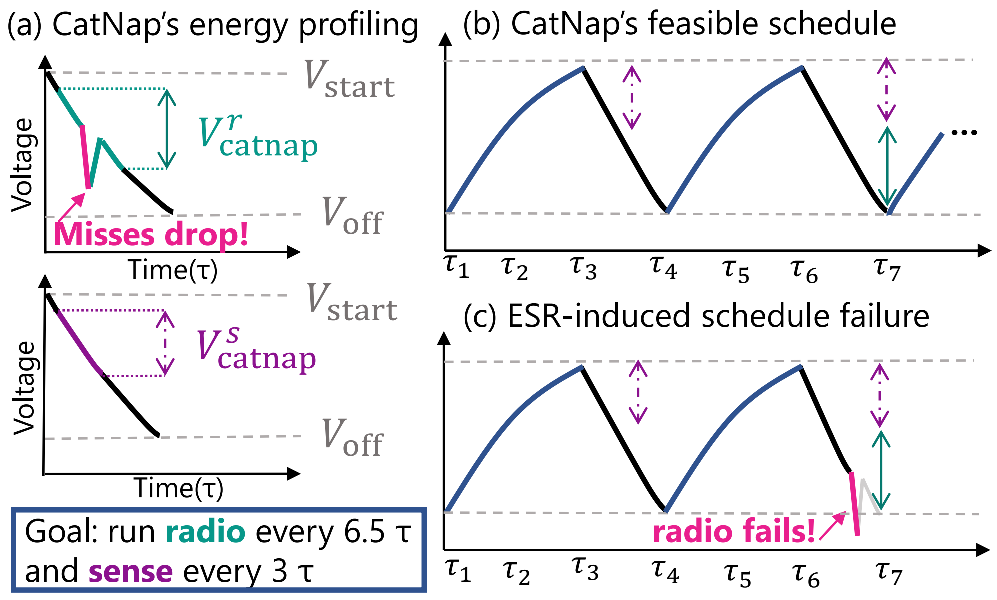


Figure 5.4: Catnap Protocol. CatNap's feasibility test will lead to failed executions

Figure 5.4 shows how CatNap's careful scheduling still results in a task failure, using an application that we evaluate in Section 5.5. CatNap must determine if it is possible to schedule two tasks: `radio` which is triggered by an interrupt with an interarrival time of 30 seconds, and `sense` repeats every 6 seconds. CatNap estimates the energy costs of the tasks by measuring voltage at the start and end of each task's execution, in Figure 5.4 (a). The graphs are of voltage over time, with the energy estimate for `radio` indicated by a green solid arrow and `sense` with a purple dashed arrow. Figure 5.4 (b) shows CatNap's feasible schedule of the tasks interspersed with sleeping to recharge—based on energy estimates alone, `sense` followed by `radio` should complete without failing a $t = 30$ Figure 5.4 (c) shows how this schedule will fail due to ESR. While there is sufficient *energy* for `radio`, the scheduler executes at a *voltage* too low to sustain the ESR drop, causing a failure. To be correct, CatNap and other schedulers must ensure that the starting voltage level is high enough to satisfy ESR drops as well as the consumed energy.

Running a task at a voltage too low to sustain ESR drops is not an edge case. Figure 5.5 shows the error between actual safe voltages to start running a task and those predicted by energy-based estimates, for a series of load profiles run on the Capybara power system. If the error is positive, the task fails to complete. We provide details on the task profiles in Section 5.5; at a high level, the profiles comprise different combinations of pulse width, intensity, and load shape. Direct energy estimates fail across the board, and voltage-based energy approximations like CatNap are highly dependent on how quickly they measure capacitor voltage after the task completes. A quick measurement can capture the voltage level before rebound, resulting in a highly conservative energy estimate that accounts for the voltage drop as a side-effect. CatNap-Measured reports the voltage estimates by the published CatNap implementation [177], and CatNap-Slow reports estimates if there is a 2 ms delay between a task's completion and the measurement. Whether a task's energy cost is obtained through direct measurement or through using voltage as a proxy, determining the safe starting voltage by energy cost alone results in task failure most of the time.

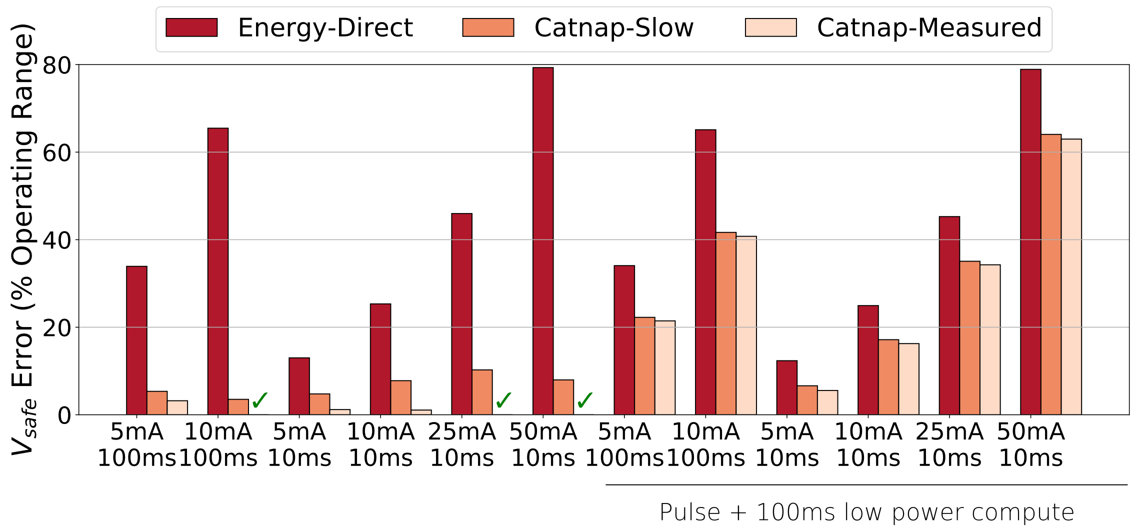


Figure 5.5: Safe voltage estimates from prior work. Estimating the safe voltage at which to run a task by energy costs alone results in wildly incorrect predictions. Depending on the implementation, voltage-based energy approximations like CatNap can fare better, but are highly dependent on the implementation.

Incorrect task starting voltage estimates have profound impacts on real application performance, as illustrated by a simple experiment. We tested three schedulers

Table 5.1: Impact of safe starting voltages. Accurate task starting voltages allow the application to capture interrupts and prevents unexpected failures.

System	% Events Captured (out of 400)	Up-time (out of 40s)	Unexpected Failures
Safe	100%	100% (40s)	0
CatNap	5.73%	5.98% (2.39s)	8
ChargeToFull	4.1%	6.4% (2.56s)	0

running an application on a Capybara that repeatedly performs a compute task on an accelerator, while the primary MCU handles sensor interrupts that arrive every 10ms. This experiment uses the hardware setup detailed in Section 5.4. The first (Safe) runs the accelerator if capacitor voltage exceeds the known safe voltage, CatNap runs if voltage exceeds an estimate based on voltage-as-energy, and ChargeToFull turns off to fully recharge before each task. Table 5.1 summarizes the results of the experiment running for 40 seconds. The accurate estimate captures *100%* of interrupts, because it can remain on and available while charging, as shown by its 40 seconds of up-time. In contrast, CatNap fails unexpectedly every time it powers on because its estimates do not consider ESR drop and captures less than 6% of interrupts. ChargeToFull avoids unexpected failures by performing a controlled shutdown after each task to recharge. However, the total time spent powered off accumulates: ChargeToFull is powered off charging for 94% of the time and misses 96% of interrupts. In summary, preliminary results indicate that existing schedulers perform poorly in the presence of ESR and simple scheduling alternatives sacrifice up-time for predictability.

5.2 Culpeo Overview

Culpeo provides hardware-software interfaces that capture the voltage and energy requirements of a software task. Culpeo's interfaces record a per-task ESR drop,V^{δ}, and produce a safe starting voltage, V^{safe}, that accounts for ESR drop. We define V^{safe} and V^{δ}, describe our compile-time and runtime calculation approaches, and demonstrate applying V^{safe} and V^{δ} to fix existing intermittent system schedulers.

5.2.1 Defining V^{safe} and V^{δ}

V^{safe} is the minimum voltage level that the system must supply for a task to complete its execution, accounting for both consumed energy and ESR drop. V^{δ} is the difference between the minimum voltage during a task (V^{min}) and the final voltage once the task completes and the voltage rebounds (V^{final}), shown in Figure 5.6. V^{safe} may be defined in terms of a single task or a series of tasks ($V^{\mathsf{safe}}_{multi}$). For a single task, starting at V^{safe} guarantees that the task will complete without voltage dropping below V^{off}. Calculating $V^{\mathsf{safe}}_{multi}$, the voltage level at which a *sequence* of tasks is safe to execute, is more complex, requiring per-task V^{δ} information.

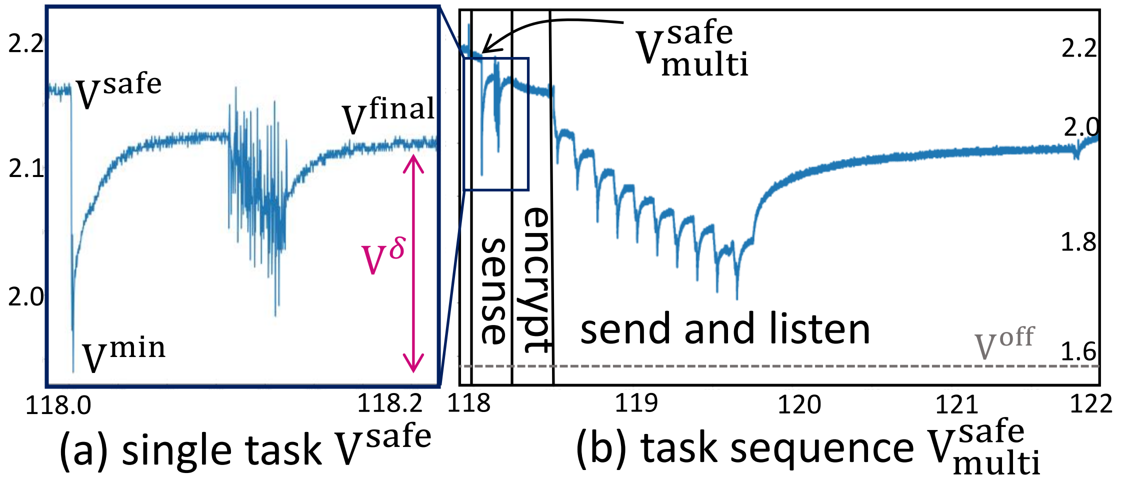


Figure 5.6: Culpeo Definitions. V^{safe} guarantees that a task will complete, but the V^{δ} parameter is required to calculate $V^{\mathsf{safe}}_{multi}$, a safe voltage for a sequence of tasks.

Formulating $V^{\mathsf{safe}}_{multi}$ requires determining a voltage level for each task sufficient to satisfy its voltage requirements *and* meet the voltage requirements of any subsequent tasks. In other words, for the initial task in the sequence:

$$V^{\mathsf{safe}}_0 = V(E_0) + penalty_0 + V^{\mathsf{safe}}_1$$

Here, $V(E_0)$ is the voltage required to satisfy the energy consumed by the task. Adding voltage to account for ESR drops depends on the V^{safe} of the subsequent task, V^{safe}_1. If V^{safe}_1 is already high enough to sustain the current task's V^{δ}, then simply adding V^{safe}_1 is sufficient. Otherwise, an additional *penalty* needs to be added to raise the voltage level so that the drop will not cross V^{off}.

$$penalty_0 = \begin{cases} V^{\mathsf{off}} + V^{\delta}_0 - V^{\mathsf{safe}}_1, & \text{if } V^{\mathsf{off}} + V^{\delta}_0 > V^{\mathsf{safe}}_1. \\ 0, & \text{otherwise.} \end{cases}$$

At the end of the task sequence, voltage must be high enough that meeting the last task's voltage requirements results in a voltage at or above the minimum operating threshold. $V^{\mathsf{safe}}_{\mathsf{final}} = V(E_{\mathsf{final}}) + penalty_{\mathsf{final}} + V^{\mathsf{off}}$. $V^{\mathsf{safe}}_{multi}$ is the summation of the voltage needed to satisfy the energy and ESR drops for each task in a sequence:

$$V^{\mathsf{safe}}_{multi} = \sum_{i=0}^{n} V(E_i) + \sum_{i=0}^{n} penalty_i + V^{\mathsf{off}}$$

If the voltage at the start of a sequence of tasks ϵ is $\geq V^{\mathsf{safe}}_{\epsilon}$, then the voltage will not dip below V^{off} while executing the tasks. As a proof sketch that $V^{\mathsf{safe}}_{multi}$ is correct, assume that for some task i in the sequence, the voltage after running i is less than the threshold, i.e., $V^{\mathsf{safe}}_i - V(E_i) - penalty_i < V^{\mathsf{off}}$. By definition, $V^{\mathsf{safe}}_i = V(E_i) + penalty_i + V^{\mathsf{safe}}_{i+1}$. Simplifying the equations results in $V^{\mathsf{safe}}_{i+1} < V^{\mathsf{off}}$. As no part of V^{safe}_{i+1} is negative, and the base case is at least V^{off}, this equation results in a contradiction.

5.2.2 Capturing V^{safe} and V^{δ}.

Culpeo provides two approaches to find accurate V^{safe} and V^{δ} values. The first is a profile guided method that determines V^{safe} at compile-time, which we call Culpeo-Profile Guided (Culpeo-PG). The second, Culpeo-Runtime (Culpeo-R), measures V^{min} and V^{final} at runtime and calculates V^{safe} onboard the MCU. Section 5.5 confirms that both approaches produce accurate V^{safe} levels.

Culpeo-PG relies on static profiling to provide V^{safe} and V^{δ} at compile time. The inputs to Culpeo-PG are generated separately by the *power system designer* and by the *application developer*. The power system designer produces a measurement-based characterization of the energy buffer and output booster behaviour. The application developer measures a current profile (using any power system) of each program task. A key advantage of Culpeo-PG's approach is that it separates the concerns of the power system and the application. Culpeo-PG then uses the task current traces and the power system model to calculate energy estimates and ESR drops, which yield each task's V^{safe}. Effectively, Culpeo-PG performs a $V^{\mathsf{safe}}_{multi}$ calculation at the granularity of the trace (e.g. 125kHz), working backwards in time. At each step, Culpeo-PG computes the ESR drop. If the ESR drop is greater than the last computed V^{safe} the algorithm adds the difference as the penalty. The advantage of Culpeo-PG is that it allows application developers to calculate V^{safe} values prior to deployment using a continuous power system, but the estimates are limited by the accuracy of the statically profiled inputs. For estimates that better capture effects of the deployment environment, we present Culpeo-R.

Culpeo-R profiles the capacitor voltage online to capture the starting (V^{start}), minimum, and final voltages during the event execution. To capture V^{min}, Culpeo-R repeatedly samples from an ADC connected to V^{cap} and records the minimum observed voltage. Culpeo-R then uses the output booster model to map the energy and ESR drop requirements from V^{start} down to the bottom of the capacitor voltage range (i.e. $V^{\mathsf{min}} = V^{\mathsf{off}}$), resulting in V^{safe}. Capturing V^{safe} at runtime reduces error from physical effects such as supercapacitor aging that are not easily captured by static profiling.

5.2.3 Applying V^{safe} and V^{δ}.

To enable system designers to integrate voltage information into software systems, Culpeo provides two interfaces, shown in Figure 5.7. The first is a library used by scheduler code, and the second is a measurement interface (e.g. as described in Section 5.2.2) between the scheduler and the power system. The Culpeo library code consists of simple functions that allow Culpeo-R to track V^{safe} and relay conclusions about the safety of calling a given task. The scheduler uses the Culpeo library to make ESR-aware scheduling decisions, e.g., to check whether V^{cap} exceeds V^{safe} for the next task to schedule, pausing to recharge if not. The measurement interface includes low level software and hardware to query the power system and calculate V^{safe} and V^{δ}. The measurement interface provides the same functionality regardless of its implementation as Culpeo-PG or Culpeo-R, but the exact division of hardware/software responsibilities varies as we explain in Section 5.3.

5.3 Culpeo Design

Culpeo interfaces model a device's power system and profile a program's tasks to ensure that the device's voltage level is sufficient for each task to complete. This

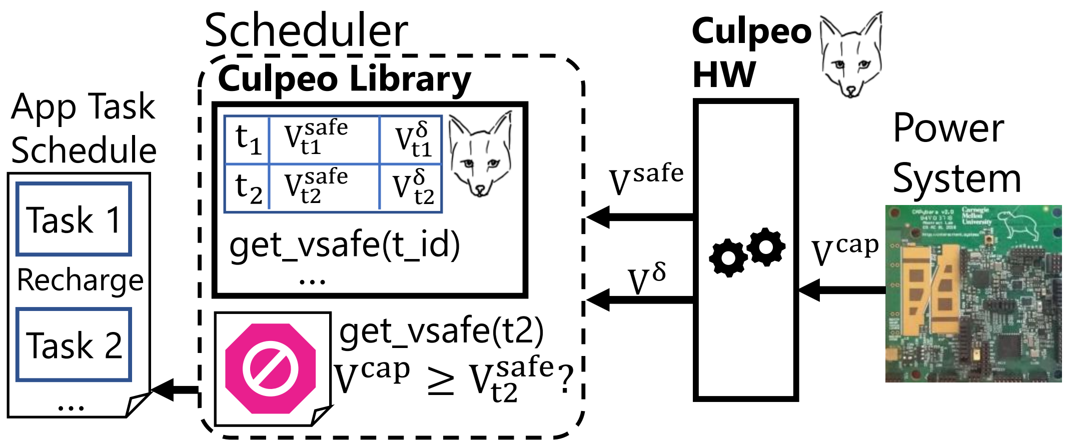


Figure 5.7: Culpeo Integration. Culpeo is integrated with a scheduler using the Culpeo hardware and software components.

section describes several possible implementations of Culpeo interfaces.

5.3.1 Modeling the Power System

To calculate V^{safe}, all Culpeo implementations model the device's energy buffer and output booster (Figure 2.4). For the input booster, Culpeo-PG assumes a worst case of no incoming power, and Culpeo-R assumes the input is stable.

The energy buffer (i.e., capacitor) is modeled differently in each Culpeo variant due to differences in the V^{safe} calculation algorithms. Culpeo-R models the energy buffer with no knowledge of the exact capacitance or resistance. It simply assumes that the buffer is an ideal capacitor in series with a variable resistor which takes unknown, time varying values. Culpeo-R ignores some non-idealities of supercapacitors, e.g., leakage [133] and charge speed effects [9], relying on the ideal $I = C\frac{dV}{dt}$ equation for capacitor analysis. Culpeo-PG models the energy buffer based on its capacitance, C, and its ESR as an ideal capacitor in series with a resistor. The value of C comes from the capacitor's datasheet and is generally conservative [23, 137, 235]. Using datasheet ESR values is too inaccurate; the ESR experienced by a load changes with the load's frequency, but many datasheets do not supply the full spectrum [23, 137, 235]. Further, small decoupling capacitors throughout the power system also affect ESR. We instead derive a curve of ESR versus frequency via direct measurement of the power system [81, 241] and fit the results to a logarithmic curve. Essentially, we apply load pulses of different widths (1ms to 1s) and amplitudes (5mA to 50mA) to the bank and measure the resulting voltage drop [36] to calculate ESR using Ohm's law. Figure 5.8 shows the resulting curve for the Capybara power system attached to a fixed size 45 mF, high ESR capacitor bank [235]. To choose a representative ESR value from the curve, Culpeo-PG uses the width of the largest current pulse, excluding high frequency noise.

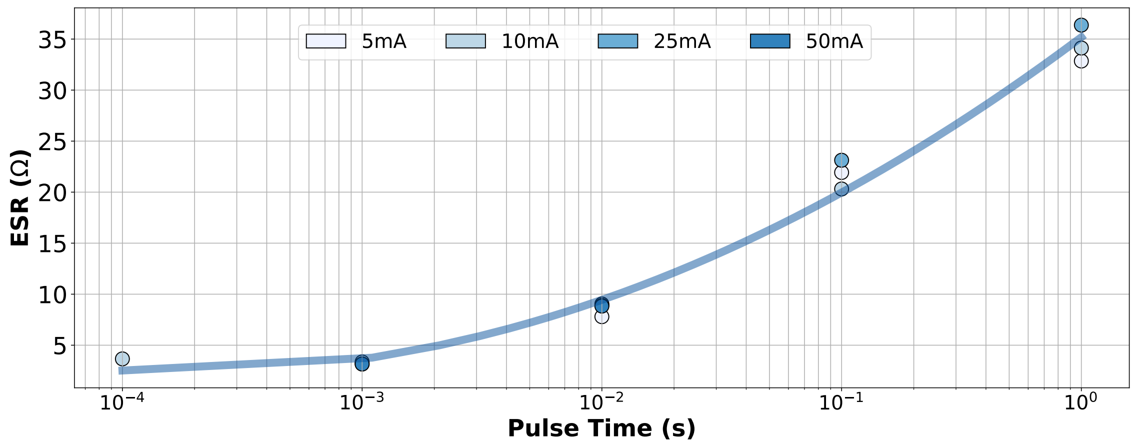


Figure 5.8: ESR versus pulse width. This graph shows the change in ESR with pulse width, and its logarithmic fit, for Capybara's capacitor 45mF bank.

