System Support for Intermittent Computing

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Alexei Colin

M.S., Electrical and Computer Engineering, Carnegie Mellon University
B.A., Computer Science, Harvard University

Carnegie Mellon University
Pittsburgh, PA

May, 2018

Acknowledgments

I would like to thank my advisor, Prof. Brandon Lucia, who provided indispensable guidance as we explored an exciting new research domain. His technical ideas, positive outlook, eloquence, and writing skills have shaped me as a researcher. I am grateful to my thesis committee — Dr. Alanson Sample, Prof. Anthony Rowe, and Prof. Yuvraj Agarwal — for their service, feedback at each stage of my dissertation work, and the experience they shared with me throughout my studies. I thank Dr. Alanson Sample for his mentorship during my internship at Disney Research in Pittsburgh, PA. This work was funded by gifts from Disney Research and Google, by NSF grant CNS-1526342 and NSF CAREER Award CCF-1751029.

I have been fortunate to join a young research group and witness it grow with great minds and personalities over the years. I thank Vignesh Balaji for discussing architecture with me for hours at a time and for tolerating my rants. I thank Emily Ruppel for her dedicated work on our joint projects, for finding bugs in my hardware circuits, and for keeping the lab in order. I thank Kiwan Maeng for his numerous insights and constructive criticisms on topics related to my research. I thank Milijana Surbatovich and Graham Gobieski for their companionship in our intermittent computing research. I am grateful to Graham Harvey for his contribution to hardware design during our collaboration.

I am grateful to every member of the faculty, staff, and student in the ECE and CS departments at CMU from whom I had the chance to learn throughout the years. Among them are Prof. Nathan Beckmann, Prof. Ragunathan Rajkumar, Prof. Onur Mutlu, Dr. Saugata Ghose, Dr. Gaurav Bhatia, and fellow graduate students Hyoseung Kim, Junsung Kim, Reza Azimi, Adwait Dongare, Niranjini Rajagopal, Max Buevich, Luis Pinto, Artur Balanuta, Nandita Vijaykumar, Jiyuan Zhang, and Diana Zhang. I also thank my collaborators at other institutions – Prof. Josiah Hester at Northwestern University, Prof. Jacob Sorber at Clemson University, and Prof. Przemysław Pawełczak, Amjad Majid, Sinan Yildirim at TU Delft.

I thank my late friend Yuri Ataman for my first ever lessons in computing. Finally, I thank my parents, Irina and Alexandr Colin, for their infinite patience and for instilling discipline that made this work possible.

Abstract

Smart things, spaces, and structures are created by embedding computation into them. Embedded computers sense, compute, and communicate at the edge, closer to the physical rather than the cyber world. Not any computer can be embedded, because many deployment settings demand small size, long lifetime, and robustness to a harsh environment. The energy source of the computer — traditionally a battery — is the most likely component to fail to meet these constrains. Batteries have a limited lifetime, degrade over time, are bulky, cannot tolerate low temperatures, and are costly to replace. A promising alternative power source is an energy-harvesting circuit that extracts solar, mechanical, radio, or thermal energy from its environment. Computers powered by energy harvesters (EHCs) are a promising platform for wearable or ingestible medical devices, industrial or environmental monitoring, and scientific instruments.

From the programmer's perspective, however, it is uniquely challenging to develop software for an energy-harvesting computer than for a traditional battery-powered embedded computer. Software that runs correctly on a battery-powered platform may produce incorrect results or fail to complete at all and is more difficult to debug on an EHC. Software execution on an EHC is *intermittent*, because the device is on for brief intervals, only when sufficient energy is available, and is abruptly interrupted on each power failure. The work in this thesis identifies the challenges of computing reliably and efficiently on intermittently-powered energy-harvesting platforms. We address the challenges of intermittent computing through support across the system stack, from the programming model and runtime systems to hardware mechanisms and tools for program analysis and diagnostics.

The first challenge of intermittent program execution is maintaining progress across power failures and keeping the program state consistent in memory. Prior work has proposed checkpointing mechanisms for maintaining the program state across power failures. We expose the overhead of checkpoints and approach this problem differently, with a new programming model for intermittent software, named Chain. A Chain program is a set of programmer-defined *tasks* that compute and exchange data through persistent *channels*. Chain programs span power failures by completing at least one task between each two power failures. A task can be safely restarted after a power failure and will never see inconsistent state in memory, because its inputs and outputs are in separate channels in memory. We implement Chain abstractions as a library for use with the C language and demonstrate that it ensures correct results despite power failures and reduces performance overhead by 2-7x compared to a checkpointing system for a set of embedded applications.

Task-based programming models, like Chain, allow intermittent execution of long-running applications, but require the programmer to decompose code into tasks. However, some task decompositions prevent the program from terminating or executing efficiently. Programs decomposed into tasks larger than the work executable on stored energy will repeatedly re-

execute a task and never terminate. On the other hand, a decomposition that fragments the program into small tasks has to spend more time and energy persisting program state across tasks. We propose CleanCut, a tool that can check for and report non-termination in a given task decomposition, as well as automatically decompose code into valid tasks. To reason about energy consumption, CleanCut relies on a statistical model for the energy of a program path, derived from measured energy consumption of basic blocks. We applied CleanCut to a set of benchmark applications, to demonstrate a non-terminating path report and a performance improvement of 2.45x using automatic decomposition relative to manual decompositions from prior work.

Tasks in a program may have different preferred modes of execution, i.e. the frequency and duration of each execution. Multiple modes are achievable using the same total energy by using it differently over time. However, software cannot control when the device is off and accumulates energy and for how long it stays on each time it uses stored energy, since this is a function of energy storage capacity and input power. We develop Capybara, a hardware power system with an energy capacity that is reconfigurable at runtime. Capybara grants software some control over when and how much energy is provided to which task. The Capybara software interface allows programmers to associate modes with application tasks, and the runtime system reconfigures hardware energy capacity to match the declared mode. Capybara allows a programmer to write *reactive* application tasks that pre-allocate a *burst* of energy that it can spend in response to an asynchronous external event. Our system decreases the number of missed events by 2-4x over fixed-capacity energy buffer and maintains a response latency within 1.5x of a continuously-powered baseline.

Like all embedded software, development of intermittent programs inevitably involves diagnosing bugs. However, the power failures on intermittent systems may create unique bugs that are not possible in a continuously-powered execution and are difficult to reproduce in debuggers that require the target to be continuously powered. We propose the Energy-interference-free Debugger (EDB), a tool for monitoring and debugging intermittent systems without adversely effecting their energy state. EDB features energy-aware variants of familiar debugging primitives, such as breakpoints, watchpoints, and assertions, and adds primitives specialized for intermittently-powered devices, such as guarded regions with energy compensation. Our evaluation quantifies the energy-interference-free property and shows its value in a set of debugging tasks on an energy-harvesting device.

The final part of this work unifies the developed system support into a design methodology for creating applications on energy-harvesting platforms. This methodology identifies key design tasks common across applications and documents how each of the tools proposed in this work can be used to accomplish each task. We apply this design flow to build the hardware and software for an energy-harvesting board-scale nano-satellite for deployment into low-earth orbit. The instrumental role the proposed system support plays in this application exemplifies how this work can facilitate the development of novel applications of energy-harvesting technology.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedding computers into the spaces and things is necessary to make them *smart*. The intelligence in smart rooms, smart appliances, and smart vehicles, will come from a built-in computer connected to sensors and actuators. Whether the device uses established sensors and actuators, e.g. accelerometers and displays, or emerging ones, e.g. brain waves and molecular dispensers, the sensor data will require processing and the actuators will require control. A processor capable enough to execute software will need to be co-located with the sensors and actuators.

Co-locating a processor with sensors and actuators, as opposed to allocating it in a rack in a data-center, inherits the constraints on size, energy, mobility, and operating conditions of the device itself. Each of these constraints limits the capabilities of the device, such as its compute power, especially indirectly by constraining its power source. Existing devices that need to be mobile, e.g. human-machine interfaces or wearables, rely on a battery for power. In the smallest devices, the battery may dominate the size, weight, and — if it is rechargeable — the cost of the device. These characteristics of batteries limit them to deployments where the device remains easily and cheaply accessible or does not need to last beyond a few years. To be deployed into an environment that does not fit these constraints, such as implanted in the body, molded into a material, or launched into space, an ideal

device would be battery-free.

A battery may compromise the viability of the device in several ways. The battery may deplete and require servicing the device for replacement or decommissioning of an otherwise functional device. For example, battery-powered tire pressure sensors operate approximately 5-10 years [54], which is below the lifetime of a car. The battery may be too large to fit in the form factor of the device, e.g. centimeter-scale for a sensing node or millimeter-scale for an implantable device. For example, on a cyborg insect that carries a sensing "backpack" of at most 1 g, a battery with only 27 mAh capacity occupies 32% of the weight alone [44]. The battery may also fail to withstand the operating conditions that the device might be subjected to. For example, satellites may be cooled to below -40C and heat above 120C while in low-earth orbit [118]. The cold temperature is outside of the rated specifications of lithium-polymer batteries, leading to severe degradation of their capacity. These limitations are inherited from the battery, but not from other parts of the devices, such as digital logic ICs and sensors.

A battery imposes size, lifetime, and other limitations in return for a reliable, predictable, and relatively high-current supply of energy. Reliability and output power of the energy supply, however, can be traded of in favor of smaller size and potentially unlimited lifetime using an energy-harvesting circuit. An energy harvester is an alternative power supply that extracts ambient energy available in the environment around the device. Existing energy harvesters can convert light [154], motion [112, 86], temperature gradients [171, 30] RF radiation [91], or beta radiation [123] into a weak electrical current, on the scale of microamps to milliamps. Research prototypes have successfully used this amount of power to create a pedometer [83], a non-intrusive energy meter [47], sensors for microclimate control [189] and occupancy detection [29], muscle sensors for prosthetics [92], an intraocular pressure monitor [33], a remote control [127], an e-ink paper tag [49], and a diesel engine monitor [184].

Each energy source can be harvested by one or more converters that relies on one of several fundamental physical effects. Light can be converted into electric current using a photovoltaic

Source	Environment	Size	Power	Reference
Light	indoor	elastic	10 μW/cm^2	[181]
Light	outdoor	elastic	$3 - 11$ mW/cm^2	[181, 154]
Thermal	human body	elastic	$10 - 60$ μW/cm^2	[181, 107, 91, 154, 171]
Thermal	hot/cool objects	elastic	$0.1 - 3$ mW/cm^2	[181, 100]
RF	ambient, < 10 km from 1MW	6x6-20x45 cm	$0.0002 - 100\mu$W/cm^2	[91, 130]
RF	dedicated, < 5 m from 1W	15 cm	0.1-2 mW	[148, 134]
Vibration	20-120 Hz	< 1cm^3	$1 - 830$ μW/cm^3	[112]
Compression	piezoelectric pushbutton	3.5x0.5 cm	50 μJ/N	[125]
Flow	1.5-40 m/s	$0.5 - 12$ cm^3	$0.007 - 2$ mW/cm^3	[112, 132, 139, 73]

Table 1.1: Sources of ambient energy and power output range with existing harvesters.

material, such as a solar panel. An RF wave harvester relies on electromagnetic induction and consists of an antenna and a small rectifier circuit. Ambient RF energy [121, 130] may originate from TV, AM/FM radio, cell towers, or Wi-Fi access points. Dedicated RF energy may be generated by an RFID reader or a similar signal generator installed for the purpose of providing energy. Thermoelectric generators (TEGs) convert flow of heat into electric current through the Seebeck effect between two adjacent conductors. Vibration or air flow can be harvested by a piezoelectric material that produces a potential difference in response to physical deformation. Motion, including flow, can rotate a generator built from permanent magnets and coils of wire. Table 1.1 lists several harvesters and their output power achieved in practice.

An energy-harvester implemented within the size constraints of a small sensor node device is less predictable and *weaker* than a battery in terms of both the output voltage and the output current. A *weak* power supply outputs less power than is required by the load to operate. The load on an embedded device is the processor, sensors, and actuators. Since the harvester cannot directly power the device, to turn the device on, the hardware must first accumulate the incoming energy in a capacitor, building up the charge and the voltage until both are sufficient to power the device for some brief interval of time. At the end of this interval, when the accumulated energy has been depleted, the device will abruptly lose power. As a result, the processor will be executing instructions *intermittently* in bursts perforated by power failures, whose spacing will be a chaotic function of input power fluctuations.

Software written for battery-powered embedded devices will not run successfully when

executed intermittently. An embedded software program assumes that the processor will execute it sequentially until completion. The program also assumes that it can keep state in memory, by writing values and reading those values back later. Both of these assumptions, however, are violated on an intermittently-powered processor. After each power failure that takes place during the intermittent execution, the state of the registers and volatile memory is lost. As a result, the execution is interrupted and the current position in the program is lost when the program counter register loses its value. Without the program counter value, the hardware has no way of continuing the execution and run the program to completion when power becomes available again. Furthermore, if the program keeps any state in memory, which includes the stack, then continuing the execution becomes impossible after the memory contents is lost after a power failure.

This fundamental *forward progress* challenge has been identified by recent research and solutions based on checkpointing volatile memory and registers into non-volatile memory have been proposed [141, 101]. However, merely preserving forward progress is not sufficient to bring intermittently-powered platforms to the embedded mainstream, because major challenges remain in reliability, efficiency, and programmability on these platforms.

1.1 Challenges of intermittent execution

The first challenge that hinders intermittent computing is the *lack of reliability* in program execution. Programs that produce correct results when executed on continuous power may produce wrong results or fail to complete at all when executed on intermittent power. Both outcomes are a consequence of an unreliable power supply that can only power the device for a short and, in general, unpredictable amount of time.

The first part of the reliability challenge are errors in the program state induced by the power failures. If the program manipulates data structures in non-volatile memory, then a power failure that interrupts an update to that data structure may leave it in an

inconsistent state. This is a manifestation of the crash consistency problem observed in file systems. However, in contrast to crashes in general-purpose systems, power failures in an intermittently-powered device are the common case and recovery is on the critical path.

The second part of the reliability challenge is the loss of forward progress despite a checkpointing mechanism. A checkpointing mechanism maintains the progress of the execution by latching the intermediate state along the execution. A checkpoint guarantees that progress achieved up to that point will not be lost even if power fails at some later point. However, that guarantee is strictly weaker than the guarantee that the program will eventually execute to the end as long as it would do so on continuous power. The latter guarantee depends on when the checkpoints are taken in the execution. If the amount of work between two checkpoints takes longer to complete than the device can stay on before losing power, then the execution will fail to advance to the next checkpoint. This possibility of *non-termination* is unique to intermittent execution.

The second challenge on the path to intermittent computing is the *efficiency degradation* incurred when executing programs intermittently. Compared to a battery-powered device, on an energy-harvesting device, energy is more scarce, yet there is more work to do due to the overhead of keeping progress of the execution across power failures. The overhead is incurred by the mechanism for latching progress, e.g. checkpointing and recovery, the repeated hardware initialization procedure that must run on every boot after power loss, and the work wasted on re-executing code that failed to execute to completion in a previous attempt. This overhead impacts the runtime of application tasks, e.g. the time to encrypt a message, which may be on the critical path in applications that must react to events in the physical world, e.g. an over-temperature monitor in a factory. Furthermore, any performance degradation from the software execution compounds with the fundamental slowdown relative to a battery-powered device that is due to the time the energy harvester spends charging the capacitor. The cumulative slowdown may make an application that is practical on a battery-powered device impractical on an energy-harvesting device.

The energy-efficiency challenge on an energy-harvesting device is not limited to minimizing energy spent on overhead, but also encompasses the distribution of the energy use over time. Assuming some level of average input energy flow, an energy-harvester has more than one possibility of when to collect, store, and disburse that energy to the load. For example, if a design with a capacitor of size $2C$ units, would disburse $2E$ units of energy every $2T$ seconds, then a design with capacitor C would disburse E every T seconds, for the same average incoming power from the harvester. Each design option determines the size of the *uninterruptible* tasks that can run on the device, the sensing coverage, and the responsiveness of the device. The shorter the charging intervals during which the device is off, the more frequently the device can sense and therefore the shorter its reaction time to sensed values is. On the other hand, the shorter the charging intervals, the less energy is accumulated and is available for the load in one continuous burst. Bursts are required for uninterruptible tasks such as transmission of a radio packet. The *distribution of energy over time* is the third challenge of intermittent computing that arises in applications that react to events in the physical world. This challenge is exacerbated when the application contains tasks with conflicting temporal requirements, such as a sensing task, for which off intervals must be minimized, and a radio transmission task for which the on intervals must be maximized.

The fourth challenge obstructing intermittent computing is the *programmability complexity* of intermittently-powered devices. Writing a program for an energy-harvesting device involves reasoning about interruptions due to power failures and the resulting inactivity intervals, the energy consumption of code, and the energy amount in the capacitor. These concerns add to the concerns of functional correctness and efficiency in embedded firmware. Statements that follow each other in program code may not execute in close succession, if the energy in the capacitor at the time of reaching the first statement is not sufficient to complete all of them. An interruption may violate intuitive assumptions made about correlation between inputs in time. Furthermore, diagnosis of issues is difficult, because such

energy-dependent behavior that is unique to intermittent execution will not manifest in an interactive debugger. Debuggers available today can only be used while the device is continuously powered, and, consequently, while the bugs induced by the intermittent power supply are masked.