To model the output booster, the power system designer sets V^{high}, the capacitor's highest voltage, and V^{off}, the voltage at which the output booster turns off. The designer also sets V^{out}, the output voltage of the output booster, which is used in conjunction with the current profile to determine P^{out}, the power delivered to the load. Culpeo uses datasheet booster efficiency curves to relate P^{in}—the power drawn from the energy buffer to the output booster—to P^{out} as the energy buffer voltage V^{cap} declines over the course of an operation. We assume the output booster has little change in efficiency w.r.t. current [2, 3], so efficiency can be modeled as a line relating input voltage to efficiency (i.e. $\eta = mV + b$) at a single current value. Combining this output booster model with the energy buffer model, Culpeo can predict the behavior of the power system in response to an arbitrary task load.

5.3.2 Culpeo-PG V^{safe} Calculation

The programmer uses Culpeo to profile software tasks and compute V^{safe} at compile-time. A scheduler can then use V^{safe} to decide when to execute a task.

Culpeo-PG interfaces with current measurement instruments [246] to capture a task's worst-case current trace. Capturing a current trace is simple because the energy consumption of a task in an intermittent execution is bounded by the device's energy capacity. Moreover, prior work showed that tasks are easily characterized using "knob" values [176, 177], in which a single variable largely dictates the energy consumption (e.g., the input dimension of a matrix-matrix multiplication kernel). Our Culpeo prototype collects a task current trace at 125kHz and then selects an ESR value from the power system's ESR curve.

Culpeo determines V^{safe} by iteratively calculating the voltage drop due to *energy consumption* and due to *ESR drop* at each step of the current trace. Algorithm 3 describes how the energy and voltage penalty are calculated. The algorithm starts using the power system model provided by the power system designer (P) and the current trace collected by the application developer (I). At each time step, dt, Culpeo calculates E using the output booster efficiency, η, given that $P^{\mathsf{in}} = P^{\mathsf{out}}/\eta$. Next, Culpeo estimates V^{cap} to calculate the current drawn from the capacitor, because $P^{\mathsf{in}} = I_{in} \times V^{\mathsf{cap}}$. Culpeo must consider V^{cap} when assessing the current from the capacitor to the output booster, because as V^{cap} decreases, the booster draws more current from the capacitor; as current increases, so too does ESR drop. Finally, Culpeo calculates the voltage penalty, which guarantees that the new V^{safe} satisfies the energy requirements of the next step, $V[i+1]$ and can sustain the ESR drop in the present step.

As shown in Section 5.5, Culpeo-PG produces accurate V^{safe} calculations for a recently profiled capacitor. However, Culpeo-PG assumes a static ESR model, but supercapacitor ESR and nominal capacitance change over the device lifetime (years). Capacitance can reduce to <80% of nominal and ESR can increase to double its nominal, beyond which the capacitor is considered dead [81, 189, 241]. Designers may obtain a more conservative model than Culpeo by derating the capacitor bank and ESR for aging effects. However, a runtime V^{safe} calculation captures aging effects without derating by rerunning periodically.

5.3.3 Culpeo-R Calculation

The goal of the Culpeo-R V^{safe} calculation is to allow the system to profile tasks starting at an arbitrary capacitor voltage and map the measured start, end and

Algorithm 3 Culpeo V^{safe} algorithm

```
1: function CulpeoVsafe(CurrentTrace $I$, PowSys $P$)
2:     $V \leftarrow \varnothing$                                  ▷ Initialize safe starting voltages
3:     $C \leftarrow$ GetCap($P$)                                    ▷ Get capacitance value
4:     $R \leftarrow$ GetESR($P$,$I$)                                 ▷ Get freq. dependent ESR
5:     for $i \leftarrow len(I)..0$ do                                ▷ Reverse through trace
6:         $E \leftarrow I[i] * V^{\mathsf{out}} * dt/n$                 ▷ Energy consumed by step i
7:         $V^{\mathsf{cap}} \leftarrow$ EstVcap($P$,$I[i]$,$V[i+1]$)           ▷ Estimate $V^{\mathsf{cap}}$
8:         $I_{in} \leftarrow I[i] * V^{\mathsf{out}}/\eta(V^{\mathsf{off}}) * V^{\mathsf{cap}}$          ▷ Current out of cap.
9:         $V^{\delta} \leftarrow I_{in} * R$                            ▷ Voltage drop from ESR
10:        $V_{penalty} \leftarrow \max V^{\mathsf{off}} + V^{\delta}, V[i+1]$            ▷ Voltage penalty
11:        $V[i] \leftarrow \sqrt{2 * E/C + V_{penalty}^2}$
12:    end for
13:    return $V[0]$
14: end function
```

minimum voltages to V^{off}. Changes in efficiency as the input voltage declines make the mapping non-trivial. Culpeo-R makes several assumptions to keep the code running on the MCU practical. The first is that efficiency decreases monotonically with voltage; since Culpeo-R approximates efficiency as a line, this assumption holds as long as the slope of the line is positive. The second is that harvested power is roughly constant during the event execution. This assumption is reasonable as the supercapacitor-enabled devices Culpeo targets generally rely on more powerful, slowly changing energy sources (e.g. solar power) than low-end batteryless motes. Culpeo-R produces different V^{safe} values for different levels of incoming power, so it is best to use Culpeo-R in conjunction with scheduler policies that re-profile as harvestable power changes [177].

Culpeo-R separates the worst case ESR drop, V^{δ}, from the energy induced voltage drop, V_E^{safe} and calculates them independently before adding the effects back together. First, we calculate the new V^{δ} in terms of the current, i_{load},the ESR R, and the efficiency at the event's V^{min}, $\eta(V^{\mathsf{min}})$, as shown in Equation 5.1a. Intuitively, as efficiency decreases with V^{min}, V^{δ} gets larger. This expression for V^{δ} is rooted in Ohm's law and converter efficiency, namely $V^{\mathsf{out}}I_{out} = V^{\mathsf{cap}}I_{in}\eta(V^{\mathsf{cap}})$, but it abstracts away the exact current profile and ESR value. We use this definition to allow us to define V^{δ}safe in terms of the original V^{δ} without directly measuring the current trace, as shown in Equation 5.1b.

$$V^{\delta} = \frac{i_{load} * R * V^{\mathsf{out}}}{V^{\mathsf{min}} * \eta(V^{\mathsf{min}})} \qquad V^{\delta}_{safe} = \frac{i_{load} * R * V^{\mathsf{out}}}{V^{\mathsf{off}} * \eta(V^{\mathsf{off}})} \tag{5.1a}$$

$$V^{\delta}_{safe} = V^{\delta}(\frac{V^{\mathsf{min}}\eta(V^{\mathsf{min}})}{V^{\mathsf{off}}\eta(V^{\mathsf{off}})}) \tag{5.1b}$$

In addition to the voltage drop caused by ESR, Culpeo-R must consider the voltage drop caused by actual energy expenditure. However, instead of predicting a drop due to energy, Culpeo-R solves for V^{safe}_{E} based on the assumption that the energy delivered to the load, E_{out}, is constant across all input voltages. Equation 5.2a defines E_{out} using the load current; we change variables from time to voltage and redefine E_{out} in Equation 5.2b. Finally, we set as equal the integrals that represent the profiling execution (V^{start} to V^{final}) and what would be the execution starting at V^{safe} (V^{safe} to V^{off})(Eq. 5.2c). The goal now is to solve for V^{safe} by resolving both definite integrals, since V^{off} is given and V^{start} and V^{final} are quantities the Culpeo interface can measure.

$$\text{Use } P_{out} = V^{\mathsf{cap}} I_{in}\eta(V^{\mathsf{cap}}) \qquad E_{out} = \int_{t_{start}}^{t_{end}} \eta(V(t))V(t)i_{in}(t)dt \tag{5.2a}$$

$$\text{Apply } I = C\frac{dV}{dt}\text{, change of var.} \qquad E_{out} = C\int_{V^{start}}^{V^{final}} \eta(V)VdV \tag{5.2b}$$

$$\text{Set Equal} \qquad C\int_{V^{\mathsf{off}}}^{V^{\mathsf{safe}}} \eta(V)VdV = C\int_{V^{\mathsf{final}}}^{V^{\mathsf{start}}} \eta(V)VdV \tag{5.2c}$$

However, even with a linear $\eta(V)$ function, solving Equation 5.2c requires multiple cubic root operations that are expensive for the low power microcontrollers that Culpeo targets. Instead, we approximate the solution as:

$$(V^{\mathsf{safe}}_{E})^2 = \frac{\eta(V^{\mathsf{start}})}{\eta(V^{\mathsf{off}})}((V^{\mathsf{start}})^2 - (V^{\mathsf{final}}))^2 + (V^{\mathsf{off}})^2 \tag{5.3}$$

Effectively, we solve Equation 5.2c after collapsing $\eta(V)$ into a constant. We use

$\eta(V^{\mathsf{start}})$ on the left and $\eta(V^{\mathsf{off}})$ on the right because they can be known quantities that can be calculated at compile time. V^{off} is set by the power system designer, and Culpeo-R may choose a known V^{start} to run the event. We finally define V^{safe} as:

$$V^{\mathsf{safe}} = V_E^{\mathsf{safe}} + V_{safe}^{\delta} \tag{5.4}$$

5.3.4 Culpeo-R Implementation

We propose two implementations for Culpeo-R; the first uses interrupt triggered measurements and the second uses a custom microarchitecture to reduce power and software overheads. Both systems rely on the same set of runtime library calls listed in Table 5.2, but with different underlying implementations. The scheduler writer uses these calls to access V^{safe} and V^{δ} data. `profile_start()` begins profiling a segment of code and `profile_stop(id)` ends the profiling and records the results in the task measurement fields of the Culpeo table, indexed by task identifier `id`. `do_calc(id)` performs V^{safe} and V^{δ} calculations if the task's measurement fields are populated, storing the results into the table. Finally, the `get` functions retrieve V^{safe} and V^{δ} values from the table if valid values exist, otherwise returning V^{high} and -1. Similar to Culpeo-PG, Culpeo-R assumes that a task profile captures the worst case voltage drop. Based on this assumption, schedulers need only profile tasks when tasks are first issued from the outer control loop and again if substantial changes in harvested power occur. Further, Culpeo-R assumes that each task will terminate within the available energy buffer voltage, e.g. $V^{\mathsf{safe}} < V^{\mathsf{high}}$. Developers can use tools like Culpeo-PG to predict task completion at compile time or use Culpeo-R before deployment to scale tasks.

Table 5.2: Culpeo-R Runtime calls. `id` is a task identifier.

Purpose	**Functions**
Profile:	`prof_start()` `prof_end(id)` `rebound_end(id)`
Calculate:	`do_calc(id)`
Use:	`get_vsafe(id)` `get_vdrop(id)`

Interrupt Driven

The interrupt-driven implementation (Culpeo-R-ISR) relies on an interrupt service routine (ISR) associated with a hardware timer that reads from the ADC and updates the minimum observed voltage. `profile_start()` sets the minimum observed voltage to infinity, enables the 1 ms timer to trigger the profiling ISR, configures an ADC (on- or off-chip) to be read from quickly in the ISR, and reads from the ADC to record V^{start}. `profile_end(id)` disables the timer interrupt and ADC and sleeps to allow the capacitor voltage to recover from any ESR drop. As the MCU sleeps, it awakens once every 50 ms to read once from the ADC and update a *maximum* observed voltage. Sleeping between ADC samples minimizes the energy consumed by the MCU to ensure an accurate V^{final}. The scheduler runs `rebound_end(id)` to exit sleep when the capacitor voltage stops increasing, and V^{final} [id] is then set to the maximum value.

We implemented a Culpeo-R-ISR interface on an MSP430 microcontroller and show in Section 5.5 that it substantially improves the performance of event driven applications on an energy harvesting device. However, Culpeo-R-ISR has several drawbacks. The first is that the on-chip ADC in most microcontrollers is relatively high power [280] which limits the profiling frequency and accuracy for tasks with small ESR-drops (e.g. compute tasks). Second, the MCUs we target are in-order, single-threaded cores, so time spent sampling the ADC in software is time taken from the application. Further, not all applications written for a low power MCU tolerate external interrupts, leading to bugs [1]. Third, monopolizing the only ADC is not an option if a task needs it. Many MCUs can multiplex ADC access [264], but this can increase the sampling delay for an application and force a programmer to rewrite their ADC driver.

Custom Microarchitecture

To overcome the challenges presented by Culpeo-R-ISR, we propose a custom, on-chip peripheral to measure the energy buffer voltage without interfering with the application code running on the MCU. Figure 5.9 shows a detailed view of the proposed circuit. The Culpeo μArch is a memory mapped peripheral block that measures V^{cap} with an 8-bit ADC and uses a digital comparator to automatically capture a minimum (or maximum) value. The Culpeo peripheral block includes a high impedance input buffer to minimize leakage from the capacitor, and an 8-bit min/max capture register. The core interacts with the block by writing to a memory mapped control register and provides a 100kHz clock to trigger the ADC sampling.

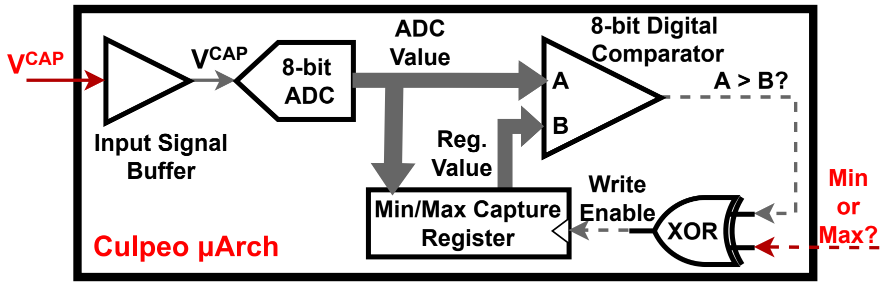


Figure 5.9: Culpeo μ Arch The Culpeo μ Arch is a low overhead design that uses an 8-bit ADC, a comparator and a single register to track capacitor voltage for Culpeo-R. Red arrows indicate inputs, solid arrows are analog signals, dashed arrows are boolean and wide arrows are 8-bit buses.

Table 5.3 shows the low-level driver commands that interface with the block's control signals and are used to implement the Culpeo-R runtime library. `Configure` and `Start Task` are used to implement `profile_start()` and will configure the block into minimum tracking mode and read the current ADC value into a data register. Software should immediately read the value and store it elsewhere to prevent it from being overwritten. Software will issue an `End Task` command from within the `profile_end(id)` function to write the minimum value out and switch the block to maximum tracking. Unlike Culpeo-R-ISR, the peripheral block continues running and will not end the rebound tracking until it receives an `End Rebound` command as part of the `rebound_done(id) call`. Waiting until a rebound done call gives the scheduler more flexibility in capturing V^{final}. The block, as we show shortly, is low power and may be kept enabled indefinitely, so the scheduler may choose to run another task immediately instead of waiting to capture a more accurate (higher) V^{final}.

This new peripheral block eliminates pain points from the interrupt-based approach. The comparator eliminates software interaction during the task; the scheduler only interacts with the peripheral before tasks begin and after they complete. Moving to a dedicated, modern, 8-bit ADC reduces power substantially and eliminates resource contention while adding minimal area. Recent work demonstrated an 8-bit ADC in a 130 nm process that consumes only 140 nW at an area of $0.01mm^2$ [82, 190].

Table 5.3: Culpeo on-chip command interface Culpeo on-chip is a memory-mapped peripheral with two control and data registers.

Command	Description
Configure	Select clock for ADC and enable ADC sampling
Start Task	Write ADC value to data register, switch to minimum sampling, reset min/max register
End Task	Write min/max to data register, switch to maximum sampling, reset min/max
End Rebound	Write min/max to data register, disable ADC

For reference, TI's MSP430 "Wolverine" series is also built in a 130 nm technology [39], but its ADC consumes 180 μW at minimum.

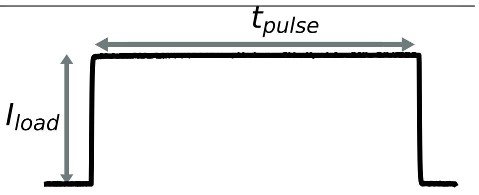
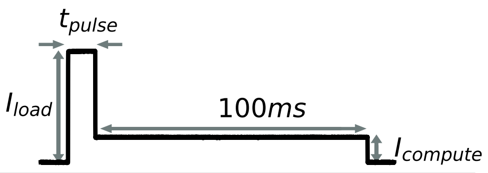
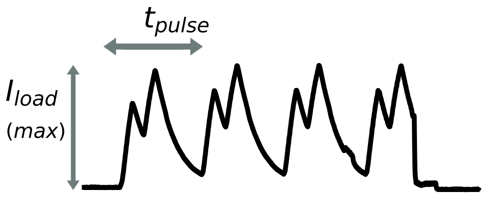
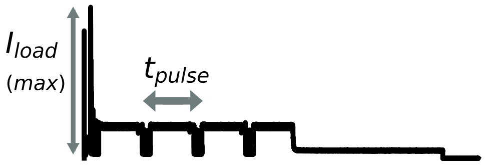
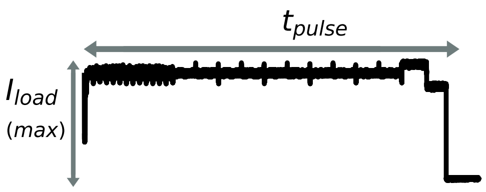
5.4 Methodology

We first evaluate Culpeo's ability to generate V^{safe} values for synthetic and real-peripheral load profiles, showing benefits by direct comparison to a voltage-as-energy baseline system. We then implement a state-of-the-art scheduler and integrate Culpeo voltage reasoning to test the end-to-end value of Culpeo in full, event-based applications. Our methodology relies on testing on real energy harvesting devices to demonstrate that Culpeo is practically useful and that accurate V^{safe} values improve application performance.

5.4.1 V^{safe} Evaluation

Our experiments use the Capybara energy-harvesting platform [57] because its power system architecture supports high-ESR supercapacitors and it is preconfigured with several sensors, an ultra-low-power MCU, and a BLE radio. We disabled Capybara's reconfigurable energy storage so that its power system closely resembles the architecture in Section 2.3, with a V^{off} of 1.6 V, a V^{high} of 2.56 V and V^{out} of 2.55 V. Unless otherwise noted, the energy buffer was a 45mF capacitor bank composed of dense supercapacitors [235]. To facilitate automated testing while validating V^{safe}, we modified Capybara to isolate the power system from the load side components by default. A test harness controls incoming power and explicitly triggers the power system to begin delivering power. In full application tests, Capybara is unmodified

Table 5.4: Loads tested. Description of different loads used in our evaluation.

Load Type	Parameters	Current Profile
Uniform	$I_{\text{load}} =$ $\{5mA, 10mA, 25mA, 50mA\}$ $t_{\text{pulse}} = \{1ms, 10ms, 100ms\}$	
Pulse	$I_{\text{load}} =$ $\{5mA, 10mA, 25mA, 50mA\}$ $t_{\text{pulse}} = \{1ms, 10ms, 100ms\}$ $I_{\text{compute}} = 1.5mA$	
Gesture Recognition	$I_{\text{load}}(max) = 25mA$ $t_{\text{pulse}} = 3.5ms$	
BLE Radio	$I_{\text{load}}(max) = 13mA$ $t_{\text{pulse}} = 17ms$	
Compute Acceleration	$I_{\text{load}} = 5mA$ $t_{\text{pulse}} = 1.1s$	

except for attaching an external capacitor bank. For all tests we simulate harvested solar energy using a 2.2V output in series with a potentiometer. A measurement harness collects time-series traces of voltage and load current [228, 257].

We used load current profiles from synthetic applications and real peripherals, shown in Table 5.4, to validate V^{safe}. Synthetic profiles are generated by toggling resistor-transistor circuits tuned to sink specific loads from V^{out} under two load shapes to explore their affect on V^{safe}: Uniform and Pulsed. The Uniform load applies a fixed current (I_{load}) for a fixed duration (t_{pulse}), representing a high-powered peripheral. The Pulsed load applies a high current pulse (I_{load} for t_{pulse}) followed by $100ms$ at $I_{\text{compute}} = 1.5mA$, representing peripheral activation followed by low-power computing. The peripheral traces were captured from the gesture-recognition sensor [22] and BLE radio [256] on Capybara as well as an external ARM Cortex-M4 [245] running a digit recognition workload [42, 154, 249].

We tested the utility of Culpeo's V^{safe} estimation by monitoring whether a software task completes without power failure when started at V^{safe}. Our test harness charges the supercapacitor bank to V^{high}, disables the charging circuit, discharges the capacitor to the V^{safe} value, and then applies a load profile. Disabling incoming power represents a worst-case scenario where the V^{safe} value must ensure that the task completes using only the stored energy. We ran the real profiles using this approach and compared the accuracy of the values produced by Culpeo-PG and Culpeo-R to two baselines, a direct energy estimate and CatNap.