Despite the potential advantages of energy-harvesting technology relative to batteries in lifetime, size, and operating conditions, the challenges outlined above pose an obstacle in adoption of this technology as a versatile platform for embedded applications. The current practice of engineering a solution to each challenge in each application has a prohibitive resource cost that would be better invested into the application itself. The objective of this work is to address the challenges of intermittent execution at the level of the system, below the application level. Our overall approach and the detailed contributions towards accomplishing this objective is presented in the following section.

1.2 Thesis statement and contributions

The objective of this work is to design a system stack that can support applications on intermittently-powered devices. The system brings value to the application by handling the intermittence-induced problems with minimal effort from the developer. The scope of the system stack spans from abstractions in the programming language to mechanisms in the hardware circuit of the device, and includes tools for diagnosis and maintenance. The system stack we propose in this work, once incorporated into an existing embedded development environment for battery-powered devices, serves as evidence for the following thesis.

> System support for intermittent execution at the level of the language, compiler, runtime system, debugger, and the hardware itself improves reliability, efficiency, and programmability of software on energy-harvesting devices.

We build-up the proposed system stack from the following contributions.

1. We propose language constructs for constructing programs that execute correctly de-

spite the interruptions inherent in the intermittent execution model. Correctness, defined as a memory state equivalent to continuous execution, is ensured by the proposed primitives for statically defining tasks and specifying communication patterns between them. Our implementation of the primitives for the C language is released in the Chain library [39].

2. We develop a program analysis that identifies non-terminating paths in intermittent programs and decomposes a program into tasks to ensure it makes forward progress when executed intermittently. Within our compiler module, we develop an energy model based on a combination of measurement and statistical estimation not previously explored in work on energy modeling. In our program analysis we propose a heuristic for decomposing a program into tasks. The heuristic makes its decisions based on estimated energy consumption of code and energy storage capacity of the device. We implemented the analysis within the LLVM framework and released it as CleanCut [40].

3. We design a hardware mechanism for reconfiguring the energy storage capacity of the device. This mechanism enables the system to match the energy demand of application tasks while satisfying their temporal constraints, i.e. how long and how frequently the device should be on. We prototype the proposed circuit in a multi-purpose energy-harvesting hardware platform named Capybara [41] and demonstrate the improved responsiveness and event detection accuracy that energy capacity reconfiguration brings.

4. We build an energy-neutral debugger for energy-harvesting devices that does not interfere with the (intermittent) power supply of the device. Energy-neutrality is a feature not available in existing debuggers, e.g. JTAG, but vital for diagnosing bugs that manifest only when the device is powered intermittently. Our debugger supports passive monitoring of the execution along with the energy level, and interactive debugging with energy-aware primitives uniquely useful in the intermittently-powered context. We use our prototype named EDB [38] to debug several malfunction scenarios on an

RF-powered energy-harvesting device.

5. We bring together the mechanisms and tools that comprise our proposed system stack into a design methodology for building an intermittent computing system. We apply the methodology to build the hardware and software of a solar-powered nano-satellite for the KickSat project [194]. The process covers the application of the Chain language constructs, the CleanCut compiler, energy-storage architecture, and an *in situ* EDB debugger for monitoring and instrumentation. The case-study validates the proposed system stack.

The methods and tools that comprise the system stack developed in this work will lower the barrier to entry into programming energy-harvesting devices. *Accessible* intermittently-powered platforms are a prerequisite for the creation of applications that are outside of the constraints of battery-powered embedded devices.

1.3 Outline

This thesis is organized into seven chapters, including the current introduction. The next chapter, Chapter 2, provides a deeper introduction to energy-harvesting platforms and intermittent execution. Chapter 3 summarizes the research that is most closely related to this work.

Each of the following five chapters elaborates on each challenge of intermittent computing briefly outlined in the introduction and presents our contribution that addresses it in detail. These chapters are ordered by the abstraction level, from language to hardware. Thus, Chapter 4 presents the proposed language primitives of Chain for writing intermittent programs. Chapter 5 presents the CleanCut compiler analysis for decomposing programs into tasks and checking them for non-terminating tasks. Chapter 6 introduces the hardware mechanism for reconfiguring energy storage capacity at runtime. Chapter 7 presents the energy-neutral debugger for energy-harvesting devices. The last chapter in this sequence,

Chapter 8 brings the preceding contributions together into a design methodology applied to build a solar-powered nano-satellite.

Chapter 9 suggests directions for future work and concludes by placing this dissertation into the broad space of intermittent computing on energy-harvesting systems.

Chapter 2

Background

An intermittently-powered device differs from a battery-powered device in the capabilities and limitations of the energy-harvesting hardware and the constraints the hardware places on the execution of software. This chapter describes the operation of energy-harvesting hardware and the intermittent execution model for the software.

2.1 Energy-harvesting hardware

The main components of an energy-harvesting computing device are a harvesting circuit, an energy buffer, and a processor with peripherals, illustrated in Figure 2.1. The harvesting circuit converts energy available from a source in the vicinity of the device into usable electric current. The types of energy harvestable with existing circuits and their approximate power

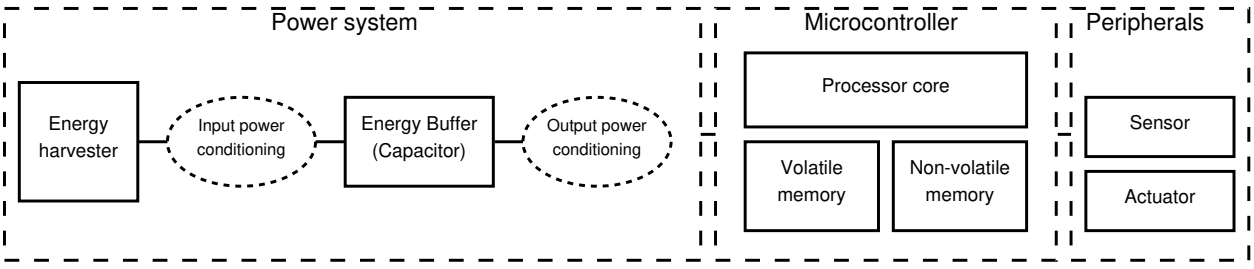


Figure 2.1: Main components of an energy-harvesting computing device.

output were listed in Table 1.1. The principle of energy conversion varies among energy sources: photoelectric effect for light, electromagnetic induction for RF radiation, MEMS generators for vibration, piezoelectric effect for mechanical compression, Seebeck effect for temperature gradients. The details and optimization of the energy conversion efficiency is outside of the scope of this work. Instead, our focus is on using the energy that has been harvested to sense and compute reliably and efficiently.

The availability of an appropriate energy source and the power output of an energy-harvester determines the feasibility of an application. The energy source might be a part of the environment naturally, e.g. light from the sun in space or in a crop field, or it may be added to the environment for the dedicated purpose of wirelessly powering sensing devices, e.g. an RF transmitter in a room for powering human-machine interface devices. The power output of the energy source must be commensurate with the workload for the application to be feasible. For example, a solar panel of 2 cm^2 that outputs 34 mW [176] can support transmissions on a CC430 radio [174] at 50% duty-cycle with a bandwidth sufficient for infrequent sensor data packets but not for a video broadcast.

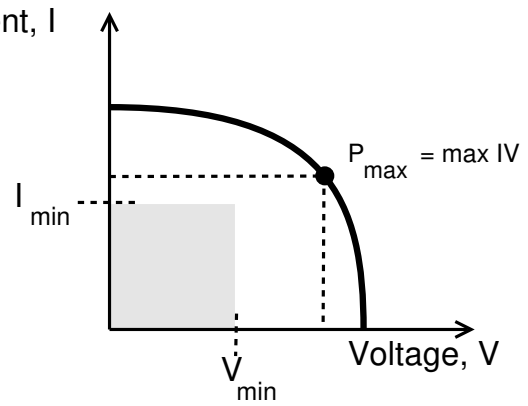


Figure 2.2: Current-voltage (I-V) curve that describes the power output of an energy harvester and the (I, V) point at which power output is maximum. The load cannot operate in the shaded region, delineated by $V_{\min}$ and $I_{\min}$, because the load cannot operate at any voltage and current but requires a minimum voltage and consumes at least a minimum current.

The output of a harvester can be characterized independently of the load by its I-V surface, i.e. a current-voltage curve per time instance [197]. An I-V curve describes how much current the harvester can supply at a given voltage, as illustrated in Figure 2.2. If the current, I, that can be supplied at the minimum voltage required by the load, $V_{\min}$ is

insufficient to power the load, then that load will never turn on despite the net positive power output from the harvester, $P_{\text{in}} = IV > 0$ for $V < V_{\text{min}}$. This particular relationship between harvester power output and load power demand, $0 < P_{\text{in}} < P_{\text{load}}$, is the common case with small devices whose harvesters must be small, e.g. a centimeter-scale solar panel or antenna for RF harvesting. In order to use the energy from such *weak* harvesters that cannot power the load directly, the harvester must be supplemented with additional hardware: a capacitor, a power conditioning circuit, or both.

The simplest power conditioning circuit capable of using weak harvesters is a rectifier, i.e. a keeper diode. This configuration is obtained from the power system diagram in Figure 2.1 by using a diode as input power conditioning and omitting the output power conditioning. The resulting circuit is illustrated in Figure 2.3. The capacitor charges whenever the harvester output voltage is above the voltage on the capacitor, and the diode prevents the capacitor from discharging when the harvester voltage is below the capacitor voltage. Eventually, even under constant ambient energy, the capacitor will charge to the open-circuit voltage output of the harvester at those ambient energy conditions. The harvester output voltage will start low, loaded by the high charging current, at the left-most edge of the I-V curve. As the capacitor charges, the charge current decreases and the harvester voltage rises, moving along the I-V curve to the right.

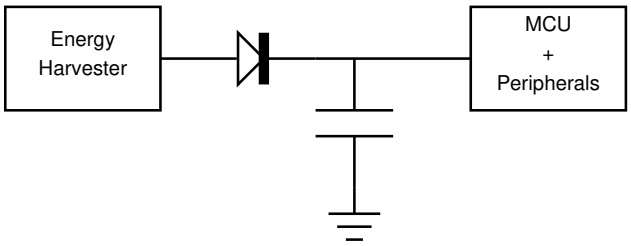


Figure 2.3: A power system with simple input power conditioning using a keeper diode.

In this simple power system, the load, i.e. the microcontroller and peripherals, are always connected to the energy reservoir and must not drain energy from it while it is charging. This requires the load to support low-power modes in which it can monitor the capacitor voltage and wake up only when that voltage is sufficient. Furthermore, this simple power system cannot guarantee that the capacitor will charge to a particular voltage, even assuming a

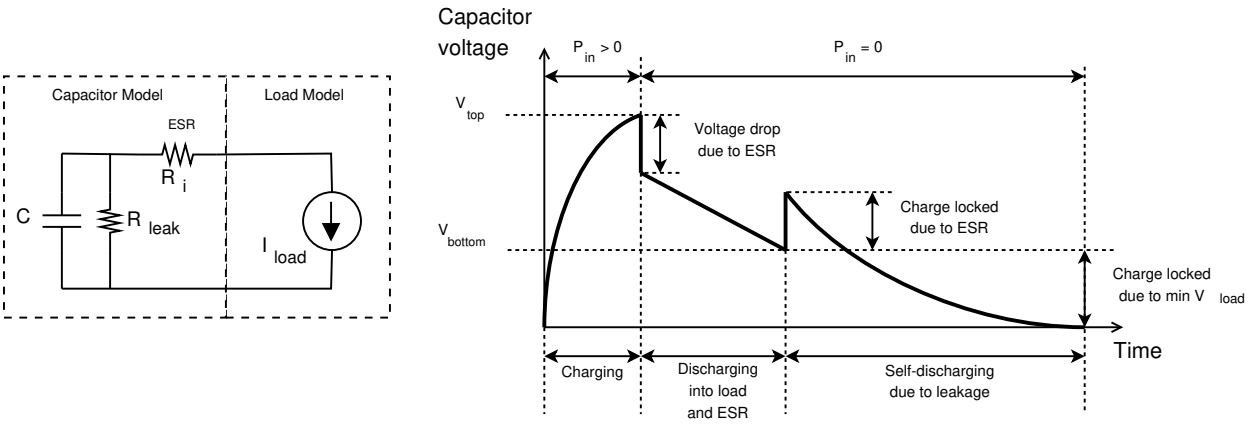


Figure 2.4: Energy losses during charging and discharging of a capacitor.

positive incoming energy, because the I-V curve shape varies with ambient energy. We will present power conditioning based on a voltage converter that can offer this guarantee in Section 2.3, after discussing energy storage hardware in the next section.

2.2 Energy storage buffer

The function of a capacitor in an energy-harvesting device is to accumulate energy over time until sufficient voltage has been built up to power the load. At that point an external trigger, discussed in more detail below, can start supplying power from the capacitor to the load. As the capacitor discharges its voltage, V_{cap}, drops, and the load can continue to operate only for as long as the capacitor voltage is above the minimum voltage required by the load, i.e. $V_{\text{cap}} \geq V_{\text{min}}$.

A capacitor introduces three major inefficiencies. Figure 2.4 illustrates the charging and discharging of the capacitor and the associated losses. First, energy is lost to the resistance on the path to the capacitor when charging and on the path from the capacitor when discharging. Second, only a fraction of the capacity of a capacitor can be used to store energy deliverable to the load, the remaining fraction must be filled with charge that cannot be used. This residual energy must be wasted before the load can be powered and the same energy is left after the load can no longer operate: $E_{\text{residual}} = \frac{1}{2}CV_{\text{min}}^2$, where V_{min} is the minimum voltage

required by either the load or the output power conditioning circuit (if present). Residual energy lengthens the time the device needs to start from an empty capacitor, which we refer to as *cold start time*. A second consequence is the need to provision a higher capacity than that strictly necessary to store the energy required by the load.

Third, energy is lost to leakage and to internal resistance, i.e. equivalent series resistance (ESR), because real capacitors are not ideal energy storage devices. Leakage current flows within a capacitor and discharges the capacitor. ESR contributes to losses during current flow into and out of the capacitor, and droops the voltage available to the load, causing the load to turn off sooner than it would with an ideal capacitor of the same size. ESR grows as the operating temperature decreases, e.g. ESR nearly doubles at -20C for an EDLC supercapacitor [167]. The leakage current, ESR, and other characteristics across capacitor technologies on the market today are listed in Table 2.1 and more are presented in a survey of storage technologies [26].

The data in the table highlights the trade-offs between energy density and the characteristics that create loss. Ceramic capacitors have negligible leakage and ESR, [1] but have a low density compared to all other listed technologies. Tantalum or niobium oxide capacitors are denser, but have higher leakage and ESR. Aluminum electrolytic capacitors are much larger in size (lower density) and have a limited lifetime. Supercapacitors and thin-film batteries are the only practical option for capacities above 1 mF due to comparatively lower density of other technologies, which translates into high volume occupied in the device. The properties of supercapacitors vary widely across models, and generally fewer capacity values per model are available, compared to other technologies. The age-induced degradation in supercapacitors, electrolytic capacitors, and thin-film batteries is counterproductive to realizing the perpetual operation potential of energy-harvesting technology. Two further disadvantages of thin-film batteries are their high ESR and low discharge current, e.g. EnFilm [160] supports currents no larger than 5 mA continuous or 10 mA burst, which is less than the

[1] Neither leakage nor ESR is specified in datasheets for most ceramic capacitors

Tech./Model	Capacity	ESR	Leakage	Lifetime	Size	Temp.
Ceramic	$\leq 470\ \mu$F	negligible	negligible	∞	2x1x1-28x9x9	-55 - 150C
AMK [163]	$0.022 - 470\ \mu$F	10 mΩ			1x1x1-5x3x3	-55 - 85C
Tantalum	$4.7 - 6000\ \mu$F	$0.0042 - 15\ \Omega$			3x2x2-7x4x3	-55 - 125C
F93 [10]	$22 - 680\ \mu$F	$0.3 - 2.5\ \Omega$	$0.5 - 6.3\ \mu$A		3x2x2-7x4x3	-55 - 125C
T491 [88]	$0.1 - 1000\ \mu$F	$0.2 - 15\ \Omega$	$0.5 - 40\ \mu$A		3x2x2-7x4x4	-55 - 125C
T52x Poly. [89]	$4.7 - 1500\ \mu$F	$4.5 - 90$ mΩ	$6 - 400\ \mu$A		3x2x2-7x4x4	-55 - 125C
Niobium Oxide	$2.2 - 1000\ \mu$F	$0.003 - 8.3\ \Omega$	$1 - 80\ \mu$A		2x1x1-7x6x4	-55 - 125C
OxiCap [11]	$2.2 - 1000\ \mu$F	$0.003 - 8.3\ \Omega$	$1 - 80\ \mu$A		2x1x1-7x6x4	-55 - 125C
CoreCap [9]	$330 - 560\ \mu$F	400 mΩ	$28 - 38\ \mu$A		2x1x1-7x6x4	-55 - 125C
Alum. El.	4.7μF $- 1$F	$0.010 - 21\ \Omega$		1000-37000 hrs	3x3-35x52	-55 - 105C
AFK [42]	$22 - 6800\mu$F	$0.035 - 1.35\ \Omega$	$3\ \mu$A	2000-5000 hrs	4x6-18x16	-55 - 105C
MZJ [120]	$22 - 1800\mu$F	$6 - 360$ mΩ	$3\ \mu$A	2000 hrs	5x6-10x10	-55 - 105C
Alum. Poly.	$4.7 - 4700\mu$F	$3 - 242$ mΩ		1000-20000 hrs	4x1-10x21	-55 - 125C
SP-Cap [126]	$2.2 - 220\mu$F	$15 - 110$ mΩ	$3 - 66\ \mu$A	1000 hrs	7x4x2	-40 - 105C
EDLC	6.8-1F	$35 - 300\ \Omega$		500-2000 hrs	3x2x1-48x30x25	-40 - 85C
BestCap [8]	6.8-1000 mF	$30 - 300$ mΩ	$5 - 120\ \mu$A	"unlimited"	15x17x2-48x30x7	-20- 70C
TDK [167]	350 mF	70 mΩ			25x20x2	-20 - 70C
CPX/CPH [153]	2.5-11 mF	$25 - 160\ \Omega$		10,000 cycles	3x2x1	-30 - 70C
XH [153]	80 mF	$100\ \Omega$		10,000 cycles	5x1	-20 - 60C
DCN [74]	0.3-1 F	$850 - 1000$ mΩ	$6 - 100\ \mu$A	500,000 cycles	4x11	-40 - 60C
DMHA [114]	70 mF	200 mΩ	60 μA	50,000 cycles	20x20x4	-40 - 85C
Thin-film battery						
EnFilm [160]	$\approx 1.6\ F$	$120\ \Omega$	3% per year	4,000 cycles	26x26 mm	-20 - 60C