We also use the harness to produce known-good V^{safe} values for the synthetic load profiles. Via a brute-force binary search, the test harness finds a profile's V^{safe} by repeatedly running the profile at different V^{safe} levels until the minimum voltage is within 5 mV of V^{off}. We validated that values below the V^{safe} produced by brute force cause failures by running multiple trials of each synthetic load profile with V^{start} above and below the known V^{safe}. Based on our analysis, estimates more than 20 mV below V^{safe} will reliably cause failures, and estimates from 20 mV below to V^{safe} will cause failures some of the time. The validated brute-force methodology allows mathematically comparing the V^{safe} calculations of Culpeo-PG, Culpeo-R-ISR and Culpeo-R μArch with CatNap. We separate the Culpeo-R implementations to tease apart the effect of an 8-bit ADC on Culpeo-R μArch versus the 12-bit precision used by Culpeo-R-ISR.

5.4.2 Application-Level Comparison

The V^{safe} tests evaluate Culpeo's accuracy, but they do not show accurate V^{safe}'s direct benefit to applications under realistic configurations. During scheduler tests, Capybara is not connected to the test harness. It charges and discharges based on the scheduling policy under test and provides constant, weak harvestable power, matched to a solar harvester [124, 193].

To understand V^{safe}'s direct benefit to applications, we integrated a Culpeo-R-ISR interface into the energy-based scheduler CatNap and tested full applications on harvested energy. We modified the CatNap implementation to support a larger capacitor bank than CatNap originally targeted, which required changing the low priority scheduling policy to account for longer recharge times. For instance, in the original CatNap, once the capacitor voltage drops below the "energy bucket" threshold, CatNap will not run low priority tasks until the buffer is fully recharged. CatNap originally tested a device with a 1 mF, low ESR capacitor bank, but when fully recharging a 45 mF capacitor bank, charging to full represents a prohibitively large quantity of energy and starves tasks. Instead, we set the low priority task threshold to 100 mV above the event bucket threshold (high priority voltage requirement). We also disabled additional features, e.g., adaptive voltage measurements, that would interfere with measuring the effect of voltage-based V^{safe} estimates, including CatNap's feasibility test. The feasibility test was replaced with a check that the current voltage is above V^{safe} before running a high priority task. We then added the Culpeo-R-ISR task profiling runtime as described in Section 5.3.3 and replaced CatNap's V^{safe} and "energy bucket" ($V^{\mathsf{safe}}_{multi}$) calculation with Culpeo-R's.

The full applications span a range of load characteristics and requirements for success. To guarantee that each application is feasible, we degraded the event frequency until the application successfully meets its requirements with V^{safe} for each task set to a safe value. We test each application by running three five minute trials.

Periodic Sensing (PS) reads 32 samples from an IMU [248] every 4.5 seconds and has a background task that reads from a photoresistor and keeps an average of the value when extra energy is available. PS uses a 15 mF energy buffer to explore Culpeo's performance with smaller buffers. An event is considered lost if the intersample deadline is not met.

Responsive Reporting (RR) triggers three high priority tasks in response to an interrupt triggered by a GPIO pin that arrives based on a Poisson distribution with λ=45 s. The first event reads from the IMU, as in PS, the second encrypts the IMU samples, and the third sends the encrypted samples over a BLE radio and performs a low-power listen for 2 seconds awaiting a response [91]. Like PS, a background task captures light levels from a photoresistor. RR must respond to interrupts within 3 seconds or the event is lost.

Noise Monitoring & Reporting (NMR) reads 256 sample from a low power microphone [4] at 12kHz every 7 seconds, while a low priority task performs an FFT on the samples in the background. Interrupts arrive with a Poisson distribution of λ=30 s, and trigger a BLE response containing the fft data followed by low-power

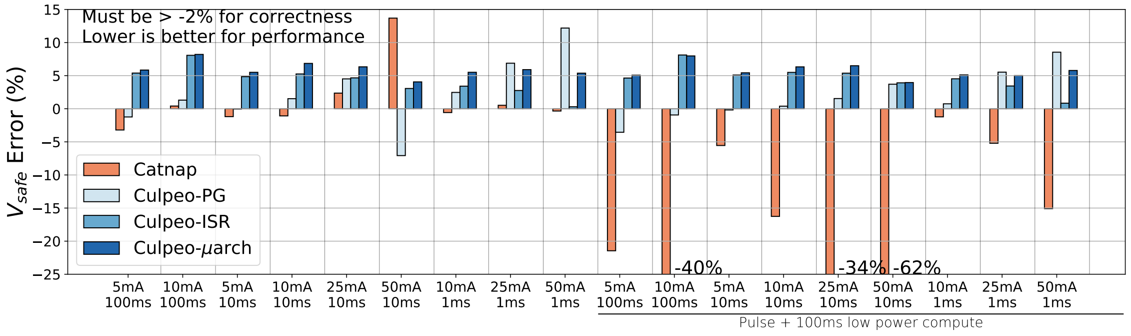


Figure 5.10: V^{safe} **Error.** The V^{safe} estimates produced by CatNap and other energy-based methods produce radically incorrect estimates. Culpeo (both static and dynamic) produce safe (> 0) and performant ($<10\%$ error) estimates.

listen that must respond within 15 seconds.

5.5 Evaluation

Our evaluation shows that Culpeo generates V^{safe} values enabling correct operation when prior systems grossly under-estimate a task's safe starting voltage. We also show that integrating Culpeo into a scheduler allows capturing events that would otherwise be missed.

5.5.1 Culpeo's V^{safes} are Accurate

Figure 5.10 shows the difference between the known-good V^{safe} value and the V^{safe} predicted by each approach for each synthetic load as a percentage of the total capacitor voltage range (2.5V-1.6V). For correctness the difference must be above -2% and greater than 0% is best. Overall, the results show that ESR-aware V^{safe} estimates are much more accurate than state-of-the-art voltage-as-energy approximations. Specifically, the results show that CatNap fails when a workload has a low current "tail" after a high current pulse. Since CatNap's V^{safe} estimate ignores ESR, as load current and ESR drop grow (from $5mA$ to $50mA$), CatNap's estimates degrade. The 50 mA, 10 ms pulse shows an important side effect of CatNap's approach– for very large ESR drops, CatNap will *overestimate* the energy required as it observes a voltage drop before rebound and treats it as consumed energy. Culpeo-PG fails in instances when the total load energy is high, such as for both of the 100ms load pulse + 100ms compute workloads , which are high energy due to pulse length, and the 50mA,10ms pulse. These failures are likely due to compounding errors in the output booster efficiency model. Compounding errors also cause Culpeo-PG's estimates to get more conservative as current increases at a given frequency. Both Culpeo-R implementations provide safe estimates for all load profiles, demonstrating the robustness of the online approach. Like Culpeo-PG, Culpeo-R-ISR's estimates are less accurate as energy increases, but its estimates are always safe. The Culpeo-R-μArch is more conservative than Culpeo-R-ISR due to its lower precision. The difference is not large, except for the 50mA,1ms pulses where, ironically, Culpeo-R-ISR's slower clock rate results in an aggressive estimate because it misses the minimum voltage.

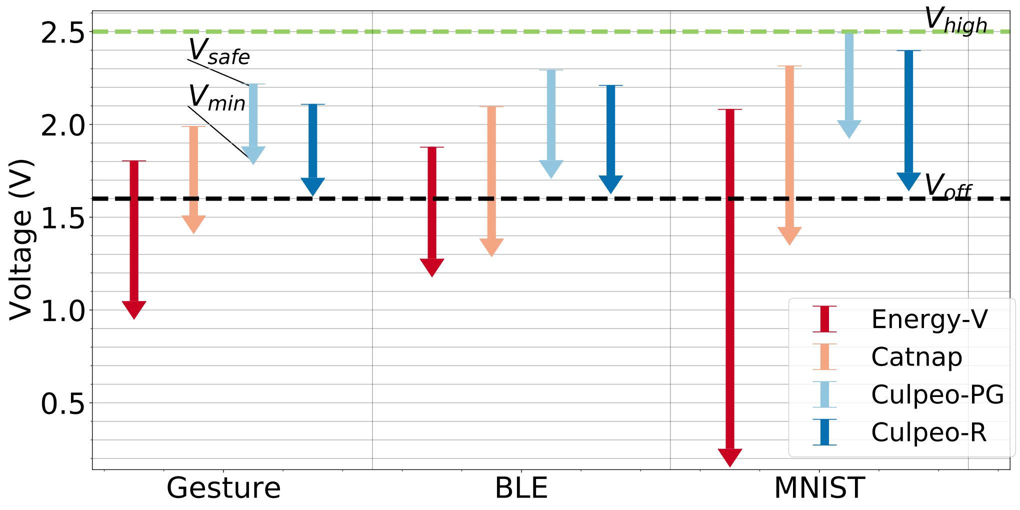


Figure 5.11: V^{safe} Accuracy on Real Peripherals. Culpeo-R and Culpeo-PG's V^{safes} (arrow tops) result in V^{min} (arrow points) above V^{off} for tests on three real peripherals. The Energy and CatNap V^{safe} estimates are unsafe.

We also show that Culpeo produces safe V^{safe} estimates for three real-world peripherals (Figure 5.11): a gesture recognition sensor, BLE, and a compute accelerator running an MNIST digit-recognition DNN. In the graph, the top of each arrow is the V^{safe} at which each systems begins the peripheral operation, and the bottom of the arrow is the minimum obsverved voltage. The closer the bottom of the arrow is to V^{off} (1.6V) without going below, the more accurate. The results show that Energy-V, an end-to-end voltage based approximation that closely tracks with direct measurements, and CatNap are not safe for realistic peripheral workloads.

The Energy-V estimates force the capacitor so low the output booster falls into a non-operational region that drags down the capacitor voltage. CatNap fares better, but all of its estimates are still below V^{off}, triggering a power-off under normal operating conditions. In contrast, both Culpeo versions perform well. Culpeo-PG provides slightly more conservative estimates than Culpeo-R as it selects a single ESR value to use for the entire operation. Culpeo-R's estimates are very accurate–they never result in a V^{min} higher than 1.7V and never fail. Overall, the data show that Culpeo correctly produces a safe voltage from which to start a task without experiencing failure, but existing systems do not.

5.5.2 V^{safe} Fixes Schedulers

We first show analytically how to correct CatNap's feasibility test with V^{safe} to ensure that tasks will not fail due to ESR drop. CatNap's feasibility test can be written as $\forall t \geq 0, e_{\mathsf{cap}}(t) > 0$. In other words, at any time, there is always energy in the capacitor *after* executing the task scheduled at time t. This test is assumed to also mean that the system will never fail to execute a task if the schedule is determined feasible, but having sufficient energy is *not* synonymous with lack of failure. CatNap's test only considers energy consumption for a task, implicitly assuming that voltage to satisfy the energy consumption is a sufficient level, i.e., $\forall t, V_t \geq V(E_t)$. Looking at the formulation for V^{safe}– $V^{\mathsf{safe}} = \sum_{i=0}^{n} V(E_i) + \sum_{i=0}^{n} penalty_i + V_{\mathsf{min}}$ – it becomes clear why this test is incorrect. If for any operation i in the task $penalty_i > 0$, then $\sum_{i=0}^{n} penalty_i > 0$ and $V_t^{\mathsf{safe}} = (\sum_{i=0}^{n} V(E_i) + \sum_{i=0}^{n} penalty_i) > V_t^{\mathsf{catnap}}$. So, the CatNap scheduler does not meet the voltage correctness constraint. Instead the feasibility test must be expanded as:

Theorem 1 *Tasks $\{\epsilon_0, ...\epsilon_n\}$ are feasible if $\forall\ t :: 0 \leq t \leq n, V_t \geq V_t^{\mathsf{safe}} \wedge e_{\mathsf{cap}}(t) > 0$, where V_t is the voltage level before executing task ϵ_t and $e_{\mathsf{cap}}(t)$ is the energy after executing.*

If a scheduler uses this feasibility test, then the voltage will not dip below the power-off threshold while running any task, *and* there will always be sufficient energy.

5.5.3 Culpeo Corrects Applications

A Culpeo enabled scheduler uses V^{safe} to eliminate the unexpected power failures that prevent CatNap from meeting application requirements. Figure 5.12 shows the percentage of captured IMU events in PS, report triggering events in RR, and both the microphone (-mic) and reporting(-BLE) events in NMR. The data demonstrate that Culpeo prevents applications from unnecessarily missing events. CatNap misses PS and NMR-mic events because of unexpected voltage drops that trigger power failure. Powering down requires the Capybara to spend time recharging that may cause further events to be missed. The missed NMR-mic events are thus actually caused by ESR drops and recharges during the BLE reporting task, not by accesses to the low power microphone. RR fails the vast majority of its responses in CatNap because the threshold level at which to run low priority work is too low *and* the V^{safe} is too low. As a result, CatNap discharges the capacitor too far when running low priority work. When an interrupt arrives, the process of sensing, encrypting and transmitting begins, but fails, and the system transmits the sensed data on the next reboot, after the deadline has passed. CatNap performs slightly better in NMR-BLE than in RR because the BLE event stands alone– the capacitor voltage is higher when starting so the misprediction in V^{safe} matters less and thus results in fewer (but still over 50%) lost events. Culpeo eliminates the vast majority of missed events sustained by CatNap. Culpeo does experience some lost events for NMR-BLE because it waits charge to V^{safe} for the radio task and does not always charge fast enough to meet the deadline.

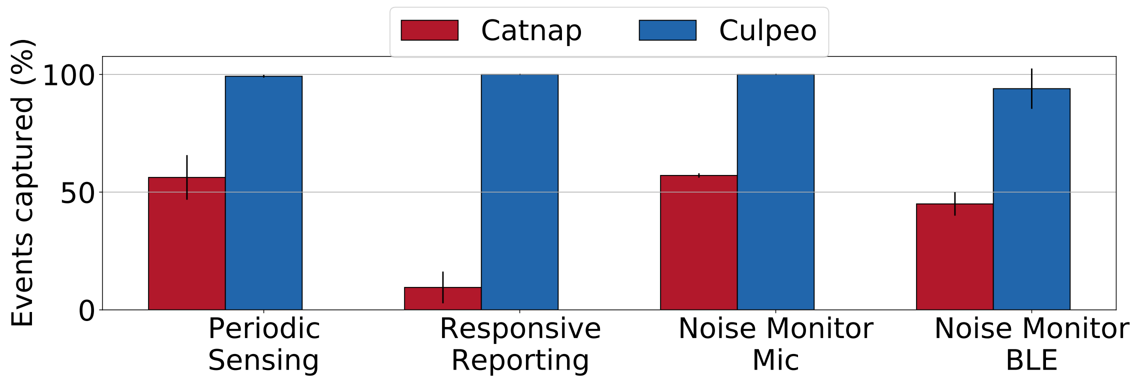


Figure 5.12: Events Captured. Culpeo's accurate V^{safe} estimates enable high event capture rates where CatNap's estimates cause it to fail.

Finally, we examine the effect of event-interarrival time on scheduler performance. Figure 5.13 shows the missed event rate for PS and RR given three sampling rates– slow (6 and 60 seconds for PS and RR respectively), achievable (4.5 and 45 seconds), and too fast (3 and 30 seconds). Running the applications at a range of interarrival times demonstrates how Culpeo and CatNap react to an energy surplus or deficit. Overall, Culpeo makes the plot make sense– once the frequency drops to an achievable level given the incoming power, Culpeo guarantees high event capture rates. CatNap, however, experiences little or inverted benefits from reducing the event frequency. This phenomenon occurs because CatNap discharges the capacitor too far performing background work. The more time between events, the further CatNap will discharge the capacitor and the more likely it will fail.

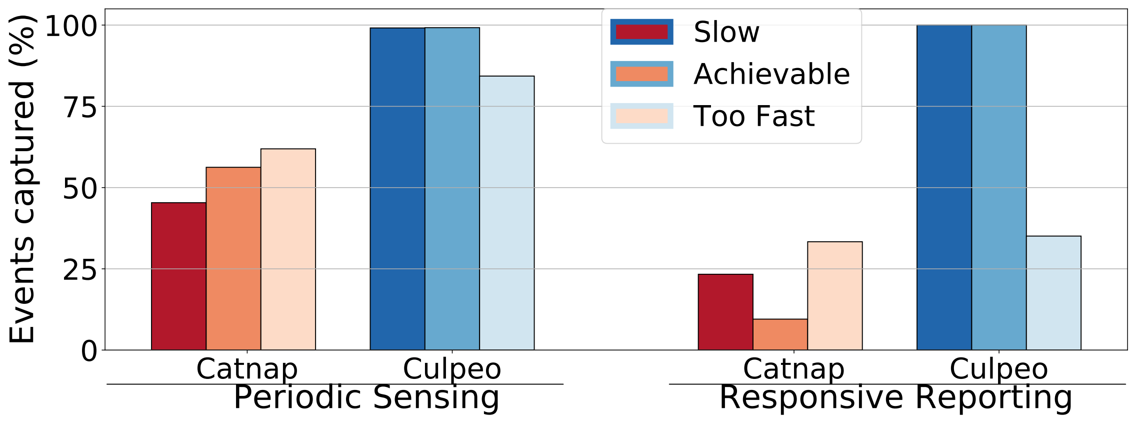


Figure 5.13: Events Captured vs. Event Frequency. Culpeo has nearly ideal event capture for achievable event rates, but CatNap discharges the capacitor too low while performing background work and inverts the expected improvements.

5.6 Related Work

Culpeo relates to work spanning a wide range of topics including intermittent systems, as detailed in Chapter 2, supercapacitor enabled sensors and energy-aware programming. Culpeo closes a gap in the literature between programming models for intermittent systems and hardware to enable higher power peripherals and compute acceleration.

Supercapacitor Enabled Embedded Systems.

Culpeo is motivated by numerous energy-harvesting platforms that rely solely on supercapacitors for energy storage. No prior work examines the effect of high ESR on applications that run on supercapacitor-only systems. Early batteryless supercapacitor motes, such as Everlast [240] and Ambimax [200] focus on charging supercapacitors efficiently and use large supercapacitors (>10F) for which ESR is not a primary concern. Additional work defined principles for building efficient power systems in energy-harvesting, supercapacitor based devices [43, 142], and are complimentary to Culpeo's efforts to improve the capability of supercapacitor based devices. More recent systems either seek out low ESR capacitors, increasing volume or cost as in the Camaroptera [193] and TA-1 [166], or engineer their applications to compensate [88]. Both the sensor node and EdbSat instantiations of the Capybara power system use compact, high ESR supercapacitors for energy storage [57], making them primary targets for Culpeo. The Capybara power system [57] uses an output booster to compensate for voltage drops due to high ESR, but the work does not describe the limitations that ESR places on how energy can be extracted from the supercapacitor. Capybara's task based programming model makes no guarantees about completion, and requires system developers to test all tasks before deployment. Several hybrid supercapacitor-battery sensing nodes including Prometheus [128], Trio [80], HypoEnergy [185] and numerous others as described in [139], use a supercapacitor to reduce the primary battery cycling. Culpeo, in contrast, targets supercapacitor only systems.

Energy-aware Programming.

Culpeo is designed to aid energy-aware programming languages and models in successful interactions with supercapacitor based systems. Energy Types [52] and ENT [46] are energy-aware type systems whose guarantees fail in the presence of ESR as neither considers the rapidly changing energy state of a batteryless system nor has constructs to easily support ESR drop. Eon associates tasks with energy levels [242] and could better evaluate available energy in batteryless systems using Culpeo. Levels permits "optional" code to run based on energy availability, estimated using a simple battery model, and would require an awares of ESR to run on a batteryless system [150]. Pixie is a WSN programming model that allocates energy to tasks within a dataflow graph via resource tickets [165]. Pixie's energy allocator and energy broker abstractions would benefit from this work's treatment of ESR as a first class concern. Additional efforts allow programmers to trade accuracy for energy via approximation [24, 115, 230] and adaptation to QoS requirements [136, 295] that all rely on a means of measuring energy consumption. While these efforts are not designed to run on energy-harvesting systems, they demonstrate the importance of accurate energy models to enable optimization at higher levels of the system stack.

5.7 Conclusion

This work is the first to identify ESR as an important factor in designing intermittent software systems. ESR-induced voltage drops break correctness assumptions of prior work, which reason about energy without considering voltage. These voltage drops are brought about by existing and emerging high power peripheral sensors and actuators from which batteryless systems derive their value, so these drops must be managed. As a remedy, we present Culpeo, a hardware/software interface that enables intermittent system designers to reason about power system effects like ESR. We present and implement two different approaches to calculate V^{safe}: Culpeo-PG, which supports compile-time reasoning and Culpeo-R for online re-evaluation. Culpeo-PG uses offline profiling of the power system and workloads to predict worst case V^{safe} values and provides insight before deployment on how a programmer can expect the power system to respond to an application. However, Culpeo-PG cannot react to changes in ESR over time. Culpeo-R uses a dynamic sampling approach to overcome this limitation and perform V^{safe} calculations online. We implement Culpeo-R-ISR on a real device using a timer interrupt and available hardware on the MCU. We also propose a new microarchitectural interface, Culpeo μArch, in the form of a memory-mapped peripheral module that eliminates the system integration challenges incurred by Culpeo-R-ISR. Culpeo μArch demonstrates that new ADC technology can reduce the power overhead of sampling the capacitor voltage by 1000x and move towards constant atomic block monitoring.