Table 2.1: Specifications of capacitor (and thin-film battery) technologies on the market, with energy storage capacity ($4.7 \geq C \geq 1$ F), voltage rating ($V \leq 16$ V), and load frequency (minimum specified) within the range typical of small embedded devices.

power consumption of most radios (e.g. CC430 [174]). This technology design space suggests minimizing the required energy storage capacity to take advantage of the smallest and least lossy ceramic capacitors.

The power losses due to the capacitor diminish the amount of useful work that an energy-harvesting device can perform using the same input energy. Without a capacitor, however, the device will not operate when the harvester current output is too low at the load's minimum voltage, as dictated by the harvester's I-V curve (cf. Figure 2.2). Furthermore, without a capacitor, it is not possible to guarantee that the load will stay on for a minimum interval of time. The load will turn off as soon as the energy source stops outputting power. And, since ambient energy sources are inherently unstable, it is not practical to predict when such a power loss will happen. The system design challenge is to include a capacitor but compensate for its non-ideal properties with power conditioning circuits, and to design a software system that performs the most useful work out of the stored energy extractable from the

capacitor. We discuss both in the following sections.

2.3 Power conditioning circuits

An energy-harvesting hardware design may choose to include power conditioning circuits to compensate for the low output voltage of the harvester and for the voltage droop in the capacitor. A power conditioning circuit converts a voltage upwards, using a DC-DC booster (inductor-based or a charge pump), or downwards, using a DC-DC buck or a Low-Dropout Regulator (LDO). LDOs are smaller but have a lower efficiency compared to inductor-based buck converters, because the former reduce the voltage by dissipating energy. In addition to voltage conversion, a booster may include a circuit for tracking the harvester's maximum power-point and supervisory circuits for signaling overvoltage or undervoltage conditions on the capacitor (e.g. BQ25504 includes both features). Maximum power-point tracking dynamically regulates the current load on the harvester so that the voltage, dictated by the I-V curve of the harvester, stays at the value that maximizes power, i.e. the product IV, as illustrated in Figure 2.2.

As shown in Figure 2.1, a power conditioning circuit may be placed on the *input* path, i.e. between the harvester and the capacitor, on the *output* path, i.e. between the capacitor and the load, or on both paths. Input power conditioning allows the capacitor to charge to a voltage that is above or below the maximum voltage output of the harvester. Output power conditioning bridges the capacitor voltage range with the often different voltage range of the load.

All types of converters, either on the input path or the output path, consume board area and add energy overhead, because their conversion efficiency is strictly below the ideal 100%. Nevertheless, power conditioning may be included at expense of area and efficiency to improve along other dimensions, e.g. capacitor volume, harvester size, i.e. solar panel area, antenna length or area, thermocouple area, vibration mass. Power conditioning also

adds versatility to the device, allowing it to work with a wider selection of capacitor models, harvester types and sizes, and loads, with minimal configuration changes to the power system circuit. Output power conditioning may generate a net positive savings in energy when the savings from powering the load with a lower voltage exceed the conversion overhead, as will be described precisely later in this section.

An input boost converter enables a harvester to charge a capacitor to a higher voltage than the harvester is able to output. Typical harvesters of small size, output a voltage that is lower than then maximum voltage rating of most capacitors. As a result, the harvester is unable to fill the capacitor with charge up to its full energy storage capacity. For example, an RF harvester in the Powercast [134] outputs up to 1V when the source is more than 1 meter away and transmitting at maximum power of 30 dBm. A harvester with output V_{h} can charge a capacitor rated for V_{top} only to $\left(\frac{V_{\mathrm{h}}}{V_{\mathrm{top}}}\right)^2$ fraction of its energy storage capacity. For a capacitor rated to $V_{\mathrm{top}} = 2.4V$ and a harvester with output $V_{\mathrm{h}} = 1\mathrm{V}$, this fraction is only 17%.

A boost converter on the output path extends the capabilities and efficiency of the device. Voltage boosting supports a load that requires a higher voltage than the capacitor is rated for, e.g. when powering a gesture sensor that requires a minimum voltage of 3.0V [7] with a CPX3225 supercapacitor rated to 2.6 V [153]. Additionally, the boost converter extracts more energy from the capacitor, since it continues to output a fixed voltage to the load after the capacitor discharges to a voltage below the minimum required by the load. A larger fraction of extracted energy reduces the capacitor volume required to support uninterruptable software operation of the same size (cf. Section 2.6). Boosting capacitor voltage may be a necessity for capacitors with a high internal Equivalent Series Resistance (ESR), defined in Section 2.2. As shown in Figure 2.4, an ESR of R_{i} causes the capacitor voltage to droop under a load current of I_{load} by $\Delta V = I_{\mathrm{load}} R_{\mathrm{i}}$. The droop renders a fraction of otherwise extractable energy in the capacitor unusable. For a capacitor charged to V_{top} and a minimum load voltage of V_{bottom}, the fraction of extractable energy to energy usable under ideal $R_{\mathrm{i}} = 0$

with no power conditioning is

$$\frac{V_{\text{top}}^2 - (V_{\text{bottom}} + I_{\text{load}} R_{\text{i}})^2}{V_{\text{top}}^2 - V_{\text{bottom}}^2}$$

For a supercapacitor rated to 2.6 V with ESR of $R_{\text{i}} = 25\Omega$ (e.g. CPX3225 [153]), thresholds of $V_{\text{top}} = 2.6$ V and $V_{\text{bottom}} = 1.8$ V (minimum operating voltage of an CC430 MCU [174]), load of $I_{\text{load}} = 33$ mA (e.g. radio transmission at full power on the CC430 integrated radio), this fraction is 10%. Adding a boost converter with a $V_{\text{min}} = 0.5V$ minimum input voltage (e.g. TPS61202), raises this fraction to 149% (i.e. more than is possible with a baseline of $V_{\text{min}} = 1.8\ V$ even with an ideal capacitor with $R_i = 0$).

A buck converter on the output can step down the capacitor voltage to within the voltage range supported by the load, V_{min} to V_{max}. This conversion is required if capacitor voltage exceeds the maximum voltage that the load can tolerate, V_{max}. However, even when this is not required, the down-conversion may be used to reduce the load power and energy consumption. A buck converter will save energy as long as its efficiency, η_{buck}, keeps conversion losses below the savings relative to operating at V_{max} [57]:

$$\eta_{\text{buck}} > \frac{2V_{\text{min}}}{V_{\text{min}} + V_{\text{max}}}$$

A buck converter may be combined with a boost converter in a single IC, to obtain the benefits of both converters in a smaller area.

The output voltage of a converter is readily configurable *at design time* with resistor values. Configurability *at runtime* is rarely supported by the converter natively, but may be achievable with an external resistor network at a significant cost in area and leakage current. Although it is desirable to persist runtime configuration across power loss in energy harvesting devices, our survey of the market did not reveal any models that support this feature. Implementing this feature using a non-volatile digital potentiometer (e.g. EEPROM-based) is an open problem.

2.4 Introspection and logic in the power system

The hardware power system of an energy harvesting device decides when to make energy available to the load. Energy can flow from the harvester into the capacitor at any time, however the flow of energy from the capacitor to the load can be opened or closed by the system. The system design determines when the load path is closed and the capacitor accumulates energy, and when the path is open and the load consumes the accumulated energy. Between the time when the load path is closed and when it is opened, the device is either completely off or is in a sleep state. In this section we will model each of these two options and discuss the assumptions in each one. Note that in this discussion at an abstract level, the opening and closing of the path need not be an explicit action taken by the power system; it may be an implicit outcome. For example, the path is implicitly closed when the load stops consuming energy after the voltage supply drops below its minimum required voltage. On the MSP430 MCU, the energy consumption is stopped (up to leakage current), because the built-in power supply supervisor stops the execution on the core.

This section presents several possible *open triggers* that a design may choose to open the load path and *close triggers* to close the load path. A trigger may be implemented entirely in hardware, or partially in software with at least some hardware support circuits. Some or all of the hardware may be on-chip inside a microcontroller or in an external circuit. The energy overhead of each trigger implementation determines the minimum input power required for the device to charge at all. The device will charge only when more power is incoming than wasted by the circuit.

The device state across charge-discharge cycles, i.e. across power failures, follows one of two models: the sleep-model or the power-off model. In the sleep model, the device enters a sleep state upon the close trigger, i.e. the instance when the energy path from the capacitor to the load is closed. In the power-off model, the device turns fully off at that instance. Unlike in the power-off model, in the sleep model, some (volatile) state persists across device charge-discharge cycles. This state may be any subset of all contents of volatile

memory, a designated region of volatile memory, registers, and peripheral configurations and memory-mapped registers (e.g. timer counters). States that retain the smallest subset may be referred to as hibernation or deep sleep, but are adequately described by the same sleep model.

The sleep model implies that the device consumes energy at all times and that the incoming power from the harvester is greater than that consumption. Whenever this assumption is not satisfied, the sleep model is inappropriate, and the device behaves according to the power-off model. In the power-off model incoming power is allowed to go to zero, and the device will charge and boot whenever power becomes available again. The appeal of the sleep model compared to the power-off model is the reduction of wake-up time and overhead needed to resume execution. However, the sleep model is less general than the power-off model. The power-off model is more general than the sleep model, because the former does not make any assumptions about the power state after the path is closed. No state that requires energy to be retained is preserved across charge-discharge cycles. In this work, we adopt the more general power-off model and do not consider the sleep model any further.

2.4.1 Open triggers

The device exits the power-off (or sleep) state in response to an *open trigger* that opens the energy path from the capacitor to the load, and the device enters the power-off (or sleep) state in response to a *close trigger* that closes that energy path.

An open trigger may be implemented as a threshold on the capacitor voltage that when crossed closes a hardware switch between the capacitor and the load or between the capacitor and the output power conditioning circuit (if present). The schematic of such a trigger circuit is shown in Figure 2.5. The threshold can be fixed in hardware at design time, fixed in hardware but configurable with small changes to the component values in hardware, or programmable at runtime. A fixed threshold can be implemented by a voltage divider, comparator and a voltage reference. The comparator and voltage reference are also available

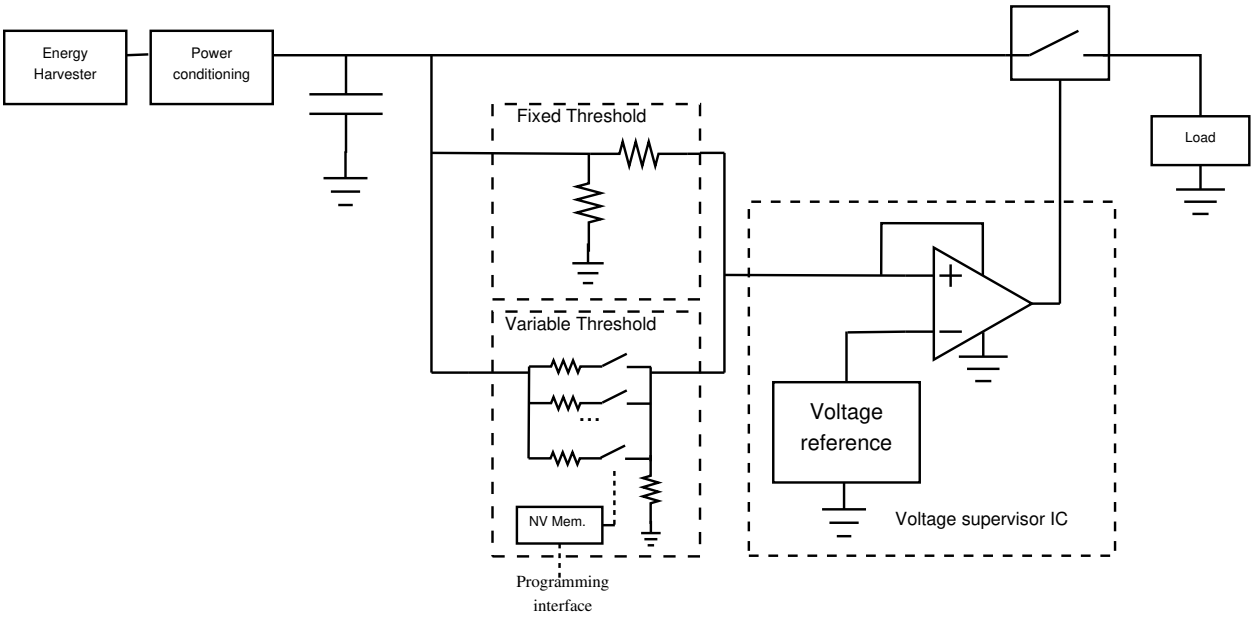


Figure 2.5: Circuit that implements a threshold-based hardware open trigger. Two variants are shown: threshold fixed by component values (either divider resistors or voltage reference) and threshold that is programmable from software.

in a single IC as a voltage supervisor (or reset supervisor) component, e.g. BU49xx. A divider may be necessary to bring the capacitor voltage within the range of the reference. The output of the comparator drives a high-side switch on the load path, e.g. a P-channel MOSFET or a dedicated switch IC like SIP32431. To reconfigure the threshold at design time, either the reference, the supervisor, or the divider components need to be replaced. Supervisors are generally available in small voltage steps, e.g. 0.1V, while references are more common in a few standard values. The overheads of the fixed hardware implementations are leakage current, area, and cost.

A hardware implementation of an open trigger that is reconfigurable at runtime is also shown in Figure 2.5. This implementation includes a resistor network, i.e. a set of voltage dividers, instead of the single voltage divider in the fixed design. One potential implementation of a resistor network is a digital potentiometer. The primary requirement is that the network maintain its setting across power failures, i.e. while the circuit is unpowered, since otherwise any threshold setting applied while the device is executing will be lost as soon

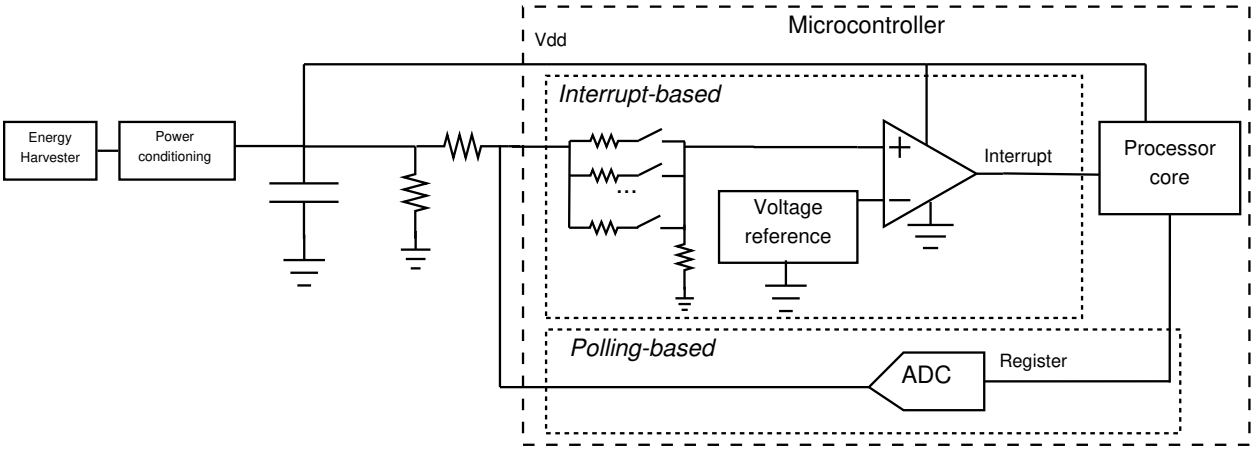


Figure 2.6: Circuit that implements a threshold-based partially-software open trigger, using on-chip components commonly available in general-purpose microcontrollers. Two variants are shown: interrupt-based using a comparator and polling-based using an ADC.

as the execution cycle ends, which will happen before the threshold gets used in the next cycle. A non-volatile digital potentiometer fulfills this requirement, however potentiometers based on EEPROM non-volatile memory have limited endurance. A potential alternative is a resistor network implemented by discrete transistor switches with gate capacitors that maintain the switch state for a limited time after power is lost. The overhead in leakage current, area, and cost of a hardware trigger with reconfigurable threshold is higher than that of a fixed hardware implementation described previously and that of a reconfigurable partially-software implementation described next.