We found in our evaluation on a real energy harvesting power system that Culpeo produces safe V^{safe} values across a range of synthetic and real peripheral loads where prior estimates fail. After establishing the accuracy of Culpeo's estimates, we implemented a state-of-the-art scheduler, CatNap, and modified it to run on Capybara. We then evaluate Culpeo versus CatNap using full, event-driven applications. Our results show that that disregarding ESR causes unexpected system failures and missed events, which the Culpeo-augmented version of CatNap avoids. In this work, we showed that it is critical to understand the effect of ESR in a supercapacitor based energy harvesting system and we demonstrate systems that can capture the effect at multiple stages of the application.

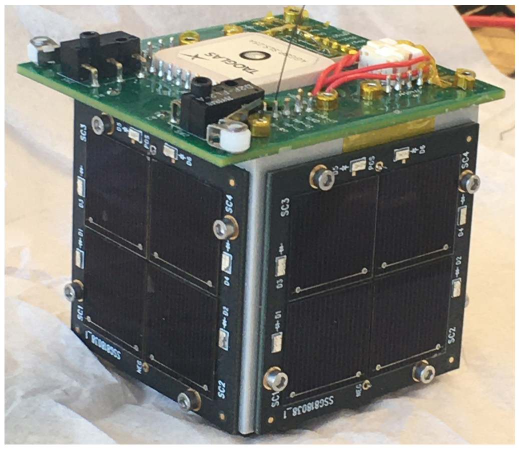
As we have shown in the last three chapters, with clearly defined programming models, applications can correctly manage shared data, power and energy state across peripherals and the MCU. However, we have only demonstrated this management on a platform that contains a single MCU, namely the MSP430 on Capybara [57]. An important question for the future of batteryless devices is what happens when systems become more complicated? How do programmers manage peripherals if more than one program is running on a device at a time? In the next chapter, we study the answers to some of these questions by building a batteryless devices with three programmable MCUs on board.

Chapter 6

Case Study: Supporting Peripherals in a Batteryless Satellite

In this chapter, we apply lessons from the preceding chapters to demonstrate how to support arbitrarily complicated peripherals in a batteryless setting. The goal is to furnish energy-harvesting capabilities and failure-aware runtime support to embedded applications that are largely agnostic to power failures. Specifically, we demonstrate battery-free operational support in the context of Tartan-Artibeus-1 (TA-1), a PocketQube satellite [215]. As shown in Figure 6.1a, TA-1 is a miniature satellite that was deployed into orbit in January 2022. The goal of the mission, albeit cut short by a launch problem, was to demonstrate the value of *orbital edge computing* [68], co-locating compute resources with on-board sensors. To support a flexible design where sensor subsystems and compute subsystems could be swapped in and out, TA-1 is composed of a stack of interchangeable subsystems (visible in Figure 6.1b). The application deployed on TA-1 makes use of all of the subsystems to capture and process telemetry data, transmit and receive messages from Earth and compute orbital position. TA-1's contribution to miniature satellites is demonstrating the value of a batteryless design for a satellite that consumes up to one Watt of power. Prior work has demonstrated batteryless nanosatellites that consume up to 100 mW, but no work has explored higher power batteryless satellites.

In building TA-1, this chapter draws on themes from across this thesis. TA-1's


(a) **TA1 Assembled.**

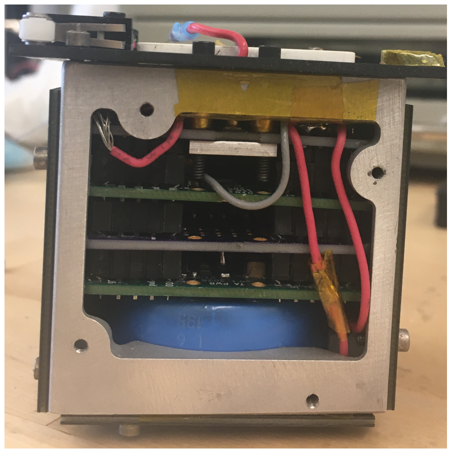

(b) **TA1 Board Stack.**

Figure 6.1: TA-1 Satellite. TA-1's modular stacking design (visible at right in a prototype) easily integrates new payloads.

power system, like Capybara described in Chapter 2, needs to maximize energy extracted from a supercapacitor, leading to a dual-booster power system for energy storage and power conditioning. The failure-aware control module adheres to lessons learned about the interrupts in task-based intermittent execution in Chapter 3 to efficiently implement ISRs to handle structured communication with peripherals. Pudu's lessons about peripheral power consumption from Chapter 4 motivate the hardware interface that peripherals expose. Finally, an understanding of ESR is essential to understand the performance of the TA-1 system as a whole. TA-1 is designed to support both high-ESR supecapacitors and high current peripherals, so lessons from Chapter 5 help to mitigate voltage drops. Building on these themes, we define the interface between failure-aware and failure-agnostic components, allowing TA-1 to support flexible workloads, off-the-shelf peripherals, and some parallelism.

TA-1 was built over the course of three years with a team of more than 10 individuals at Carnegie Mellon that coordinated with more than five external vendors. In short, TA-1 required a substantial engineering effort that is orthogonal to this thesis' research goals. We refer interested readers to the open-source design files [222, 223] and a description of the TA-1 project [67] for complete details on the TA-1 implementation. This chapter instead centers on the benefit of creating a strict separation between failure-aware and failure agnostic components in a batteryless system. To accomplish this separation, we expand the definition of a "peripheral" to include any subsystem component that will not save its state across power failures, no matter how complex its functionality. After briefly introducing batteryless nanosatellites, this chapter will discuss the power system design considerations, failure-aware application and runtime software as well as the hardware interface to failure-agnostic peripherals.

6.1 Implementing a Batteryless Nanosatellite

Nanosatellites, e.g. CubeSats [212] are a fraction of the size and cost of a traditional satellite and support exciting Earth observation applications [192]. Due to their reduced per-device cost and capabilities compared to a traditional, monolithic satellite, nanosatellites may be deployed as constellations [73] in low-Earth orbit. Nanosatellite constellations allow multiple small satellites to work together to perform applications like full Earth imaging for precision agriculture [17], disaster management [232], and others detailed in a review of the area [192]. However, the more satellites in a constellation, the less bandwidth for each satellite to communicate with a ground station on Earth [74]. It quickly becomes infeasible for constellations of nanosatellites to downlink all of their captured data back to Earth, which limits the performance of nanosatellite constellations.

Recent work proposed orbital edge computing, OEC, a strategy that moves data processing from ground stations to the nanosatellite. The goal is to increase the value of downlinked data by sending digests of high-volume sensor data, e.g. downlinking the results of a building counting application, not the images used to determine the count. Work in OEC demonstrated that equipping nanosatellites with specialized computing hardware like GPUs substantially increased the land area that a nanosatellite constellation can monitor [68]. Studies of OEC also found that increasing the energy capacity of nanosatellites does not improve total throughput for OEC. Instead, harvestable power limits the work a nanosatellite can complete on orbit. As a result, batteries do not hold a fundamental performance advantage over supercapacitors in this domain.

The energy harvesting limited nature of OEC workloads presents a compelling motivation for batteryless nanosatellites that can provide Watts of power. Since higher energy capacity does not necessarily improve the performance of OEC applications, system designers are free to consider using supercapacitors for their advantages over batteries. For instance, supercapacitors have higher power-density than batteries which allows them to deliver more power to the load within a tight volume budget [140]. Additionally, supercapacitors continue to operate at wider temperature ranges than batteries, and can eliminates the power required to heat batteries in low Earth orbit (LEO) [143, 182]. Finally, supercapacitors support up to 2000 times more cyclability over rechargeable batteries [213]. Giving batteryless nanosatellites substantially longer lifetimes than battery-powered nanosatellites. However, accelerators, sensors and actuators that consume several hundred milliwatts, typical for nanosatellites [195], will quickly drain supercapacitors and can force the device into intermittent execution.

Existing hardware and software techniques for intermittent systems minimize atomic energy requirements to guarantee that miniature batteryless devices will make forward progress, sacrificing flexibility in the process. Batteryless nanosatellites need both flexibility and resilience to power failures to meet the application requirements of inexpensive deployments to LEO. No work addresses how supercapacitor enabled systems should handle power failures that occur at the scale of minutes– a failure rate that the satellite community considers untenable, and far too infrequent for intermittent systems. We address this gap in the literature by developing a batteryless nanosatellite called TA-1.

6.1.1 TA-1 Overview

TA-1 is a flexible open-source PocketQube design that harvests energy into a supercapacitor to perform a range of operations. Figure 6.2 illustrates the different subsystems in TA-1 and their organization. The TA-1 bus, (TAB), is the blue box in the middle that separates the failure-aware components (below) from the failure-agnostic components (above). The core, failure-aware TA-1 design includes power and control subsystems that perform energy harvesting and power conditioning as well as basic flight tasks. The failure-aware components use the TAB to hide power failures from the failure-agnostic peripheral subsystems. The TAB exposes power rails, control signals and communication lines to allow additional subsystem PCBs to be stacked directly on top of the power and control systems. This simplified logical interface allowed the TA-1 team to quickly incorporate a communication module, experimental computing payload and a GNSS module [118].

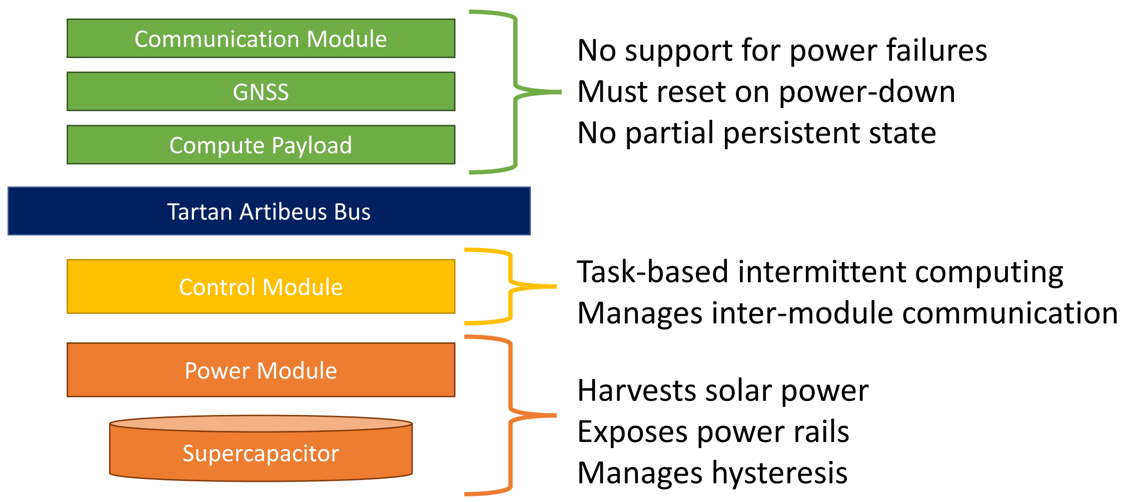


Figure 6.2: TA-1 Module Stack. TA-1 separates the failure-aware components from the failure-agnostic components using the Tartan-Artibeus Bus interface.

To explore the software and (electrical) hardware needs of TA-1, we had to adhere to the mechanical requirements of a nanosatellite specification. We targeted a PocketQube form factor because its dimensions of 5x5x5cm reduce satellite cost further than CubeSats [212, 215] while providing more power and volume than chipsats [285]. The PocketQube standard, however, is less mature than the CubeSat standard, and as a result fewer parts are commercially available for PocketQubes. To overcome this problem, we built or commissioned all of the subsystems used in TA-1, all of which are shown in Figure 6.3. In flight, the experimental compute payload (EXPT), radio subsystem (COMM) and command & control subsystem (CTRL) are stacked on top of the power system (POWR) and energy storage capacitor, as shown in Figure 6.4. The entire stack of boards is then placed inside an aluminium chassis and affixed to a base-plate according to the PocketQube standard [215].

Supporting batteryless OEC requires several changes to existing system support for batteryless devices, given that intermittent computing originally targeted μW scale systems. First, the power system must support load currents more than 10 times previous systems. Even a low power 915 MHz radio may consume over 500 mW for short bursts, and the current requirements to release the "burn" wire restraining an antenna exceed 500 mA. Second, nanosatellites are subject to more testing and verifications, in their mechanical, electrical and software design than terrestrial sensors, but they pale in comparison to traditional satellites. To streamline

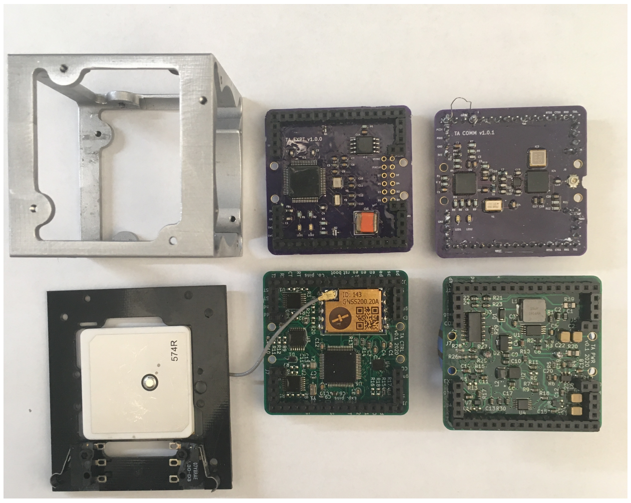


Figure 6.3: TA-1 hardware. TA-1 circuit boards, aluminium chassis and FR4 base plate.

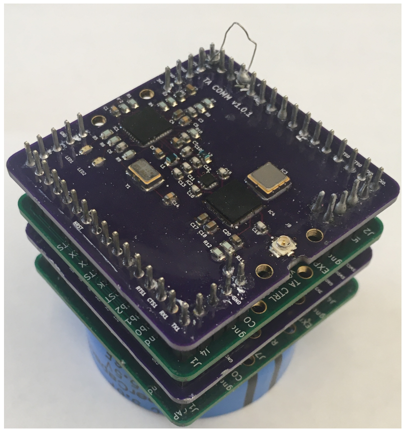


Figure 6.4: TA-1 hardware stack. Stack of TA-1 circuit boards and supercapacitor.

the integration of a batteryless runtime, it needs to be exposed to the application programmer with a minimal set of rules for writing code that will survive occasional power failures. Finally, to support the integration of existing sensors, radios, and accelerators, TA-1 should support peripherals that run programmable functions independent of the primary MCU with no knowledge of power failures. These requirements allow us to build failure-aware hardware and software components that can support failure-agnostic peripheral subsystems.

6.1.2 Peripheral Interface

For the TAB to effectively insulate peripherals from failures, each peripheral subsystem must obey a set of rules to integrate with the failure-aware components. TA-1 treats peripherals to mean any discrete subsystem that performs a specific function for an application. For instance, a radio subsystem containing a dedicated MCU is a peripheral, as is an LED. There are hardware and software design considerations when building a peripheral that integrates with TA-1.

Simple hardware changes to peripherals remove uncertainty about how each subsystem responds to power failures. First, each peripheral subsystem needs to include hardware connections that allow it to be enabled and disabled by the failure-aware MCU on the CTRL board. Inter-subsystem connections need to allow for each peripheral to be electrically isolated when it is not powered to avoid current leaks that can leave the peripheral in an undefined state. To reduce uncertainty about the state of components on peripheral subsystems, they should also minimize the size of capacitors that cannot be drained by the failure-aware components. The goal of these hardware changes is to allow the peripherals to be fully reset by the failure-aware subsystem so that power failures do not change the operation of the peripherals.

Similarly, peripheral software must follow several rules to ensure the software reboots in the correct state for it to be managed by the CTRL MCU. The first constraint is software running on the peripheral subsystems must also operate without being affected by power failures. TA-1 assumes peripheral subsystem software is not only idempotent, but that any temporary program state is cleared on power failures. For instance, programs running on peripheral subsystems should not retain partial states or write to persistent memory without indicating it to the failure-aware components. The software running on the subsystems should also wait for signals from the CTRL board to enable high power operating modes so that the total system power can be tracked. Finally, each peripheral needs to use the TAB messaging protocol to communicate between subsystems– point-to-point communication is not allowed. The goal is to prevent peripheral subsystems from considering partial state corrupted by power failures. This thesis does not define the details of the TAB protocol, but at a high level, the protocol allows the CTRL board marshal messages between programmable payloads on TA-1, namely the communication and computing subsystems.

6.2 Artibeus Power System

The TA-1 power system builds on the design of predecessors in energy-harvesting systems to support failure-agnostic peripheral subsystems. Overall, TA-1 expects a much higher harvested power than other batteryless systems because of its (relatively) large form factor. The 37 cm^2 of solar panels framing TA-1 are expected to provide an excess of 500 milliWatts, in contrast to the 90 mW expected from EdbSat's miniature solar panels [166] or 5 mW from RF harvesting [210]. The system design eschews many of the compromises other systems make to support operation under minimal power in favor of reduced complexity and broader applicability.

Figure 6.5 shows a high level schematic of the TA-1 power system. Like prior systems that optimize for extractable energy in a given volume, TA-1 uses a dual input-output booster design [57, 167, 193]. The input booster charges the capacitor as high as 5.5 V at which point the output booster is turned on by the voltage supervisor. The output booster outputs a stable 3.3 V to the VDD rail and discharges the capacitor as low as 2.0 V before the supervisor disables the output booster to avoid the input booster's cold start while the energy buffer recharges. TA-1's dual-buffer design also allows it to support supercapacitors with high-ESR. Even as voltage loss over the ESR degrades the capacitor voltage, the output voltage remains stable [57]. TA-1 is equipped with a 5.6 F supercapacitor [138] because it maximizes the energy that can be stored inside the PocketQube form factor (in a capacitor), but the design is not tied to a specific capacitor.

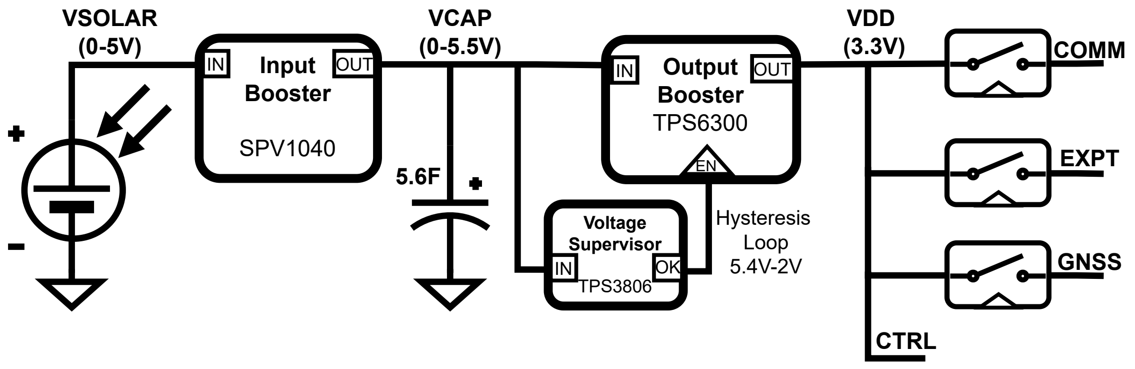


Figure 6.5: TA-1 Power System. The TA-1 power system harvests solar power into a 5.6 F supercapacitor and uses separate voltage rails to support isolated peripheral subsystems.

Despite its similarities to other architectures, TA-1 makes several choices that reduce complexity in the power system. For instance, TA-1 omits the input booster bypass path used in both EdbSat and Capybara to avoid the booster's slow cold start phase [57, 167]. Instead, TA-1 opts for an input booster with a built-in bypass diode and tolerates the slight efficiency loss because the PocketQube form factor allows for much larger solar panels than EdbSat's ChipSat form factor [285]. TA-1 also uses a unified energy buffer, in contrast to several efforts to efforts to minimize charging time for a given operation [57, 109]. Given the size of the capacitor that TA-1 supports, managing energy availability via sleeping [124, 177] instead of multiple energy buffers reduces parasitic leakage from persistent switching hardware. However, not all of

TA-1's choices reduce power system complexity.

TA-1 seeks to balance power system complexity and applications effort, so in some cases, the POWR board takes on complexity to shield applications and the failure-aware runtime on the CTRL module. First and foremost, the dual booster system supports peripherals that cannot tolerate the full voltage range experienced by the capacitor. This approach simplifies peripheral subsystem hardware development and decouples peripheral subsystems from the voltage swing experienced by the supercapacitor. The subsystem developer can simply expect a stable 3.3 V input to the subsystem until the whole system powers down. Further, the separate power rails for each peripheral subsystem minimize the effort on the part of the CTRL MCU to disable peripherals to save power. The CTRL MCU need only flip a single GPIO pin to disable a peripheral, which allows systems like Pudu to be implemented with little programming overhead.