A partially-software implementation of an open trigger relies on the processing core and on-chip peripherals in a microcontroller, as shown in Figure 2.6. This design charges the capacitor with the MCU connected to the capacitor, assuming that the MCU sinks negligible current until the capacitor voltage reaches the MCU's minimum voltage. When the capacitor voltage reaches the microcontroller reset threshold, the microcontroller begins executing an initialization routine that configures the wakeup interrupt and puts the core to sleep to wait for the capacitor to continue charging. The wakeup interrupt comes from an on-chip peripheral that monitors the voltage while the core is in a sleep state and wakes up the core when the voltage crosses the threshold.

An interrupt-based implementation of the partially-software open trigger uses the on-chip comparator while the polling-based implementation uses the on-chip ADC and a timer to periodically wake-up the core to sample the capacitor voltage. A potential advantage of the polling implementation is the lower current in sleep mode, since only a (slow) clock for the timer needs to be kept active as opposed to a reference and a comparator. In both cases, an external voltage divider is required to scale the capacitor voltage within the reference voltage. The value of the divider resistors must be maximized to limit the leakage current through the divider, subject to the input impedance of the input to the comparator or ADC.

The interrupt-based implementation may support either a fixed threshold or a threshold configurable at runtime. A fixed-threshold variant of this design uses the on-chip reference connected to the on-chip comparator. The edge-triggered interrupt on the comparator output wakes up the core. A programmable-threshold variant is implemented using the on-chip resistor network on the input to the comparator. The resistor network tap point is set by the initialization routine that runs on every boot. The polling-based implementation sets up a periodic timer to run a routine that acquires a sample from the ADC channel connected to the capacitor (through a divider) and compares the result to a threshold.

The partially-software design minimizes the area and cost of the hardware, however it adds energy overhead and consumes an analog input pin on the microcontroller. The energy overhead is the power consumption of the microcontroller and any sensors or actuators on the same voltage domain, the power consumption of the on-chip reference, comparator or ADC, resistor network, and the external voltage divider, and the output power conditioning circuit. Since the power conditioning current is part of overhead, the partially-software design is not attractive when output power conditioning is present. The output power conditioning may have a power consumption (78 μA for TPS61202) much larger than that of a sleeping microcontroller ($<$ 1 μA for MSP430FR5969 [173]). The energy loss in this design, even without a power conditioning circuit, may be significantly larger (e.g. 10 μA for MSP430FR5969 [173]) than that of a fixed-threshold external hardware implementation

(e.g. 0.5 μA for the BU49xx supervisor [144]).

The partially-software design does not rely on an external switch on the load path. Instead, it assumes that when the microcontroller is in a sleep state the load consumes less power than the power incoming from the harvester: $P_{\text{sleep}} < P_{\text{in}}$. When the load is in this low-power state, we consider the load energy path closed. For the assumption about relative sleep power and input power to hold, the sleep state must be efficient and the external peripherals (e.g. sensors, radios) must either be in a separate voltage domain with a power-gating switch or or startup into low-power sleep mode by default. A simple implementation of a separate voltage domain is using a microcontroller general-purpose I/O (GPIO) pin as a power supply, in cases when it can source sufficient current for the peripheral, or an external switch. Separate voltage domains may require a level shifter between the sensor and data buses or GPIO pins.

2.4.2 Close triggers

Besides choosing an open trigger, the power system design must also choose a close trigger at which the flow of energy from the capacitor to the load will stop. The simplest close trigger is the implicit brown-out event in the microcontroller. The brown-out event occurs when the microcontroller supply voltage drops below the minimum required for its operation. A supervisory circuit inside the microcontroller might assert the reset line to stop instruction execution and prevent corruption. A brown-out state is distinct from a proper sleep state and may have a higher power consumption initially. However, this close trigger design assumes that as the supply voltage continues to drop, the microcontroller will eventually reach a state with a power consumption that is low enough to charge. This brown-out close trigger is implicit: it requires no components or software logic and incurs no overhead. However, it also provides no information to the software about the power failure event.

An explicit close trigger is a bottom threshold on the capacitor voltage. When the capacitor voltage drops below the threshold, the energy path from the capacitor to the

load is closed. Similarly to the open trigger designs introduced above, this close trigger design requires introspection hardware to monitor the capacitor voltage. Implementations may support fixed or programmable threshold and may be based on interrupts or polling. An interrupt-based fixed-threshold implementation uses an external comparator, while a programmable-threshold implementation uses the internal comparator with a resistor network. In either case, if output power conditioning outputs a regulated voltage to the load, that voltage may be used instead of a dedicated reference, to reduce power consumption overhead. Incidentally, a regulated output reference is not an option for the open trigger implementations since the output is not active during charging.

The internal comparator design can re-use the same comparator to implement both a partially-software open trigger and a close trigger. An external comparator design must dedicate a comparator to each trigger. A polling-based programmable-threshold relies on the ADC to periodically sample the capacitor voltage and can re-use the same ADC channel and timer as a partially-software implementation of the open trigger. The trigger points may be generated by a timer interrupt, in the same way as for the partially-software implementation of an open trigger, or by instructions inserted into the program statically. The requirements for an external voltage divider are the same as for the close trigger implementations, and the same considerations for minimizing leakage apply.

The energy overhead of the close trigger designs is the same as for the corresponding open trigger designs. However, in contrast to the open trigger circuits, the circuits for the close trigger consume energy while the device is on and the microcontroller is executing code. In this state, the energy overhead of the trigger circuit is dominated by the energy consumption of the active microcontroller (e.g. over 100 μA at lowest frequency for MSP430FR5949, which is at least an order of magnitude higher than the trigger overhead cited previously). Besides the hardware overhead concerns, the close trigger introduces the problem of tuning the threshold. A system that employs a threshold-based close trigger to interrupt software execution, such as the just-in-time checkpointing systems that will be reviewed in Section 3.3,

must choose a threshold value appropriate for the software. The threshold choice transcends hardware design, because it must take into account the energy cost of software operations, such as saving a checkpoint. These costs are challenging to estimate, because they vary with execution context, i.e. the state of the program and of the hardware configuration.

Having introduced the implicit close trigger and the explicit close trigger based on capacitor voltage, we must also consider an explicit close trigger that is linked not to the capacitor voltage but to a software action. The software may choose to close the energy path from the capacitor to the load in order to stop the device from consuming energy and start accumulating energy instead. The simplest, software-only implementation of this trigger is a instruction that transitions into a sleep state. An alternative, hardware implementation requires a switch between the capacitor and the output power conditioning circuit (if present) or the load. The switch may be a dedicated high-side load switch IC, e.g. SIP32431, a discrete p-channel FET, or an enable switch that is built into the power-conditioning IC (e.g. the enable feature of a DC-DC converter). The hardware switch control is exposed to the software through a digital GPIO pin on the MCU.

The software and hardware implementations differ in their efficiency of accumulating energy. The energy that can be accumulated is the difference between the input energy and the energy that continues to flow to the load. In the software-only implementation, when the microcontroller is in a sleep state, parts of its internal circuits are power-gated, however the microcontroller and the output power conditioning circuits (if present) continue to draw current from the capacitor. In contrast, a dedicated switch can gate power to the entire circuit connected to the capacitor, limiting the overhead current to the reverse leakage current through the switch (e.g. under 1 μA for SIP32431 load switch). When the hardware switch is closed, however, it also dissipates power due to its non-ideal on resistance, $P_{\text{lost}} = I_{\text{load}}^2 R_{\text{on}}$. For a load of $I_{\text{load}} = 2.6$ mA (MSP430FR5949 at maximum frequency and $R_{\text{on}} = 500$ mΩ (worst-case specification for SIP32431), the lost power is below 5 μW, or 0.05% of the active power of the MCU at 2.4 V. Similarly to the close trigger based on

capacitor voltage threshold, the implementation of the capability opens the challenge of deciding when should the software close the energy path, which we discuss in Section 2.6.

Besides supporting an open or close trigger, introspection of the capacitor voltage may be needed for high-level decision-making in the application, e.g. to reduce workload in response to the depleting energy resource. This introspection can be implemented by connecting the capacitor to an ADC channel, through a voltage divider to match the range of the internal reference. An important consideration in this circuit are two leakage currents through the divider: (1) the path to ground through the top and bottom resistors in series, and (2) the path through the protection diode from the analog input pin to the microcontroller supply rail (VDD). To minimize leakage through the first path, either resistor can be maximized. The second path is a concern when there is an output power conditioning circuit (including a simple switch) between the capacitor and the load, because in this case the voltage on the analog input pin will exceed the voltage on the supply rail whenever the output is not activated. In this state, the protection diode conducts, wasting energy. To minimize this waste, the top resistor in the divider should be maximized, and the output impedance of the divider should be reduced (such that it is well below the input impedance of the analog pin) by reducing the value of the bottom resistor. An alternative approach is to include a high-side analog switch between the capacitor and the pin and enable it only when the MCU is powered (and introspection is desired). A second alternative is to use an external ADC with unprotected inputs.