Additional hardware associated with the power system further reduces the CTRL MCU's responsibilities. Features like the hardware defined hysteresis thresholds also reduce the flight software complexity by removing the need for a software interrupt handler that disables the output booster to avoid cold start [57, 176, 177]. The TA-1 power board includes on-board measurement hardware to capture load current, harvested power, and capacitor voltage without involving off-board subsystems. A four port op-amp buffers the voltage measurements and passes the results to a 16-bit ADC that can be accessed by any subsystem via I2C. The ADC and op-amp are both powered by VDD, so when VDD is low, both are powered off. However, the op-amp will sink current from its inputs if they are higher voltage than VDD, which would reduce the system efficiency during charging. High impedance voltage dividers between the measured voltages and the op-amp prevent extra current from draining. By making a series of careful design decisions informed by prior work, the POWR board balances complexity across TA-1.

6.3 Artibeus Control Module

The goal of TA-1 is to define a minimal set of components (hardware and software) that are aware of power failures and can maintain the execution context. We therefore program our flight control and persistency management software on a single control subsystem, CTRL, which then controls all peripheral subsystems. Hardware and software design choices allow CTRL to coordinate peripheral subsystems while maintaining the isolation requirements outlined in Section 6.1.

The CTRL module uses intentional choices in its MCU selection and peripheral communication strategy to provide power-failure awareness at little cost to the peripheral subsystems. First, the CTRL MCU is an MSP430FR5994 with 256 KB of byte-addressable non-volatile FRAM [265]. As illustrated in Chapter 3.1, byte-addressable NVM reduces the cost of persisting state compared to technologies like Flash that can only be read/written at a page-granularity. Second, the CTRL module uses level shifters [258] to isolate peripheral subsystem electrically from the CTRL module when implementing the TAB communication over UART. The isolation allows for peripheral submodules that operate at different voltages than the CTRL MCU and prevents current from leaking across the peripheral connections when the peripherals are unpowered. Finally, the CTRL module contains a low-power 9-DoF IMU [247] that provides baseline telemetry information that can be accessed by other subsystems. The CTRL module hardware provides a starting point for batteryless satellite controllers.

While the CTRL module contains application specific code, much of the underlying power-failure aware runtime and flight control software is portable. Figure 6.6 shows a diagram of how the task-based code for failure resilience is written. The code uses a `switch` statement to atomically walk through different tasks in the application, updating a persistent `task` variable after exit from the `switch` statement. If power fails in the middle of a task, execution will resume from the start of the task on the next boot. We omit sleep management, i.e. low-power wait states, because TA-1 is designed to support applications that improve performance with additional compute capacity. Since sleeping is not beneficial to the final outcome of the application, CTRL opportunistically executes tasks until power fails. In contrast to checkpointing [31, 168, 176], tasks allow the programmer to explicitly define atomic regions of code which simplifies development for well separated tasks. Developers can write calls to peripheral subsystems, place the calls inside a case statement, and assume that they will always execute from beginning to end. Since peripheral subsystems are designed to operate independently, adhering to rigid control flow at the top-level of the application is not an undue burden and is a common pattern in embedded systems [61, 193, 284].

The primary downside to tasks in an intermittent system, is the problem of write-after-reads (WAR) when an interrupted task is re-executed [168]. Effectively, the dynamic execution gets out of sync with the persistent state and leads to corrupted values. To prevent WAR bugs, the CTRL software provides a logging interface that the programmer uses to write variables involved in WAR conflicts to an undo-log [175]. Programmers may use a variety of tools to *identify* variables involved in WAR conflicts [168, 174, 252], but we recommend that they avoid using the modified code these tools produce. The performance gap between code produced by the production MSP430 GCC implementation [267] and LLVM-based compiler tooling for intermittent systems is prohibitive for real systems.

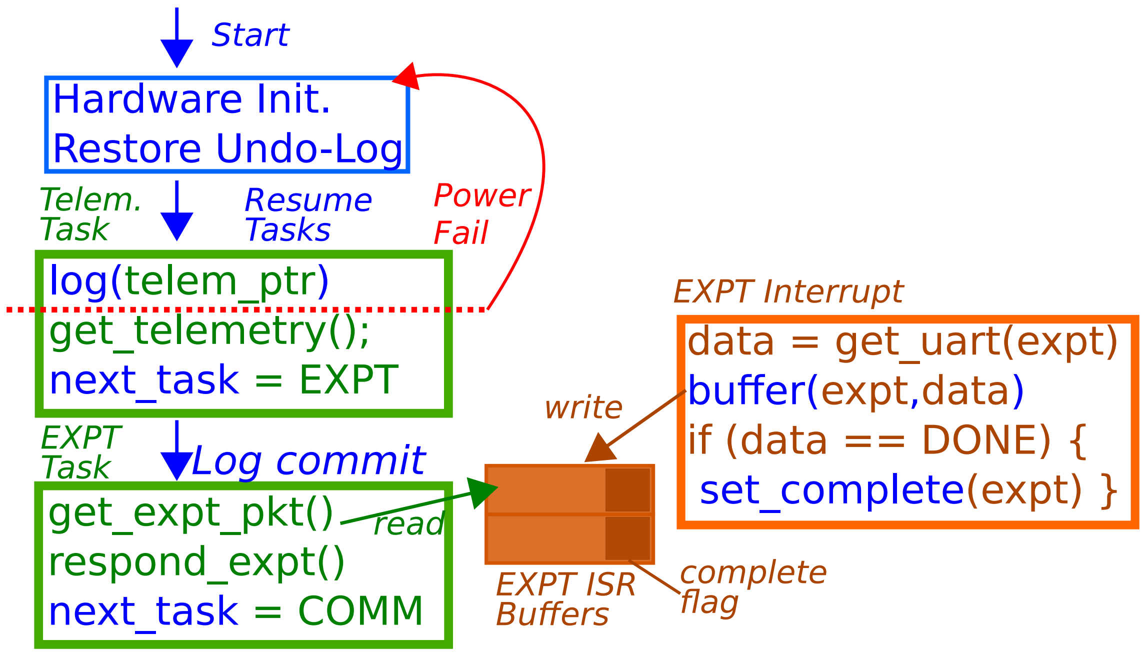


Figure 6.6: CTRL failure-aware programming.CTRL programs are written as a series of tasks (shown in green). The power-failure-aware runtime components (shown in blue) correctly restart the program after a power-failure. Interrupts from the TAB (in orange) share data with tasks through a managed buffer (center).

As shown in Figure 6.6, the CTRL software minimizes the interface between application defined tasks and peripheral-triggered interrupts. This policy is drawn from the lessons learned in Coati– like the Coati memory interface for split-phase interrupts, CTRL defines a restricted set of buffers that each interrupt may write into and tasks may read from. Unlike Coati events do not schedule tasks (i.e. event bottom-halves). Interrupts mark the buffers as "ready" when they may be read by tasks, at which point tasks may extract data from the buffers and mark them as empty. Consecutive interrupts can therefore add to the same buffer of data, but interrupts may not read from this buffer nor any other persistent data. Instead, ISRs, provided by peripheral subsystem drivers, should use only the buffer interface to share data, and use `static` variables for state that should be retained across invocations of the ISR. Following the task/interrupt interfacing rules prevents WAR violations from corrupting shared data after a power failure.

As an example, we describe the final program that shipped on the CTRL board aboard TA-1 during it's launch into LEO in January 2022. TA-1 was equipped with three peripheral subsystems: a COMM board for communication with ground stations on Earth and a EXPT payload to perform interesting compute in orbit, and a GNSS unit (with COCOM limits removed) [118]. On the first boot after launch, the CTRL code enables the nichrome "burn" wire that releases the COMM antenna from it's tied down position and then, as it will on each subsequent boot, powers all three peripheral subsystems. After booting the subsystems, the CTRL code replays the undo log from the last power failure to restore the non-volatile memory to a valid state, and begins running the next task. The first task updates the telemetry data structure every minute, reading from the 9 DoF IMU on the CTRL board, as well as the GNSS unit and the POWR ADC measurements. Next, the application reads from the interrupt buffers populated by the EXPT board to check for incoming messages and either respond to them, or re-route them to the COMM board. This task also uses the time information provided by the GNSS unit to start the EXPT payload's RTC if it is not initialized. The last task reads any incoming messages from the COMM board, and only moves on to the telemetry task if no messages are received. Otherwise the COMM task re-executes to quickly process any data being shipped to the COMM board from Earth.

6.4 Conclusion

TA-1 and the components it developed serve as a starting point for research in future batteryless nanosatellites. In particular, the interface defined by the TAB and its implementation provide a baseline for comparison in future systems. We propose

TAB as a baseline because it stands in contrast to protocols for intermittent systems and satellite buses. TAB provides flexibility *and* support for power failures.

The advantage of clearly delineating the interface between the subsystems can be illustrated by several examples of changes made to the TA-1 satellite through the course of its development. For instance, the transmission frequency of the COMM board was changed after after a year spent developing TA-1, which required changing the hardware on the COMM board, impacting its power draw. This substantial change was made with no modification to the POWR board hardware because the TAB protocol hid the changes. We also demonstrated the ability to reprogram the EXPT payload using data received over the COMM board, again, without changes to the CTRL board's software. By providing the underlying persistence and initialization guarantees, the CTRL and POWR boards are able to support a wide range of programs, even those that change during deployment.

Chapter 7

Conclusion & Future Work

This thesis explored peripheral management strategies for batteryless systems across the stack. We examined how hardware, runtime systems, and programming models can be coordinated to correctly share data, power and energy across multiple peripherals. However, there is still substantially more work to do at all levels of the stack to make batteryless devices practical for wide-scale usage. In the subsequent sections, we will examine the future engineering and research required to advance batteryless systems, and review the conclusions we can draw from this thesis.

7.1 Towards Performant Batteryless Hardware

Application performance must be considered not only as a function of how fast an application can complete, but whether it meets all of its objectives. For instance, a batteryless device that computes quickly but is powered down when new data should be collected is not performant. Hardware research for batteryless systems must examine both the hardware and its potential software interfaces to improve application performance. In this section, we describe two directions for future hardware research: reliability studies and supercapacitor-based power systems research.

7.1.1 Reliability Testing

Batteryless, energy-harvesting devices offer the potential benefit of greatly expanded lifetimes beyond battery powered devices. In contrast to devices with only primary (non-rechargeable) batteries, energy-harvesting allows the maintenance-free device lifetime to extend beyond the initial battery capacity. Further, compared to devices with secondary (rechargeable) batteries, high quality ceramic capacitors and super-capacitors can tolerate orders of magnitude more cycling (charging and discharging). The result is substantially longer device lifetimes if we assume the energy buffer is the only source of failure [69, 70, 213]. This design benefit is particularly important for larger batteryless systems, like Capybara and Tartan-Artibeus referenced in this thesis [57, 166, 193], where small rechargeable batteries fit within the allowed volume. The question, however, is whether longer capacitor cycling tolerances actually yield significantly longer lifetimes for batteryless devices.

For an energy-harvesting device, lifetime is bound by the reliability of the system components, not by energy availability. For instance, while we know that capacitors have longer cycling lifetimes than batteries, the MCU may fail before the capacitor. To declare that a batteryless device has a longer lifetime than a device with rechargeable batteries, we need to study the integrity of the key components in each system and the stresses applied by their execution models. In particular, the intermittent execution model experienced by batteryless devices represents an extreme point in the usage of a capacitor and all of the on-board SoCs. If a batteryless device uses high power peripherals, the high load current may inadvertently shorten the capacitor lifetime [149]. Further, repeated power-cycling is not the intended use case of commercially available MCUs, DC-DC converters or peripheral sensors [277].

More research is needed to understand how intermittent execution affects the lifetime of power-system and application components in batteryless devices. Prior work in power converter reliability points to large temperature swings associated with power cycling as the cause of mechanical failures (e.g. wire bonds and solder joints) [101], not the power cycling itself. Similarly, MCU failures caused by silicon degradation are predicted using a devices "on-time" rather than power cycle count [12]. It is therefore hopeful that intermittent execution does not substantially reduce IC reliability if the average power is low (i.e. unlikely to cause temperature increases). Long term studies, however, are needed to confirm this hypothesis. New techniques are needed for accelerated lifetime-testing in batteryless system so that developers can predict if their application lifetime will benefit from batteryless hardware.

7.1.2 Supercapacitor-based Systems

This thesis worked extensively with supercapacitor-based systems for two reasons. The first is that they provide a proving ground for peripherals that, at present, require more energy than a ceramic capacitor bank can reasonably provide. The second, is intermittent execution using a supercapacitor allows system designers to support higher power peripherals than would be prudent on a battery powered system. Failure *is* an option, so exceeding power neutral operation is not a failing design point [29]. Instead, intermittent execution paired with high power density from supercapacitors allow developers to explore power versus energy efficiency trade-offs in small volume envelopes. Additional research in optimal power system architectures and hardware/software interfaces is required to make the most out of supercapacitor based devices.

Efficient Hysteresis Management

Supercapacitors with low ESR and high capacity per volume allow batteryless systems to extract very large quantities of energy at high current. However, the intrinsic properties that make such a capacitor useful, are also liabilities. High capacity means recharge times are long, so the device may miss critical data while it is powered off and unresponsive. Low ESR is also indicative of high self-discharge which leaks current from a supercapacitor even when it is not in use. Improvements to power system architectures for supercapacitor based systems could allow batteryless devices to overcome such drawbacks.

For instance, TA-1 uses a 5.6 F capacitor to reduce the frequency of power failures far below that of other batteryless devices, but recharging such a large capacitor requires TA-1 to be turned off for long periods of time. This refrain is familiar– as Chapter 2 discussed, Capybara solves the problem of long recharge times by using multiple capacitor banks. At minimum, adding a second, smaller capacitor that charges quickly when coming out of eclipse would allow the CTRL MCU to to determine energy availability and set application priorities before deciding to charge the large bank. The challenges is that any resistance in the switches used to connect and disconnect capacitors to the primary bank acts like additional ESR. To support peripherals with high load currents, substantial improvements to the Capybara switch designs are needed to reduce ESR drop and allow the system to extract a useful fraction of the total energy.

Given efficient switching mechanisms, power system architects would be free to combine low and high ESR supercapacitors to get the advantages of both. High ESR supercapacitors are generally less leaky [198] and may have a higher energy density than low ESR supercapacitors. If a system programmatically switched from low ESR to high ESR capacitors throughout an application, it could modulate the expected power system behavior. For instance, switching to the high ESR supercapacitor to reduce leakage may increase the length of time a device can remain in a sleep state, as opposed to fully powered down. Strategically charging the low ESR supercapacitor could allow for extremely high current bursts at opportune points in the application. However, such a hardware system is only valuable if given tools that better map software requirements to hardware characteristics.

Expanded Power System Modeling

The problem with existing modeling for supercapacitor based, batteryless systems is that it does not capture the end-to-end behavior of the device. Culpeo successfully modeled the *discharge* portion of the capacitor voltage curve in an intermittent execution, and Pudu modeled peripheral power, but much more work is needed to model *charging*. Numerous models for supercapacitor behavior over time have been developed, but they are expensive to calculate online [9, 43, 47, 89, 133, 184, 278, 282, 283]. Prior work for intermittent systems measures capacitor voltage changes over time to estimate incoming power [177], however, ESR breaks this model. If the voltage is measured while an ESR drop is applied, the scheduler will record an artificially high incoming power if it measures after the voltage rebounds. Instead, schedulers need low-overhead models of the entire device from the input-booster through to the states of load side components (e.g. sensor operating mode) to better schedule recharges and estimate feasibility.

Language Constructs

Even before low-overhead power system models are established, language constructs based on well understood supercapacitor characteristics can improve the status quo. Indeed, disregarding capacitor ESR break existing energy-aware language constructs. Energy-Types [52] (ET) provides a type system to enable energy-aware programming.

A well-typed program preserves an invariant that program elements associated with high energy availability (e.g. battery full) may interact with elements associated with low availability (e.g. battery nearly empty), but not vice-versa. Intuitively, if there is little energy, invoking an expensive element could force the device to power down. ET's types are a suitable abstraction for energy consumption but do not accurately reflect the complications introduced by ESR drop. A program element could take little energy but have a high ESR drop. Calling this element with little energy respects the invariant but could cause the system to fail. Further, ESR drop is transient but its effects compound. If ESR drop decreases capacitor voltage, output booster efficiency will decline, making any subsequent tasks more expensive than normal. Thus, it may be unsafe for even moderately expensive code elements to run before the ESR drop rebounds. To enable sophisticated resource-aware programming on intermittent systems, languages must have abstractions for voltage-awareness.

7.2 Towards an Operating System

As batteryless devices move well past the original CRFID model and become complex distributed systems, peripheral management needs to be handled by a lightweight operating system (OS). Without an operating system, developing complex applications for batteryless devices will remain prohibitively difficult because application behavior is tightly coupled to device hardware. This thesis develops programming techniques to correctly share data, power and energy among several peripherals, but more work at the OS level is required to abstract the hardware further from the prorammer. Before developing an OS to support batteryless devices, two areas of future research that will inform the design and performance of an OS should be explored. In this section, we describe the additional research necessary to expand concurrency in batteryless devices and improve the schedulers that can be included in future operating systems.

7.2.1 Expanding Concurrency

The type of concurrent programming that is useful on a batteryless device will vary depending on resource availability. Truly miniscule systems, like CRFIDs, require programming models that minimize runtime and memory overheads while improving access to concurrent updates from peripherals. In contrast, larger systems that operate for at least seconds at a time approach traditional hybrid memory systems [51, 206, 207, 271], albeit with more frequent power failures. For these larger systems, researchers need to examine the cost of implementing a persistency model in terms of both performance and programmability. Research on expanding concurrency in batteryless devices should proceed in two different directions. First, user-studies are needed to define useful concurrency models for low-resource systems. Second, the cost of centralized persistency in large systems should be explored in full system evaluations.

User-Study for Concurrency

For highly-constrained systems, programming models that support concurrency must sacrifice flexibility for runtime overhead. For instance, Mayfly [111], Ink [284] and CatNap [177] provide varying levels of support for interrupt-driven execution, but they all use restrictive memory models to manage data shared with interrupts. Coati (presented in Chapter 3) provides a more general shared memory model than other systems but forces programmers to use split-phase interrupts. The buffering alternative to Coati, Buffi, supported a much more flexible model for serializing interrupts and transactions, but its memory and runtime overheads were prohibitively high. Overall, very small batteryless devices need to support a minimal concurrency model that allows programmers to easily process data from peripherals, and no more.

The question then arises, how do programmers actually write code that interacts with interrupts in a highly-constrained intermittent device? How would programmers write their code if system support made arbitrary accesses to shared memory simple? To answer this question, future work should carry out a literature review of the applications to stem from schedulers for small (CRFID scale) intermittent systems *and* low-power embedded systems. The survey should then assess whether useful interrupt driven paradigms in (continuously powered) embedded systems are avoided in intermittent systems because of a lack of system support. A user study could then examine Coati's effectiveness in supporting both novice and expert intermittent systems programmers in carrying out the more complicated interrupt-driven paradigms from the review. The results of the user study would then inform the design of future runtime and programming language support for concurrency on constrained, intermittent systems.

Persistency Models

In addition to supporting concurrent accesses to memory, larger scale batteryless devices can, in theory, support multiple threads of execution running in parallel. However, no work has defined a model that allows peripherals to operate in parallel on batteryless devices. Far more work is needed to develop *persistency* models to allow for parallel writes to persistent memory on batteryless devices. Tartan-Artibeus (TA-1) took steps in this direction. TA-1 centralizes the responsibility for persisting data from failure-agnostic peripheral subsystems on the control board. Failure-agnostic peripherals allow TA-1 to support off-the-shelf peripherals and reduces the need for multiple experts in failure-aware programming to build a working satellite. However, this approach wastes any partial work performed by the peripheral subsystems and introduces an artificial bottleneck in the application. Future work should examine the cost of the centralized persistency approach and explore whether existing solutions from server-class persistency models [51, 58, 78, 100, 145, 197, 206, 207, 294] could reduce the cost of logging and re-execution in batteryless systems. The end-to-end cost of a software persistency model in a batteryless system, however, will vary depending on the underlying hardware of the device.

The performance of a persistency model on a batteryless device is affected by non-volatile memory technology, energy buffer size, and communication bus design. Several works explore the cost of persisting memory for an intermittent execution running a single thread of execution in different memory layouts and technologies [55, 127, 261]. Future work needs to explore how such models perform in the presence of multiple threads persisting data as the device experiences frequent power failures, a condition that is fundamentally different from traditional persistency models. Prior work demonstrated that the energy buffer size determines the frequency of power failures [57] and provided tools to characterize intermittent execution models at design time [231]. Future work needs to go one step further and study the interaction of runtime overhead, restore costs and energy buffer size when multiple threads of execution are running in parallel. Finally, more work is needed to understand the cost of a given persistency model in the context of deploying an entire batteryless sensing and computing system. For instance, the point-to-point communication used in the TA-1 TAB hardware connections limit the number of peripherals that the CTRL board can support, but it centralizes communication. A bus protocol, e.g. CAN or multi-master I2C, would remove the fixed hardware limit, but require either additional constraints on the peripheral subsystem software or snooping by the failure-aware MCU. If peripheral subsystems are to remain failure-agnostic, inter-peripheral commands with persistent side-effects (e.g. data or operating mode) either need to be eliminated or repeated by the CTRL MCU on reboot. Future work needs to explore the interaction of such system level design trade-offs to define low-overhead persistency models specific to batteryless devices.

However, efficient, intuitive concurrency and persistency models are not sufficient to support multiprogramming on batteryless devices. System designers must also consider the effect that concurrent execution contexts have on *energy* costs. For instance, multiprogramming a batteryless device requires that one application cannot starve another by always consuming the available energy or voltage. Beyond expanding concurrency models for batteryless systems, future work needs to improve *schedulers* for batteryless systems.

Improving Schedulers

Prior work in intermittent systems begins the task of defining schedulers for batteryless devices, but none fully controls the shared resources consumed by peripherals. Systems like InK and Catnap [177, 284] implement simple priority scheduling schemes for batteryless systems, but they lack the enforcement mechanisms needed to support multiple applications. Celebi moved towards real time scheduling for preemptive events with deadlines using a persistent timer [63, 120], but does not manage resources other than time. Prior work in resource-constrained operating systems demonstrated language level constraints that allow for low-overhead memory and timing isolation, but no work has defined techniques for isolating the effects of peripherals [160]. Given the effect of peripheral power (Chapter 4) and current spikes (Chapter 5) on concurrent code, a tight coupling emerges between the software and hardware that makes batteryless devices difficult to program. More work in isolating applications is necessary to allow for programs that are portable across batteryless hardware platforms without an expert programmer in the loop.

As schedulers for intermittent systems become more advanced [177, 284], they would benefit from Pudu's insights on peripheral power consumption and Culpeo's information on energy buffer voltage. For instance, the CatNap scheduler sleeps in between tasks to recharge the energy buffer, but charging proceeds at a slower rate if peripherals are accidentally left enabled. Pudu Auto-Toggle would identify peripherals left on during sleep periods (which are typically long) and automatically disable them. Some engineering effort would be required to integrate Pudu Auto-Toggle's clock with CatNap's event timer, but this task is manageable, as described in Section 4.4.3. Similarly, INK would benefit from Pudu-Static's analysis to confirm that separate tasks do not affect each other by leaving peripherals enabled unintentionally. Future work on operating systems needs to begin by implementing the techniques proposed in this thesis for fully capturing the state of peripheral devices and their effects on the entire system.

Simply incorporating the power and peripheral management strategies proposed in this thesis, however, is not sufficient for multiprogrammed workloads that may be competing for a batteryless device's limited resources. More work on equitable scheduling is necessary to map hardware characteristics up to the level of the programming model. For instance, given TA-1's deployment in LEO, its large energy buffer and the fact that it supports some reprogramming, TA-1 is a target for multiprogramming. Multiprogramming TA-1 could allow scientists from different institutions to easily deploy and test orbital edge computing workloads at once. However, equitable scheduling in the presence of high-ESR is undefined; any workloads that cause a voltage drop over the TA-1 energy buffer could impact others. Using the Culpeo-R strategy from Chapter 5, a scheduler could establish the individual voltage/energy requirements of an application online, but Culpeo does not define how to compose applications running at the same time. Future work needs to build on the lessons from Chapters 5 and 4 to define fair allocation strategies that allow multiple applications to tolerate a batteryless device's energy and power limitations.

7.3 Towards Wide-Scale System Integration

While recent work in batteryless devices is quite promising, batteryless device usage is far from common. Beyond the hardware and operating system limitations described above, several barriers exist to widespread batteryless device usage that can be overcome with a combination of engineering and research effort. First, programming a batteryless system, as we describe in this thesis, is prohibitively difficult. Any number of pitfalls surrounding memory consistency [168, 224, 252], atomicity violations [176, 253] and peripheral power consumption (Chapters 4 and 5) can occur even when using a programming model intended for intermittent execution. Without guarantees that a batteryless device will operate within a well-defined execution model, it is unrealistic to expect that they will see widespread adoption. Second, batteryless, energy-harvesting devices are designed to act as the “leaf” nodes in an edge-computing network. These nodes capture information from the environments and ultimately relay it to a continuously powered device. However, the vast majority of the batteryless device research has focused on getting a single device to behave as expected. To overcome these barriers, future engineering effort should be devoted to building a unified test bed for batteryless systems and future research effort should define how batteryless devices operate in sensor networks.

Community-Supported Debugging Platforms

Much of the work defining intermittent execution and batteryless devices deals with managing the software and hardware bugs that result in unintuitive differences between batteryless and continuously powered systems. Research papers in this area often define programming models [111, 148, 168, 176, 224, 252] or compile/runtime checkers [178, 252] to prevent and remove bugs, but these systems tend to be designed around single categories of bugs. As a result, the research stays siloed and subtle interactions between runtime systems, like those exposed in Chapter 4, are missed. The engineering challenge, then, is to build community support for a unified debugging platform that researchers can easily extend.

A unified debugging platform needs to incorporate both software and hardware tools, a feature that distinguishes this future work from existing approaches. For instance, testbeds like `ScEpTIC` have started to incorporate multiple software-level debugging strategies for intermittent execution in one publicly available code base [16]. `ScEpTIC` tests at compile time for bugs due to re-execution [168] and multiple forms of input/output (IO) bugs [252]. `ScEpTIC` has no knowledge of a device’s hardware, but incorporating work from the broader community could add hardware-awareness at compile time. For instance, given `ScEpTIC`’s LLVM implementation, incorporating

Pudu-Static's analysis should require limited engineering effort. Beyond compile-time tools to reveal bugs that can be identified strictly using a program's code, a unified debugging tool must include a hardware harness. Several works define and build hardware harnesses that check for non-termination due to expensive atomic blocks [56, 175, 176, 288]. Others are more general, supporting hardware debugging [54] or environmental simulation [93, 291]. Again, the problem is that these systems do not build on each other in a substantive way and they do not pull in developments from research on identifying bugs at compile-time. For instance, incorporating Pudu's knowledge of peripheral state would make non-termination checkers faster and more robust by defining the peripheral typestates that need to be tested for each atomic block. Overall, future work needs to define an extensible hardware and software platform for testing and debugging to reduce the verification burden for batteryless systems.

Networking with Batteryless Devices

To integrate batteryless devices into a broader network, system developers must overcome three challenges that have not been addressed simultaneously in prior work: sensor coverage, workload partitioning and wireless communication. To incorporate batteryless devices as part of a larger network of devices, these three challenges must be considered together, because the final application running on the network will be affected by them all at once. For instance, an application may run on a hierarchy of continuously powered, battery-powered and batteryless nodes where sensing responsibilities are fixed to their physical location, but computational loads can be moved [214]. Prior work has addressed some of the challenges such a system experiences when all leaf nodes are battery powered, but intermittent execution changes the guarantees the network can make to the applications. First, sensor coverage from batteryless nodes is not guaranteed– if harvestable energy is scarce, the batteryless sensors will not report consistently [177]. Second, workload partitioning on energy-harvesting devices differs from continuously powered devices. Changes in harvestable power dynamically change the benefit of local versus remote compute [72]. Finally, establishing wireless connections between batteryless devices is non-trivial due to the sporadic nature of intermittent execution [94]. Prior work demonstrated techniques for handling these three challenges individually, but future research should focus on developing each technique within the context of the device's role in a greater system.

First, prior work defines the impact of sampling coverage and benefit of on-device compute primarily for a single batteryless sensor. Systems like Capybara modify the energy buffer size to improve coverage [57], Catnap [177] and TICS [148] use timing to meet sampling deadlines, and Ambient Batteries [125] define a slow, predictable coverage rate. In contrast, CIS demonstrates a technique to improve the sensing coverage of a network of batteryless devices [179]. However, future work should explore whether applications running on the network can tolerate gaps in sensor coverage. If applications can reconstruct missing data, e.g. through interpolation, the cost (e.g. power, conservative scheduling) of consistent sensing on the batteryless device may be reduced. Similarly, intensive on-device processing improves the end-to-end performance of individual batteryless nodes if communication is expensive [72, 98]. However, no work has explored workload partitioning for batteryless devices if communication is inexpensive [164] or varies over time. Future work should examine the benefit of on-device processing in batteryless systems given topologies that reflect use cases like instrumented "smart"-buildings [45]. Wireless communication energy costs will have a substantial impact on sensing and computing decisions in future systems.

Prior work demonstrated protocols for efficient wireless communication to a single device [255] or between two batteryless devices [63] and simulated networks of batteryless devices [169]. Recent work has made exciting strides in communicating between devices executing intermittently [94, 95, 238] and re-implementing existing protocols for intermittent execution [65]. However, no work has defined how batteryless devices should communicate in a heterogeneous network. Future work needs to develop strategies that consider a mix of battery powered and batteryless nodes, as well as a hierarchy of batteryless nodes (e.g. TA-1 versus a CRFID). The resulting communication protocols should leverage batteryless devices to capture and process critical data while maintaining high availability and lifetimes for the battery-powered devices in the network.

The overall question for future work is how to leverage the last 10 years of development in batteryless devices to build large scale networks that judiciously include batteryless devices. The value in batteryless devices in such a network will be their ability to access peripherals to capture, process and transfer information from specific locations. The tools in this thesis for managing peripherals and their resulting power consumption assist in the development of such impactful batteryless applications.

7.4 Recurring Themes

Moving away from battery powered devices allows embedded systems to reach new targets for size, weight, cost, temperature range and environmental constraints [32, 143]. A major challenge for the batteryless devices that take on these emerging applications will be supporting a diverse set of peripheral sensors and actuators so that these devices can continue to gather critical data. A large set of explicit and implicit state shared between peripherals and the MCU is cleared on every reboot from the peripheral's perspective. As we show in this thesis, managing shared state across power failures is error prone. Through programming models implemented at compile-time and run-time, this thesis reduces the programmer burden of incorporating peripherals into batteryless applications by managing concurrent peripheral state changes and power.

Shared State Management for Peripherals

To effectively manage the shared data and power state that peripherals change concurrently with the MCU, the space of what is allowed to change needs to be limited. Chapter 3 explored how to strategically limit the shared data between peripheral triggered ISRs and the MCU. By splitting events into top and bottom halves, Coati removed concurrency control bugs that cause prior approaches to fail, and did so with minimal overhead in both programmer effort and runtime. Chapter 4 handled changes to peripheral operating power that ultimately affect the performance of all code running on the batteryless device. Again, bugs arise because too many state changes are possible. If a peripheral may be at multiple operating power levels at the same point in the code, there is no guarantee that the programmer has tested both. Pudu-Static introduces compile-time analysis feedback to work with the programmer to limit the space of operating modes at a given point in the program to those that the programmer actually intended. Programmers use Pudu-Static's reports of peripheral state changes (which point to specific lines of code in the application) to remove unintended changes or affirm that multiple operating modes have been confirmed. We demonstrate the importance of limiting the interface between peripherals and failure-aware control logic in Tartan-Artibeus. The restricted communication allows the CTRL module to manage failure-agnostic peripherals and ensure predictable application behavior. Overall, limiting the changes to shared state before the program runs simplifies peripheral management with little runtime overhead, which is essential for highly constrained batteryless devices. However, viewing the shared state with peripherals strictly as data that can reside in the MCU's persistent memory is misleading. Peripheral power changes affect the energy buffer status for the entire device.

Power Management

The challenge with managing peripheral power is that its relationship to energy and energy buffer voltage are based on time. The time a peripheral is on determines the energy it consumes, and the time between when a peripheral is powered off and when a task ends determines the measured state of the energy buffer. Ultimately, due to input-dependent control flow decisions, statically predicting time, e.g. between peripheral accesses, in a batteryless device is inaccurate, highly limiting to the programming model, or both. However, dynamic approaches with well defined interfaces that push computational complexity to compile time and use the results at runtime can capture time with minimal overhead.

In Chapter 4, Pudu-Dynamic relies on pre-computed break-even times to determine when toggling a peripheral will be beneficial. At runtime, Pudu-Dynamic manages the toggling table to capture dynamic operation, but ultimately the function to determine whether or not to toggle is just a single look up and comparison. Pudu-Dynamic can take advantage of timing hardware to resolve ambiguities about the time between peripheral accesses and improve application performance with little programmer involvement. Similarly, Chapter 5 touches on the tradeoffs of dynamically versus statically estimating safe starting voltages by comparing Culpeo-PG and Culpeo-R. Culpeo-R compensates for changes in the energy buffer characteristics and application timing changes by estimating V^{safe} at runtime. By dynamically reading from the ADC and statically calculating constants for the V^{safe} equations, Culpeo-R balances the on-device compute with the accuracy of in situ power system measurements. The need to dynamically react to changes in peripheral power informed the design of TA-1. Peripheral subsystems' power is controlled by the failure aware MCU via single GPIO triggered switches. The central MCU does not need to rely on pre-profiling to fully characterize the subsystem behavior– the hardware exists to allow the MCU to monitor the energy buffer level and quickly disable misbehaving subsystems. The case study in building TA-1 justifies this thesis' peripheral-first approach to system support for batteryless devices.

Bibliography

[1] Msp430f5529 dma hangs in uart rx. https://e2e.ti.com/support/microcontrollers/msp-low-power-microcontrollers-group/msp430/f/msp-low-power-microcontroller-forum/310488/msp430f5529-dma-hangs-in-uart-rx. Accessed: 2022-04-20. Cited on page 109.

[2] Tps61200 0.3-v input voltage, adjustable ouput voltage boost converter with 1.3-a switches, 3-mm x 3-mm qfn. https://www.ti.com/product/TPS63000. Accessed: 2022-04-21. Cited on page 104.

[3] Tps61200 0.3-v input voltage, adjustable ouput voltage boost converter with 1.3-a switches, 3-mm x 3-mm qfn. https://www.ti.com/product/TPS61200. Accessed: 2022-04-21. Cited on page 104.

[4] Spu0414hr5h-sb amplified sisonictm microphone. https://media.digikey.com/pdf/Data%20Sheets/Knowles%20Acoustics%20PDFs/SPU0414HR5H-SB.pdf, 2012. Accessed: 2022-04-21. Cited on page 114.

[5] B. Lucia A. Colin, E. Ruppel. Capybara energy-harvesting platform. https://github.com/CMUAbstract/capybara/blob/master/Project%20Outputs%20for%20Capybara/CapybaraSchematic.PDF, 2017. Accessed: 2022-05-14. Cited on page 73.

[6] ABSTRACT Research Lab. Tartan-artibeus-1. https://abstract.ece.cmu.edu/ta-1/index.html, 2020. Accessed: 2022-06-06. Cited on page 8.

[7] Joshua Adkins, Bradford Campbell, Branden Ghena, Neal Jackson, Pat Pannuto, and Prabal Dutta. The signpost network: Demo abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 320–321, New York, NY, USA, 2016. ACM. Cited on page

1.

[8] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 368–381, New York, NY, USA, 2020. Association for Computing Machinery. Cited on pages 2 and 56.

[9] Jun Ick Ahn, Daeyong Kim, Rhan Ha, and Hojung Cha. State-of-charge estimation of supercapacitors in transiently-powered sensor nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. Cited on pages 103 and 140.

[10] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles, techniques, and tools second edition, 2007. Cited on page 68.

[11] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022. ACM, 2009. Cited on page 89.

[12] Allan Webber. Calculating Useful Lifetimes of Embedded Processors. https://www.ti.com/lit/an/sprabx4b/sprabx4b.pdf?ts=1653251800432, 2020. Accessed: 2022-05-23. Cited on page 138.

[13] Mohammad Alshboul, James Tuck, and Yan Solihin. Lazy persistency: a high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018. Cited on page 49.

[14] Analog Devices. 3-axis, ±2 g/±4 g/±8 g/±16 g digital accelerometer. https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf, 2015. Accessed: 2022-11-05. Cited on page 73.

[15] C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson, and Sean Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages

316–327. IEEE, 2005. Cited on page 49.

[16] Andrea Maioli. ScEpTIC. https://neslabpolimi.bitbucket.io/ScEpTIC, 2020. Cited on page 146.

[17] Bruno Aragon, Rasmus Houborg, Kevin Tu, Joshua B. Fisher, and Matthew McCabe. Cubesats enable high spatiotemporal retrievals of crop-water use for precision agriculture. *Remote Sensing*, 10(12), 2018. Cited on page 125.

[18] Arduino. Arduino Uno Rev3. https://store.arduino.cc/usa/arduino-uno-rev3, 2018. Accessed: 2018-05-03. Cited on page 43.

[19] Arduino. Arduino forum. https://forum.arduino.cc/search?q=%22peripheral%22%20%22bug%22, 2022. Accessed: 2022-02-15. Cited on page 60.

[20] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, 2000. Cited on page 89.

[21] Tim Ashworth. Characterizing the esr of a supercapacitor with the mfia. https://www.zhinst.com/americas/en/blogs/using-mfia-impedance-analyzer-characterize-esr-super-capacitor, 2016. Accessed: 2021-11-11. Cited on page 96.

[22] Avago Technologies. Apds-9960 digital proximity, ambient light, rgb and gesture sensor– data sheet. https://cdn.sparkfun.com/assets/learn_tutorials/3/2/1/Avago-APDS-9960-datasheet.pdf, 2013. Accessed: 2020-03-06. Cited on pages 56, 73, 80, 92, and 113.

[23] AVX. BestCap® Ultra-low ESR High Power Pulse Supercapacitors. http://catalogs.avx.com/BestCap.pdf, 2019. Cited on pages 16, 96, and 103.

[24] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010. Cited on page 121.

[25] Sara S. Baghsorkhi and Christos Margiolas. Automating efficient variable-grained resiliency for low-power IoT systems. In *Proc. CGO*, Vienna, Austria, February 24–28, 2018. ACM. Cited on page 13.

[26] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, 2019. Cited on page 60.

[27] Abu Bakar, Alexander G Ross, Kasim Sinan Yildirim, and Josiah Hester. Rehash: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–42, 2021. Cited on page 77.

[28] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, jul 2011. Cited on page 60.

[29] Domenico Balsamo, Benjamin J. Fletcher, Alex S. Weddell, Giorgos Karatziolas, Bashir M. Al-Hashimi, and Geoff V. Merrett. Momentum: Power-neutral performance scaling with intrinsic mppt for energy harvesting computing systems. *ACM Trans. Embed. Comput. Syst.*, 17(6), January 2019. Cited on page 139.

[30] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016. Cited on pages 14 and 26.

[31] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1):15–18, 2015. Cited on pages 14, 26, 67, and 132.

[32] Todd M Bandhauer, Srinivas Garimella, and Thomas F Fuller. A critical review of thermal issues in lithium-ion batteries. *Journal of the Electrochemical Society*, 158(3):R1, 2011. Cited on page 149.

[33] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. Peripheral state persistence for transiently-powered systems. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6, June 2017. Cited on pages xx, 3, 4, 16, 17, 53, 75, and 82.

[34] Naveed Anwar Bhatti and Luca Mottola. HarvOS: efficient code instrumentation for transiently-powered embedded sensing. pages 209–219. ACM Press, 2017. Cited on pages 13, 91, and 92.

[35] Kevin Bierhoff and Jonathan Aldrich. Plural: checking protocol compliance under aliasing. In *Companion of the 30th international conference on Software engineering*, pages 971–972. ACM, 2008. Cited on page 89.

[36] B&K Precision. 8600 Series - Programmable DC Electronic Loads. https://www.bkprecision.com/products/dc-electronic-loads/8600-150-w-programmable-dc-electronic-load.html. Cited on page 104.

[37] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, June 2007. Cited on pages 24 and 49.

[38] Colin Blundell, Arun Raghavan, and Milo MK Martin. Retcon: transactional repair without replay. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 258–269. ACM, 2010. Cited on page 49.

[39] Jacob Borgeson. Ultra-low-power pioneers: Ti slashes total mcu power by 50 percent with new “wolverine” mcu platform. https://www.ti.com/lit/wp/slay019a/slay019a.pdf, 2012. Accessed: 2022-04-21. Cited on page 111.

[40] Bosch Sensortec. Bmg250 low noise, low power triaxial gyroscope. https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmg250-ds000.pdf, 2021. Accessed: 2022-11-05. Cited on page 73.

[41] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 55–67, 2019. Cited on pages xx, 3, 4, 16, 17, 53, 66, 73, 74, 75, and 82.

[42] Jason Brownlee. Handwritten digit recognition using convolutional neural networks in python with keras, 2016. Cited on page 113.

[43] Davide Brunelli, Clemens Moser, Lothar Thiele, and Luca Benini. Design of a solar-harvesting circuit for batteryless embedded systems. *IEEE Transactions*

on Circuits and Systems I: Regular Papers, 56(11):2519–2528, 2009. Cited on pages 120 and 140.

[44] Michael Buettner, Ben Greenstein, and David Wetherall. Dewdrop: an energy-aware runtime for computational rfid. In *Proc. USENIX NSDI*, pages 197–210, 2011. Cited on pages 2 and 23.

[45] H. Burak Gunay, Weiming Shen, and Guy Newsham. Data analytics to improve building performance: A critical review. *Automation in Construction*, 97:96–109, 2019. Cited on page 148.

[46] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. pages 217–232. ACM Press, 2017. Cited on pages 62 and 120.

[47] Ruizhi Chai, Ying Zhang, Geng Sun, and Hongsheng Li. Self-aware power management for maintaining event detection probability of supercapacitor-powered cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.*, 4(4), July 2020. Cited on page 140.

[48] Wei-Ming Chen, Tai-Sheng Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. Value-based task scheduling for nonvolatile processor-based embedded devices. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 247–256. IEEE, 2016. Cited on page 97.

[49] Maryline Chetto. Optimal scheduling for real-time jobs in energy harvesting computing systems. *IEEE Transactions on Emerging Topics in Computing*, (1):1–1, 2014. Cited on page 97.

[50] Holly Chiang, Hudson Ayers, Daniel Giffin, Amit Levy, and Philip Levis. Power clocks: Dynamic multi-clock management for embedded systems. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks*, EWSN '21, 2021. Cited on page 88.

[51] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011. Cited on pages 49, 142, and 143.

[52] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM,

2012. Cited on pages 62, 120, and 140.

[53] Alexei Colin. Space data collection. `https://github.com/CMUAbstract/app-space-data-chain`, 2018. Error Loc.: main.c: line 287. Cited on page 79.

[54] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *ACM SIGPLAN Notices*, 51(4):577–589, 2016. Cited on pages 12, 62, and 147.

[55] Alexei Colin and Brandon 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*, pages 514–530. ACM, 2016. Cited on pages 3, 4, 13, 24, 26, 43, 54, 61, 74, 77, 78, 79, 91, and 143.

[56] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 116–127, 2018. Cited on pages 12, 13, 91, and 147.

[57] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 767–781, New York, NY, USA, 2018. ACM. Cited on pages 3, 4, 12, 13, 14, 15, 18, 23, 25, 27, 42, 43, 53, 56, 73, 74, 75, 77, 79, 80, 92, 96, 111, 120, 122, 130, 131, 138, 143, and 148.

[58] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009. Cited on page 143.

[59] Peter Corke, Philip Valencia, Pavan Sikka, Tim Wark, and Les Overs. Long-duration solar-powered wireless sensor networks. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 33–37, 2007. Cited on page

1.

[60] Alexander Curtiss and Blaine Rothrock. Facebit firmware. https://github.com/ka-moamoa/facebit-firmware/blob/master/src/FaceBitState.cpp, 2022. Error Loc.: FaceBitState.cpp : lines 145,94. Cited on page 79.

[61] Alexander Curtiss, Blaine Rothrock, Abu Bakar, Nivedita Arora, Jason Huang, Zachary Englhardt, Aaron-Patrick Empedrado, Chixiang Wang, Saad Ahmed, Yang Zhang, Nabil Alshurafa, and Josiah Hester. Facebit: Smart face masks platform. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(4), dec 2022. Cited on pages 3, 77, 79, and 132.

[62] Canan Dagdeviren, Byung Duk Yang, Yewang Su, Phat L Tran, Pauline Joe, Eric Anderson, Jing Xia, Vijay Doraiswamy, Behrooz Dehdashti, Xue Feng, et al. Conformal piezoelectric energy harvesting and storage from motions of the heart, lung, and diaphragm. *Proceedings of the National Academy of Sciences*, 111(5):1927–1932, 2014. Cited on page 2.

[63] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–67, 2020. Cited on pages 4, 14, 15, 70, 73, 77, 144, and 148.

[64] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), September 2020. Cited on pages 2, 13, 14, 77, and 92.

[65] Jasper de Winkel, Haozhe Tang, and Przemysław Pawełczak. Intermittently-powered bluetooth that works. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 287–301, New York, NY, USA, 2022. Association for Computing Machinery. Cited on page 148.

[66] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004. Cited on page 89.

[67] Brad Denby, Emily Ruppel, Vaibhav Singh, Shize Che, Chad Taylor, Fayyaz Zaidi, Swarun Kumar, Zac Manchester, and Brandon Lucia. Tartan artibeus:

A batteryless, computational satellite research platform. In *Proceedings of the AIAA/USU Conference on Small Satellites Recent Launches - Research & Academia*, volume SSC22-WKII-08, 2022. Cited on page 124.

[68] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Architectural Support for Programming Languages and Operating Systems*, 2020. Cited on pages 123 and 125.

[69] Eric DeRose, Bob Knopsnyder, and Bharat Rawal. Reliability of supercapacitors: Paper 1 unique performance at 85°c & self-balancing. https://www.kyocera-avx.com/docs/techinfo/whitepapers/AVX-Whitepaper-Reliability-of-SuperCapacitors-Paper1.pdf. Accessed: 2022-06-08. Cited on page 138.

[70] Eric DeRose, Bob Knopsnyder, and Bharat Rawal. Reliability of supercapacitors: Paper 2 long-term reliability test data. https://www.kyocera-avx.com/docs/techinfo/whitepapers/AVX-Whitepaper-Reliability-of-SuperCapacitors-Paper2.pdf. Accessed: 2022-06-08. Cited on page 138.

[71] Harsh Desai and Brandon Lucia. Power-aware heterogeneous architecture scaling for energy-harvesting computers. *IEEE Computer Architecture Letters*, 2020. Cited on pages 54, 76, and 85.

[72] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. Camaroptera: A long-range image sensor with local inference for remote sensing applications. *ACM Trans. Embed. Comput. Syst.*, jan 2022. Just Accepted. Cited on pages 15, 76, 79, 147, and 148.

[73] Kiruthika Devaraj, Ryan Kingsbury, Matt Ligon, Joseph Breu, Vivek Vittaldev, Bryan Klofas, Patrick Yeon, and Kyle Colton. Dove high speed downlink system. In *Proc. AIAA/USU Conf. Small Satellites*, 2017. Cited on page 125.

[74] Kiruthika Devaraj, Ryan Kingsbury, Matt Ligon, Joseph Breu, Vivek Vittaldev, Bryan Klofas, Patrick Yeon, and Kyle Colton. Dove high speed downlink system. 2017. Cited on page 125.

[75] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding

paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48, April 2017. Cited on page 88.

[76] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968. Cited on page 48.

[77] Pedro C Diniz and Martin C Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 71–84, 1997. Cited on page 89.

[78] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014. Cited on page 143.

[79] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *Proc. First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, November 2004. Cited on page 48.

[80] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *2006 5th International Conference on Information Processing in Sensor Networks*, pages 407–415, 2006. Cited on page 120.

[81] Eaton. Application Guidelines. https://www.eaton.com/content/dam/eaton/products/electronic-components/resources/technical/eaton-supercapacitor-application-guidelines.pdf, 2017. Accessed August 12, 2021. Cited on pages 96, 103, and 105.

[82] Maged ElAnsary, Jianxiong Xu, José Sales Filho, Gairik Dutta, Liam Long, Aly Shoukry, Camilo Tejeiro, Chenxi Tang, Enver Kilinc, Jaimin Joshi, et al. 28.8 multi-modal peripheral nerve active probe and microstimulator with on-chip dual-coil power/data transmission and 64 2 nd-order opamp-less $\delta\sigma$ adcs. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 400–402. IEEE, 2021. Cited on page 110.

[83] Embedded Rust Resources Team. The Embedded Rust Book. https://rust-embedded.github.io/book/, 2018. Accessed: 2018-11-05. Cited on

page 89.

[84] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: an energy-aware resource-centric rtos for sensor networks. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–265, 2005. Cited on page 1.

[85] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008. Cited on page 89.

[86] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 323–338, USA, 2008. USENIX Association. Cited on page 88.

[87] ESP8266 Community Forum. Apds-9960 rgb and gesture sensor problems. https://www.esp8266.com/viewtopic.php?f=32&t=12504, 2016. Accessed: 2021-04-16. Cited on pages 60, 61, and 80.

[88] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh Gupta. Pible: Battery-free mote for perpetual indoor ble applications. In *Proceedings of the 5th Conference on Systems for Built Environments*, BuildSys '18, page 168–171, New York, NY, USA, 2018. Association for Computing Machinery. Cited on pages 1, 92, and 120.

[89] Francesco Fraternali, Bharathan Balaji, Dhiman Sengupta, Dezhi Hong, and Rajesh K. Gupta. Ember: Energy management of batteryless event detection sensors with deep reinforcement learning. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 503–516, New York, NY, USA, 2020. Association for Computing Machinery. Cited on page 140.

[90] Ronald Garcia, Roger Wolff, Eric Tanter, and Jonathan Aldrich. Featherweight typestate. Technical report, Technical Report CMUISR-10-115, Carnegie Mellon University, 2010. Cited on page 89.

[91] Paul Gavrikov, Pascal E. Verboket, Tolgay Ungan, Markus Müller, Matthias Lai, Christian Schindelhauer, Leonhard M. Reindl, and Thomas Wendt. Using bluetooth low energy to trigger an ultra-low power fsk wake-up receiver. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems*

(ICECS), pages 781–784, 2018. Cited on page 114.

[92] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 49(4):41–51, 2014. Cited on pages 24, 26, 32, and 48.

[93] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless iot. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 83–95, New York, NY, USA, 2019. Association for Computing Machinery. Cited on page 147.

[94] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021. Cited on pages 147 and 148.

[95] Kai Geissdoerfer and Marco Zimmerling. Learning to communicate effectively between battery-free devices. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 419–435, Renton, WA, April 2022. USENIX Association. Cited on page 148.

[96] Hussein EL Ghor, Maryline Chetto, and Rafic Hage Chehade. A real-time scheduling framework for embedded systems with environmental energy harvesting. *Computers & Electrical Engineering*, 37(4):498–510, 2011. Cited on page 97.

[97] Phil Gibbons. 15-745, lecture 5: Introduction to data flow analysis. http://www.cs.cmu.edu/afs/cs/academic/class/15745-s19/www/lectures/L5-Intro-to-Dataflow.pdf, 2019. Accessed: 2020-05-26. Cited on page 66.

[98] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the International Symposium on Architecture Support for Programming Languages and Operating Systems*, 2019. Cited on pages 1, 27, 74, 76, and 148.

[99] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. Manic: A vector-dataflow architecture for ultra-

low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 670–684, New York, NY, USA, 2019. Association for Computing Machinery. Cited on pages 1, 2, and 76.

[100] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM. Cited on pages 49 and 143.

[101] Lakshmi Reddy GopiReddy, Leon M Tolbert, and Burak Ozpineci. Power cycle testing of power switches: A literature survey. *IEEE Transactions on Power Electronics*, 30(5):2465–2473, 2014. Cited on page 138.

[102] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992. Cited on page 24.

[103] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 389–398. IEEE, 2013. Cited on page 62.

[104] Marin S Halper and James C Ellenbogen. Supercapacitors: A brief overview. *The MITRE Corporation, McLean, Virginia, USA*, pages 1–34, 2006. Cited on page 96.

[105] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Sigplan Notices*, volume 38, pages 388–402. ACM, 2003. Cited on page 49.

[106] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *ACM Sigplan Notices*, volume 41, pages 253–262. ACM, 2006. Cited on pages 24 and 49.

[107] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. Cited on pages 24 and 49.

[108] Josiah Hester, Sarah Lord, Ryan Halter, David Kotz, Jacob Sorber, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin

Storer, Steven Hearndon, and Kevin Freeman. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. pages 216–229. ACM Press, 2016. Cited on page 14.

[109] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 5–16. ACM, 2015. Cited on pages 27 and 130.

[110] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, pages 19:1–19:13, New York, NY, USA, 2017. ACM. Cited on pages 2, 14, and 15.

[111] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017. Cited on pages 4, 12, 13, 23, 24, 28, 43, 54, 91, 142, and 146.

[112] Robin Heydon. *Bluetooth low energy: the developer's handbook*, volume 1. Prentice Hall Upper Saddle River, 2013. Cited on page 92.

[113] Matthew Hicks. Clank: Architectural Support for Intermittent Computation. pages 228–240. ACM Press, 2017. Cited on pages 13, 24, 26, and 29.

[114] Himax. Hm01b0-mna-01ft870 compact camera module. https://cdn.sparkfun.com/assets/b/3/e/8/e/HM01B0-MNA-Datasheet.pdf, 2019. Accessed: 2022-11-05. Cited on page 73.

[115] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 198–214, 2015. Cited on page 121.

[116] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. Proactive energy-aware programming with {PEEK}. In *2014 Conference on Timely Results in Operating Systems ({TRIOS} 14)*, 2014. Cited on page 88.

[117] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and*

implementation, pages 38–48, 2003. Cited on page 89.

[118] Hyperion Technologies. Gnss200. https://space-for-space.com/wp-content/uploads/2020/04/HT_GNSS200_v2.1-flyer.pdf, 2019. Accessed: 2022-05-25. Cited on pages 126 and 134.

[119] iota Biosciences. Neural dust platform. https://iota.bio/Technology, 2022. Accessed: 2022-05-17. Cited on page 2.

[120] Bashima Islam and Shahriar Nirjon. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 95–109, 2020. Cited on pages 91 and 144.

[121] Bashima Islam and Shahriar Nirjon. Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), September 2020. Cited on pages 2 and 76.

[122] Vikram Iyer, Hans Gaensbauer, Thomas L Daniel, and Shyamnath Gollakota. Wind dispersal of battery-free wireless devices. *Nature*, 603(7901):427–433, 2022. Cited on page 2.

[123] Vikram Iyer, Ali Najafi, Johannes James, Sawyer Fuller, and Shyamnath Gollakota. Wireless steerable vision for live insects and insect-scale robots. *Science Robotics*, 5(44), 2020. Cited on page 85.

[124] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, page 193–204, New York, NY, USA, 2019. Association for Computing Machinery. Cited on pages 1, 113, and 130.

[125] Dhananjay Jagtap and Pat Pannuto. Repurposing cathodic protection systems as reliable, in-situ, ambient batteries for sensor networks. 2021. Cited on page 148.

[126] Hrishikesh Jayakumar, Arnab Raha, and Vijay 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. Cited on pages 14 and 67.

[127] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst.*, 16(3), April 2017. Cited on pages 2, 14, and 143.

[128] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 65. IEEE Press, 2005. Cited on pages 1 and 120.

[129] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 253–266, New York, NY, USA, 2013. Association for Computing Machinery. Cited on page 62.

[130] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, New York, NY, USA, 2013. Association for Computing Machinery. Cited on page 62.

[131] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the International Symposium on Computer Architecture*, 2018. Cited on page 49.

[132] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *ACM Sigplan Notices*, volume 37, pages 96–107. ACM, 2002. Cited on page 1.

[133] Aravind Kailas, Davide Brunelli, and Mary Ann Ingram. A simple energy model for the harvesting and leakage in a supercapacitor. In *2012 IEEE International Conference on Communications (ICC)*, pages 6278–6282, 2012. Cited on pages 103 and 140.

[134] kangaruppel. App apds9960 chain. `https://github.com/CMUAbstract/app-apds9960-chain/blob/capy_v1/src/main.c`, 2017. Error Loc.: main.c: line 534. Cited on page 79.

[135] kangaruppel. Hmc app. https://github.com/CMUAbstract/hmc-app, 2017. Error Loc.: main.c: line 436. Cited on page 79.

[136] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. pages 661–676. ACM Press, 2013. Cited on page 121.

[137] Kemet Electronics Corporation. Supercapacitors FT Series. https://content.kemet.com/datasheets/KEM_S6014_FT.pdf, 2020. Cited on pages 16, 96, and 103.

[138] Kemet Electronics Corporation. Supercapacitors FT Series. https://content.kemet.com/datasheets/KEM_S6014_FT.pdf, 2020. Cited on page 130.

[139] Junaid Ahmed Khan, Hassaan Khaliq Qureshi, and Adnan Iqbal. Energy management in wireless sensor networks: A survey. *Computers & Electrical Engineering*, 41:159–176, 2015. Cited on page 120.

[140] Brian Kihun Kim, Serubbable Sy, Aiping Yu, and Jinjun Zhang. Electrochemical supercapacitors for energy storage and conversion. *Handbook of Clean Energy Systems*, pages 1–25, 2015. Cited on pages 15 and 126.

[141] Hyung-Sin Kim, Michael P. Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. System architecture directions for post-soc/32-bit networked sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 264–277, New York, NY, USA, 2018. Association for Computing Machinery. Cited on page 1.

[142] Sehwan Kim and Pai H. Chou. Size and topology optimization for supercapacitor-based sub-watt energy harvesters. *IEEE Transactions on Power Electronics*, 28(4):2068–2080, 2013. Cited on page 120.

[143] Vaclav Knap, Lars Kjeldgaard Vestergaard, and Daniel-Ioan Stroe. A review of battery technology in cubesats and small satellite solutions. *Energies*, 13(16):4097, 2020. Cited on pages 126 and 149.

[144] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411,

New York, NY, USA, 2016. ACM. Cited on page 49.

[145] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 58:1–58:13, Piscataway, NJ, USA, 2016. IEEE Press. Cited on pages 49 and 143.

[146] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Bfree: Enabling battery-free sensor prototyping with python. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4):1–39, 2020. Cited on pages 14 and 77.

[147] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Greehnouse monitoring. https://doi.org/10.5281/zenodo.3563082, 2020. Error Loc.: tics/test/functional/checkpoint/greenh-temp-time/greenh_temp_time.c : line 236. Cited on page 79.

[148] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, New York, NY, USA, 2020. ACM. Cited on pages 4, 53, 77, 79, 146, and 148.

[149] Paul Kreczanik, Pascal Venet, Alaa Hijazi, and Guy Clerc. Study of supercapacitor aging and lifetime estimation according to voltage, temperature, and rms current. *IEEE Transactions on Industrial Electronics*, 61(9):4895–4902, 2014. Cited on page 138.

[150] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. Meeting lifetime goals with energy levels. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, page 131–144, New York, NY, USA, 2007. Association for Computing Machinery. Cited on page 120.

[151] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. Cited on page 48.

[152] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, February 1980. Cited on page 48.

[153] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. Cited on pages 7 and 66.

[154] Y. Le Cun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, and W. Hubbard. Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989. Cited on page 113.

[155] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998. Cited on page 27.

[156] Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. A modular 1mm 3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 402–404. IEEE, 2012. Cited on page 2.

[157] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002. Cited on page 48.

[158] Philip Levis and David Gay. *TinyOS programming.* Cambridge University Press, 2009. Cited on page 48.

[159] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and others. TinyOS: An operating system for sensor networks. *Ambient intelligence*, 35:115–148, 2005. Cited on pages 24, 26, 32, 33, 43, and 48.

[160] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Shane Leonard, Pat Pannuto, Prabal Dutta, and Philip Levis. The tock embedded operating system. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 45. ACM, 2017. Cited on pages 48 and 144.

[161] Xiang LingXiang, Huang JiangWei, Sheng Weihua, and Chen TianZhou. The

design and implementation of the dvs based dynamic compiler for power reduction. In *International Workshop on Advanced Parallel Processing Technologies*, pages 233–240. Springer, 2007. Cited on page 89.

[162] Linux Kernel Organization. Software interrupt context: Softirqs and tasklets. https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html. Accessed: 2018-11-16. Cited on page 34.

[163] Shaobo Liu, Qinru Qiu, and Qing Wu. Energy aware dynamic voltage and frequency selection for real-time systems with energy harvesting. In *Proceedings of the conference on Design, automation and test in Europe*, pages 236–241. ACM, 2008. Cited on page 97.

[164] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. ACM. Cited on pages 3 and 148.

[165] Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh. Resource aware programming in the pixie os. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 211–224, 2008. Cited on page 121.

[166] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. Computational nanosatellite constellations: Opportunities and challenges. *GetMobile: Mobile Comp. and Comm.*, 25(1):16–23, June 2021. Cited on pages 4, 15, 73, 79, 92, 120, 129, and 138.

[167] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. Computational nanosatellite constellations: Opportunities and challenges. *GetMobile: Mobile Computing and Communications*, 25(1):16–23, 2021. Cited on pages 14, 73, and 130.

[168] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM SIGPLAN Notices*, volume 50, pages 575–585. ACM, 2015. Cited on pages 3, 9, 10, 24, 30, 76, 79, 91, 132, 133, and 146.

[169] Kaisheng Ma, Xueqing Li, Mahmut Taylan Kandemir, Jack Sampson, Vijaykr-

ishnan Narayanan, Jinyang Li, Tongda Wu, Zhibo Wang, Yongpan Liu, and Yuan Xie. Neofog: Nonvolatility-exploiting optimizations for fog computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 782–796, New York, NY, USA, 2018. ACM. Cited on page 148.

[170] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 526–537. IEEE, 2015. Cited on pages 2 and 26.

[171] Shuai Ma, Modi Jiang, Peng Tao, Chengyi Song, Jianbo Wu, Jun Wang, Tao Deng, and Wen Shang. Temperature effect and thermal impact in lithium-ion batteries: A review. *Progress in Natural Science: Materials International*, 28(6):653–666, 2018. Cited on page 2.

[172] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K Saul, and Geoffrey M Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 57–70, 2013. Cited on page 62.

[173] Kiwan Maeng. Photo sense jit. https://github.com/CMUAbstract/samoyed/blob/master/src/photo_sense_jit.c, 2020. Error Loc.: photo_sense_jit.c: line 166. Cited on page 79.

[174] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2017. Cited on pages 3, 4, 11, 12, 13, 24, 25, 26, 29, 30, 32, 37, 42, 43, 91, and 133.

[175] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 129–144. USENIX Association, 2018. Cited on pages 12, 13, 91, 133, and 147.

[176] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1101–1116, 2019. Cited on pages 4, 13, 14, 16, 53, 54, 61, 67, 73, 74, 75, 77, 79, 91, 105, 131, 132, 146, and 147.

[177] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1005–1021, 2020. Cited on pages 77, 88, 91, 95, 97, 98, 105, 106, 130, 131, 140, 142, 144, 147, and 148.

[178] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks*, EWSN '21, 2021. Cited on page 146.

[179] Amjad Yousef Majid, Patrick Schilder, and Koen Langendoen. Continuous sensing on intermittent power. In *2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 181–192, 2020. Cited on page 148.

[180] Robert Margolies, Peter Kinget, Ioannis Kymissis, Gil Zussman, Maria Gorlatova, John Sarik, Gerald Stanje, Jianxun Zhu, Paul Miller, Marcin Szczodrak, Baradwaj Vigraham, and Luca Carloni. Energy-Harvesting Active Networked Tags (EnHANTs): Prototyping and Experimentation. *ACM Transactions on Sensor Networks*, 11(4):1–27, November 2015. Cited on page 1.

[181] Pierre Mars. Coupling a supercapacitor with a small energy harvesting source. https://www.eetimes.com/coupling-a-supercapacitor-with-a-small-energy-harvesting-source, 2012. Accessed: 2021-11-15. Cited on page 96.

[182] Barbara McKissock, Patricia Loyselle, and Elisa Vogel. Guidelines on lithium-ion battery use in space applications. https://ntrs.nasa.gov/api/citations/20090023862/downloads/20090023862.pdf, 2009. Cited on page 126.

[183] Louis Meile, Anja Ulrich, and Michele Magno. Wireless power transmission

powering miniaturized low power iot devices: A review. In *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 312–317, 2019. Cited on page 4.

[184] Geoff V. Merrett, Alex S. Weddell, A. P. Lewis, Nick R. Harris, Bashir M. Al-Hashimi, and Neil M. White. An empirical energy model for supercapacitor powered wireless sensor nodes. In *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, pages 1–6, 2008. Cited on page 140.

[185] Azalia Mirhoseini and Farinaz Koushanfar. Hypoenergy. hybrid supercapacitor-battery power-supply optimization for energy efficiency. In *2011 Design, Automation Test in Europe*, pages 1–4, 2011. Cited on page 120.

[186] Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz 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. Cited on page 13.

[187] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, David A Wood, et al. Logtm: log-based transactional memory. In *HPCA*, volume 6, pages 254–265, 2006. Cited on pages 24 and 49.

[188] Clemens Moser, Davide Brunelli, Lothar Thiele, and Luca Benini. Real-time scheduling for energy harvesting sensor nodes. *Real-Time Systems*, 37:233–260, 10 2007. Cited on pages 91 and 97.

[189] Murata. Murata supercapacitor technical note, 2020. Cited on pages 96 and 105.

[190] B. Murmann. Adc performance survey 1997-2021. `http://web.stanford.edu/~murmann/adcsurvey.html`. Accessed: 2022-04-21. Cited on page 110.

[191] Saman Naderiparizi, Aaron N Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R Smith. Wispcam: A battery-free rfid camera. In *RFID (RFID), 2015 IEEE International Conference on*, pages 166–173. IEEE, 2015. Cited on page 2.

[192] Gustavo Willy Nagel, Evlyn Márcia Leão de Moraes Novo, and Milton Kampel. Nanosatellites applied to optical earth observation: a review. *Revista Ambiente*

& Água, 15, 2020. Cited on page 125.

[193] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, ENSsys'19, page 8–14, New York, NY, USA, 2019. Association for Computing Machinery. Cited on pages 2, 14, 15, 56, 73, 74, 77, 79, 92, 113, 120, 130, 132, and 138.

[194] NASA. Estimating the temperature of a flat plate in low earth orbit. https://www.grc.nasa.gov/www/k-12/Numbers/Math/Mathematical_Thinking/estimating_the_temperature.htm, 2017. Accessed: 2022-06-06. Cited on page 2.

[195] Cristóbal Nieto-Peroy and M Reza Emami. Cubesat mission: From design to operation. *Applied Sciences*, 9(15):3110, 2019. Cited on page 126.

[196] NVIDIA. NVIDIA Jetson TX2/TX2i System-on-Module Data Sheet. https://developer.nvidia.com/jetson-tx2-nx-system-module-data-sheet, 2021. Accessed August 12, 2021. Cited on page 92.

[197] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 336–349. IEEE, 2018. Cited on pages 49 and 143.

[198] EE World Online. Supercapacitor esr, specification and life time, 2020. Cited on pages 15 and 140.

[199] James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015. Cited on page 88.

[200] Chulsung Park and Pai H. Chou. Ambimax: Autonomous energy harvesting platform for multi-supply wireless sensor nodes. In *Sensor and Ad Hoc Communications and Networks, 2006. SECON'06. 2006 3rd Annual IEEE Communications Society on*, volume 1, pages 168–177. IEEE, 2006. Cited on page 120.

[201] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In

Proceedings of the 10th ACM Workshop on Hot Topics in Networks, pages 1–6, 2011. Cited on page 62.

[202] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, 2012. Cited on page 62.

[203] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 267–280, New York, NY, USA, 2012. Association for Computing Machinery. Cited on page 62.

[204] Dimitris Patoukas. Cm accel sound. https://github.com/TUDSSL/InK/tree/master/Application/cm_accel_sound/cm_acc, 2018. Error Loc.: thread2.c: line 148. Cited on page 79.

[205] Dimitris Patoukas. Powercast ar. https://github.com/TUDSSL/InK/tree/master/Application/powercast_ar, 2018. Error Loc.: thread2.c: line 140. Cited on page 79.

[206] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 265–276. IEEE, 2014. Cited on pages 49, 142, and 143.

[207] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015. Cited on pages 49, 142, and 143.

[208] Matthai Philipose, Joshua R Smith, Bing Jiang, Alexander Mamishev, Sumit Roy, and Kishore Sundara-Rajan. Battery-free wireless identification and sensing. *IEEE Pervasive computing*, 4(1):37–45, 2005. Cited on page 2.

[209] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 364–369, 2005. Cited on page 1.

[210] Powercast Corporation. P2110B 915MHz RF Powerharvester Receiver. http://www.powercastco.com/products/powerharvester-receivers/,

2017. Cited on pages 73 and 129.

[211] Proteus Digital Health. Proteus Discover. http://proteus.com, 2016. Cited on page 2.

[212] Jordi Puig-Suari, Clark Turner, and William Ahlgren. Development of the standard cubesat deployer and a cubesat class picosatellite. In *Aerospace Conference, IEEE Proceedings*, 2001. Cited on pages 125 and 127.

[213] R. Demcko, A. Stanziola, D. West. Powering iot modules using solar panels, supercapacitors, and an automatic buck/boost controller ic. https://www.kyocera-avx.com/docs/techinfo/PulseSupercapacitors/Powering_IoT_Modules.pdf, 2020. Accessed: 2022-05-23. Cited on pages 126 and 138.

[214] Jan M Rabaey. The swarm at the edge of the cloud the new face of wireless (keynote presentation). In *Proc. Symp. VLSI Circuits, Kyoto, Japan*, pages 6–8, 2011. Cited on pages 1 and 147.

[215] S Radu, M Uludag, S Speretta, J Bouwmeester, A Dunn, T Walkinshaw, P Kaled Da Cas, and C Cappelletti. The pocketqube standard issue 1. Technical report, TU Delft, 2018. Cited on pages 123 and 127.

[216] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. Design considerations for solar energy harvesting wireless embedded systems. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 64. IEEE Press, 2005. Cited on page 1.

[217] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170, 2012. Cited on pages 2, 10, and 13.

[218] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. A probabilistic approach to energy-constrained mixed-criticality systems. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019. Cited on page 88.

[219] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

Cited on page 49.

[220] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V Merrett, and Alex S Weddell. Restop: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 18(1):172, 2018. Cited on pages xx, 3, 4, 16, 17, 53, 75, and 82.

[221] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 139–152, New York, NY, USA, 2011. Association for Computing Machinery. Cited on page 88.

[222] Emily Ruppel and Bradley Denby. Tartan artibeus hardware. https://github.com/cmuabstract/tartan-artibeus-hw, 2022. Cited on page 124.

[223] Emily Ruppel and Bradley Denby. Tartan artibeus hardware. https://github.com/cmuabstract/tartan-artibeus-sw, 2022. Cited on page 124.

[224] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy harvesting, computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019. Cited on pages 7, 54, 87, and 146.

[225] Emily Ruppel, Kiwan Maeng, Graham Gobieski, Milijana Surbatovich, and Brandon Lucia. Getting started with intermittent computing. Interactive Tutuorial held at International Symposium on Microarchitecture (MICRO), 2017. Cited on page 85.

[226] Adnan Sabovic, Ashish Kumar Sultania, and Jeroen Famaey. Demonstration of an energy-aware task scheduler for battery-less iot devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, page 586–587, New York, NY, USA, 2021. Association for Computing Machinery. Cited on pages 91 and 92.

[227] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages

187–197. ACM, 2006. Cited on page 49.

[228] Saleae, Inc. Saleae logic 8 tech specs. https://saleae.com, 2020. Accessed: 2020-03-06. Cited on pages 73 and 112.

[229] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008. Cited on pages 1, 2, 14, 15, and 73.

[230] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011. Cited on page 121.

[231] Joshua San Miguel, Karthik Ganesan, Mario Badr, and Natalie Enright Jerger. The eh model: Analytical exploration of energy-harvesting architectures. *IEEE Computer Architecture Letters*, 17(1):76–79, 2017. Cited on pages 92 and 143.

[232] Giancarlo Santilli, Cristian Vendittozzi, Chantal Cappelletti, Simone Battistini, and Paolo Gessini. Cubesat constellations for disaster management in remote areas. *Acta Astronautica*, 145:11–17, 2018. Cited on page 125.

[233] Koen Schaper. Transiently-powered battery-free robot. https://github.com/TUDSSL/IpRobot, 2017. Accessed: 05-12-2022. Cited on pages 2, 14, 15, and 79.

[234] Koen Schaper. Inkbot. https://github.com/TUDSSL/InK/tree/master/InkBot, 2018. Error Loc.: appinit.c: line 21. Cited on page 79.

[235] Seiko Instruments Inc. CPX Capacitors CPX3225A752D. https://www.sii.co.jp/en/me/datasheets/chip-capacitor/cpx3225a752d/. Cited on pages 16, 96, 103, 104, and 111.

[236] SEMTECH. LoraWAN Standard. https://www.semtech.com/lora/lorawan-standard, 2021. Visited August 11, 2021. Cited on page 92.

[237] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995. Cited on pages 24 and 49.

[238] Lukas Sigrist, Rehan Ahmed, Andres Gomez, and Lothar Thiele. Harvesting-

aware optimal communication scheme for infrastructure-less sensing. *ACM Trans. Internet Things*, 1(4), jun 2020. Cited on page 148.

[239] Silicon Labs. I2c humidity and temperature sensor. https://cdn-learn.adafruit.com/assets/assets/000/035/931/original/Support_Documents_TechnicalDocs_Si7021-A20.pdf, 2016. Accessed: 2022-11-05. Cited on page 73.

[240] Farhan Simjee, Devyani Sharma, and Pai H. Chou. Everlast: Long-life, supercapacitor-operated wireless sensor node. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, page 315, New York, NY, USA, 2005. Association for Computing Machinery. Cited on page 120.

[241] Skeleton Technologies. White Paper: Equivalent Series Resistance. https://cdn2.hubspot.net/hubfs/1188159/Whitepapers/160809_whitepaper_ESR.pdf, 2016. Accessed August 12, 2021. Cited on pages 103 and 105.

[242] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D Corner, and Emery D Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 161–174. ACM, 2007. Cited on page 120.

[243] Phillip Stanley-Marbell and Diana Marculescu. An 0.9x1.2, low power, energy-harvesting system with custom multi-channel communication interface. In *Proceedings of the conference on Design, automation and test in Europe*, pages 15–20. EDA Consortium, 2007. Cited on page 14.

[244] J Gregory Steffan and Todd C Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 2–13. IEEE, 1998. Cited on page 49.

[245] STMicroelectronics. NUCLEO-G474RE. https://www.st.com/en/evaluation-tools/nucleo-g474re.html. Cited on page 113.

[246] STMicroelectronics. Stm32 power shield, nucleo expansion board for power consumption measurement (um2243). https://www.st.com/en/

evaluation-tools/x-nucleo-lpm01a.html. Cited on page 105.

[247] StMicroelectronics. 9-axis inemo inertial module (imu): 3d magnetometer, 3d accelerometer, 3d gyroscope with i2c and spi. https://www.st.com/en/mems-and-sensors/lsm9ds1.html, 2015. Accessed: 2022-05-25. Cited on page 132.

[248] StMicroelectronics. Nemo inertial module: always-on 3d accelerometer and 3d gyroscope, datasheet - production data. https://www.st.com/resource/en/datasheet/lsm6ds3.pdf, 2017. Accessed: 2019-07-15. Cited on pages 53, 59, 73, and 114.

[249] STMicroelectronics. Ai expansion pack for stm32cubemx, 2020. Cited on page 113.

[250] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 180–195, 2001. Cited on page 89.

[251] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008. Cited on page 48.

[252] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/o dependent idempotence bugs in intermittent systems. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2019. Cited on pages 10, 13, 77, 133, and 146.

[253] Milijana Surbatovich, Limin Jia, and Brandon Lucia. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 851–866, New York, NY, USA, 2021. Association for Computing Machinery. Cited on pages 4, 16, 53, 54, 77, and 146.

[254] Milijana Surbatovich, Brandon Lucia, and Limin Jia. Towards a formal foundation of intermittent computing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020. Cited on pages 4 and 10.

[255] Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R Smith. Wisent: Robust downstream communication and storage for computational rfids. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016. Cited on page 148.

[256] Texas Instruments. SimpleLink™ 32-bit Arm Cortex-M3 multiprotocol 2.4 GHz wireless MCU with 128kB Flash. https://www.ti.com/product/CC2650. Cited on page 113.

[257] Texas Instruments. 10ua-100ma, 0.05% error, high-side current sensing solution reference design tipd135. http://www.ti.com/tool/TIPD135, 2020. Accessed: 2020-03-06. Cited on pages 73 and 112.

[258] Texas Instruments Inc. 4-bit bidirectional multi-voltage level translator for open-drain & push- pull. https://www.ti.com/product/LSF0204, 2021. Accessed: 2022-05-25. Cited on page 132.

[259] The SciPy Community. numpy.random.poisson . https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.poisson.html, 2018. Accessed: 2018-05-03. Cited on page 43.

[260] Moritz Thielen, Lukas Sigrist, Michele Magno, Christofer Hierold, and Luca Benini. Human body heat for powering wearable devices: From thermal energy to application. *Energy conversion and management*, 131:44–54, 2017. Cited on page 1.

[261] Sandeep Thirumala, Arnab Raha, Sumeet Gupta, and Vijay Raghunathan. Exploring the design of energy-efficient intermittently powered systems using reconfigurable ferroelectric transistors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, 2021. Cited on pages 2 and 143.

[262] Sandeep Krishna Thirumala and Sumeet Kumar Gupta. Reconfigurable ferroelectric transistor–part ii: Application in low-power nonvolatile memories. *IEEE Transactions on Electron Devices*, 66(6):2780–2788, 2019. Cited on page 2.

[263] TI Inc. Overview for MSP430FRxx FRAM. http://ti.com/wolverine, 2014. Accessed: 2014-07-28. Cited on page 10.

[264] TI Inc. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf, 2017. Cited on pages 71

and 109.

[265] TI Inc. Products for msp430frxx fram. http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/ultra-low-power/msp430frxx-fram/products.page, 2017. Accessed: 2017-04-08. Cited on pages 10 and 132.

[266] TI Inc. Msp430fr5994 launchpad development kit. https://www.ti.com/lit/ug/slau678b/slau678b.pdf, 2019. Accessed: 2022-11-05. Cited on pages 73 and 79.

[267] TI Inc. User's guide msp430 gcc toolchain. https://www.ti.com/lit/ug/slau646f/slau646f.pdf, 2020. Accessed: 2022-05-25. Cited on page 133.

[268] TI Inc. Msp430fr599x, msp430fr596x mixed-signal microcontrollers (rev d). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf, 2021. Cited on pages 2, 4, and 10.

[269] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 17, 2016. Cited on pages 2, 13, 24, 26, 29, and 30.

[270] Rafael Vicentini, Leonardo Morais Da Silva, Edson Pedro Cecilio Junior, Thayane Almeida Alves, Willian Gonçalves Nunes, and Hudson Zanin. How to measure and calculate equivalent series resistance of electric double-layer capacitors. *Molecules*, 24(8):1452, 2019. Cited on page 96.

[271] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011. Cited on pages 49 and 142.

[272] Michael J Voss and Rudolf Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102, 2001. Cited on page 89.

[273] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. *Leibniz International Proceedings in Informatics, LIPIcs 106 (2018)*, 106:24, 2018. Cited on page 88.

[274] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 105–114. IEEE, 2015. Cited on page 88.

[275] Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, Volkmar Sieh, and Wolfgang SchröDer-Preikschat. Operating energy-neutral real-time systems. *ACM Trans. Embed. Comput. Syst.*, 17(1):11:1–11:25, August 2017. Cited on page 88.

[276] Z. G. Wan, Y. K. Tan, and C. Yuen. Review on energy harvesting and energy management for sustainable wireless sensor networks. In *2011 IEEE 13th International Conference on Communication Technology*, pages 362–367, 2011. Cited on page 88.

[277] Webber, Allan. Tms320f28069: power cycle. `https://e2e.ti.com/support/microcontrollers/c2000-microcontrollers-group/c2000/f/c2000-microcontrollers-forum/945343/tms320f28069-power-cycle`, 2020. Accessed: 2022-06-09. Cited on page 138.

[278] Alex S Weddell, Geoff V Merrett, Tom J Kazmierski, and Bashir M Al-Hashimi. Accurate supercapacitor modeling for energy harvesting wireless sensor nodes. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 58(12):911–915, 2011. Cited on page 140.

[279] Alex S Weddell, Dibin Zhu, Geoff V Merrett, SP Beeby, and BM Al-Hashimi. A practical self-powered sensor system with a tunable vibration energy harvester. 2012. Cited on page 1.

[280] Harrison Williams, Michael Moukarzel, and Matthew Hicks. Failure sentinels: ubiquitous just-in-time intermittent computation via low-cost hardware support for voltage monitoring. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 665–678. IEEE, 2021. Cited on pages 2 and 109.

[281] Fan Yang, Ashok Samraj Thangarajan, Sam Michiels, Wouter Joosen, and Danny Hughes. Morphy: Software defined charge storage for the iot. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 248–260, 2021. Cited on pages 14, 73, 91, and 92.

[282] Hengzhao Yang and Ying Zhang. Analysis of supercapacitor energy loss for power management in environmentally powered wireless sensor nodes. *IEEE transactions on power electronics*, 28(11):5391–5403, 2013. Cited on pages 96 and 140.

[283] Hengzhao Yang and Ying Zhang. A task scheduling algorithm based on supercapacitor charge redistribution and energy harvesting for wireless sensor nodes. *Journal of Energy Storage*, 6:186 – 194, 2016. Cited on page 140.

[284] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 41–53. ACM, 2018. Cited on pages 3, 12, 13, 23, 26, 73, 76, 77, 78, 79, 91, 132, 142, and 144.

[285] Zac Manchester. KickSat. http://zacinaction.github.io/kicksat/, 2015. Cited on pages 1, 127, and 130.

[286] Heng Zeng, Carla S Ellis, Alvin R Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review*, 36(5):123–132, 2002. Cited on page 88.

[287] Heng Zeng, Carla Schlatter Ellis, Alvin R Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2003. Cited on page 88.

[288] Jie Zhan, Alex S. Weddell, and Geoff V. Merrett. Adaptive energy budgeting for atomic operations in intermittently-powered systems. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys '20, page 82–83, New York, NY, USA, 2020. Association for Computing Machinery. Cited on page 147.

[289] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009. Cited on page 97.

[290] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. Moo: A batteryless computational rfid and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011. Cited on

pages 2 and 14.

[291] Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob 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*, page 9. ACM, 2011. Cited on page 147.

[292] Pengyu Zhang, Deepak Ganesan, and Boyan Lu. QuarkOS: Pushing the operating limits of Micro-Powered sensors. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM, May 2013. USENIX Association. Cited on page 2.

[293] Xuan Zhang, Tao Tong, Svilen Kanev, Sae Kyu Lee, Gu-Yeon Wei, and David Brooks. Characterizing and evaluating voltage noise in multi-core near-threshold processors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 82–87. IEEE, 2013. Cited on page 95.

[294] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 421–432. IEEE, 2013. Cited on pages 49 and 143.

[295] Yuhao Zhu and Vijay Janapa Reddi. Greenweb: Language extensions for energy-efficient mobile web computing. *SIGPLAN Not.*, 51(6):145–160, jun 2016. Cited on page 121.

[296] Dennis Zogbi. Supercapacitors: A 25-year market review. https://www.tti.com/content/ttiinc/en/resources/marketeye/categories/passives/me-zogbi-20200806.html, 2020. Accessed: 2021-11-11. Cited on page 15.

[297] Dmitry N Zotkin, Ramani Duraiswami, and Larry S Davis. Joint audio-visual tracking using particle filters. *EURASIP Journal on Advances in Signal Processing*, 2002(11):162620, 2002. Cited on page 85.