2.5 Non-volatile memory

To run programs that take more time to complete than the time the capacitor takes to discharge, the program state must be preserved across power failure and restored each time the processor reboots. Since in the general power-off model no power is available to power any volatile memory, the program state must be preserved in non-volatile (NV) memory that

does not require any power to keep its contents. The presence of non-volatile memory on the device becomes a requirement in the power-off model. The properties of and the interface to the non-volatile memory available on the device must inform the design of the mechanisms for saving and restoring state to and from it. The key properties of non-volatile memory are the energy cost for reads and writes, the granularity and type of the read/write interface, and endurance. Non-volatile memory may be available on-chip in the microcontroller or in a dedicated IC that is connected to the microcontroller over a serial or parallel bus. However, a dedicated memory IC, regardless of the underlying technology, cannot provide a LOAD/STORE interface since a microcontroller does not expose its internal memory bus.

The memory technologies potentially suitable for energy-harvesting devices that are available today include Flash (including EEPROM) and ferroelectric memory (FRAM). Both Flash and EEPROM require a relatively high voltage and current for the write (and erase) operation. Flash is dense but has a low endurance and must be erased before over-writing. The erase operation must be performed explicitly and at block granularity as opposed to at byte granularity. EEPROM, a subset of Flash devices, can be read and written at byte granularity, but the endurance of such devices is low. FRAM is a byte-addressable, high-endurance memory [168, 169] with low power reads and writes. When it is integrated on-chip, FRAM is accessible over the same LOAD/STORE interface as volatile SRAM. The main downside of FRAM is lower density compared to Flash, which is a relatively low priority requirement for the primary non-volatile memory in energy-harvesting devices. High-density storage in the form of Flash SD cards, for example, may be added alongside a primary FRAM-based store. In summary, FRAM is the best non-volatile technology for these devices available today.

2.6 Software on intermittently-powered platforms

Under a tight size constraint on the harvester, the hardware power system cannot power the processor continuously. The processor can stay on only for a relatively short interval, T_{on},

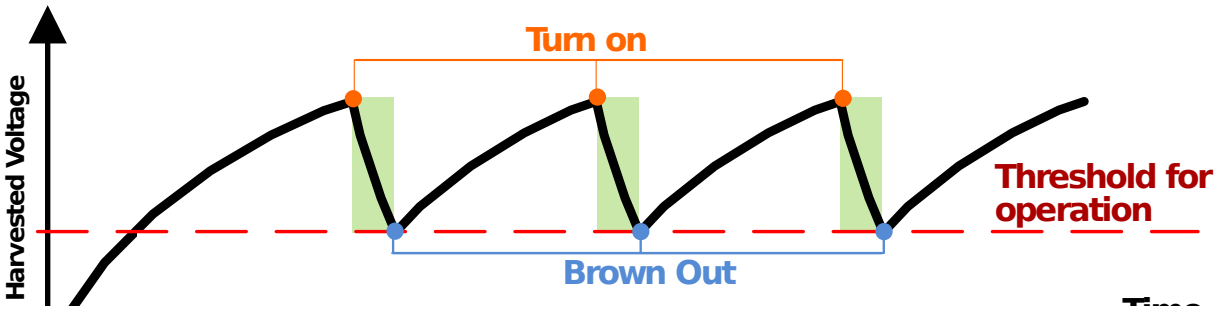


Figure 2.7: Charge-discharge cycle of an energy-harvesting device that forces the processor to compute intermittently.

before the energy in the capacitor is exhausted and power to the processor is lost. The power loss event corresponds to the close trigger that closes the energy path from the capacitor to the processor. After the T_{on} interval, the processor must stay off for a T_{off} interval, until the capacitor is charged again, and the open trigger re-opens the energy path. For example, for the WISP [149] RF-powered device equipped with a 10 μF capacitor, at 1 meter away from the antenna of a 30 dBm RF source, the intervals are approximately $T_{\text{on}} < 5$ ms and $T_{\text{off}} > 80$ ms. On intermittently-powered hardware, software can only execute in bursts that are T_{on} long and is interrupted at the end of each such interval. Since in our general power-off hardware model (cf. Section 2.4), the processor loses power at each interruption, the software loses some program state, which includes the program counter. We define this execution in bursts with loss of state between bursts as the *intermittent execution model*.

Intermittent execution is illustrated on the timeline in Figure 2.7. The intervals when the processor computes are highlighted in green. The length of each compute interval is determined by the capacitor size, and by the incoming power and efficiency of input power conditioning, η_{in}, and of output power conditioning, η_{out}, (if either is present). Under the simplifying assumption of a constant input power, constant load current, and constant efficiency (as a function of voltage and time), the length of the active interval is

$$T_{\text{on}} = \frac{1/2C(V_{\text{top}}^2 - V_{\text{bottom}}^2) + \eta_{\text{in}} P_{\text{in}}}{\frac{1}{\eta_{\text{out}}} I_{\text{load}} V_{\text{load}}}$$

At the end of each compute interval, the processor loses power, and the program loses

some of its state. The program loses all state that resides in volatile structures of the processor, i.e. registers, SRAM memory, caches, memory-mapped I/O registers of on-chip peripherals, and the off-chip peripheral registers and configuration. To resume the execution after the device boots again at the beginning of the following compute interval, the program or the runtime system must save some state to non-volatile memory during the execution and restore it upon reboot. Which state is being saved and when it is saved determines the overhead the program incurs in the intermittent execution model. The following chapter will present several systems for preserving progress using non-volatile memory that were proposed previously and Chapter 4 will present our approach.

Interruptions of the program due to power failure may change the semantics of the program when that program contains I/O. A program that performs only computation has the same semantics in the continuously-powered execution model and in the intermittent execution model, because the computation code will produce the same result whether or not the task is interrupted and resumed. However, a program that performs I/O will not have the same semantics in the intermittent execution model, because most I/O tasks will not succeed if interrupted and cannot be resumed from a partial execution. For example, the transmission of a radio packet must continuously modulate the radio wave for the duration of the packet. If the device loses power in the middle of a packet transmission, the receiver will not be able to decode the symbols and the packet will be lost.

Furthermore, intermittent execution may change semantics if the program makes assumptions about time between different operations. For example, a program that reads two samples in sequence, e.g. temperature and pressure, will get a pair of samples correlated in time if executed on continuous power but not on intermittent power. In the intermittent execution model, a power failure may occur between the two samples, introducing a potentially large delay between them. Due to this problem, the software and hardware of an energy-harvesting system must be co-designed such that the capacitor supports an active interval, T_{on}, that is long enough for uninterruptible software tasks in the application.

In addition to the change in semantics, the intermittent execution model also brings a change in efficiency. Efficiency is a composition of tightly related, but not necessarily interchangeable, metrics: time taken to execute a program and energy consumed to execute that program. The immediately apparent increase in program execution time, relative to the continuously powered baseline, is the time the device spends off to charge its capacitor. Additional energy is lost when charging and discharging the capacitor (Section 2.2) and converting energy in power conditioning circuits (Section 2.3). A secondary source of time and energy inefficiency are the software overheads. First, the software incurs an overhead to save and restore program state to and from non-volatile memory to resume execution after power failures. Second, intermittent execution raises the possibility of *wasting work*, i.e. time and energy, on re-execution. Some code that already executed in a previous compute interval may need to be re-executed in the following compute cycle, unless the system is able to resume computation precisely at the point where it was interrupted. In the next chapter we discuss the trade-offs in the previously-proposed systems that fit this category. Minimizing these software overheads motivates the work proposed in this thesis.

2.7 Debugging and maintenance

After the hardware and software of an energy-harvesting device has been designed, with the considerations described in this chapter, but before the device can be deployed, it must be debugged and tested. After deployment, the device may require maintenance in the form of software updates and telemetry data collection. Tools available to an application developer today are simulators, oscilloscopes, logic analyzers, and JTAG debuggers.

The part of the development that can be done in simulation with an instruction-level or a cycle-accurate architectural simulator is limited by the simulator's limited scope. An architectural simulator focuses on the processing core and cannot easily simulate the other components on the board, i.e. on-chip peripherals and off-chip sensors, actuators, and power

conditioning circuits. An architectural simulator does not model the energy input from the environment and the behavior of all devices on the board under a variable voltage, especially, their energy consumption. An environmental model for energy production and consumption that is sufficiently detailed to capture the phenomena experienced by energy-harvesting devices would be difficult, if not impossible, to construct analytically [108] and expensive to simulate in real-time.

Without a simulator, development proceeds by building custom circuit boards and using tools designed for continuously-powered devices to characterize and debug both the energy-harvesting hardware and the software. The main tools include oscilloscopes, logic analyzers [147], energy consumption gauges (e.g. TI Energy-Trace [170]), and JTAG debuggers. Debugging software using these tools is challenging, because oscilloscopes and logic analyzers can inspect hardware interfaces but not program state, while JTAG debuggers cannot operate with an intermittently-powered device. Recently, specialized tools, e.g. for emulating an energy-harvesting environment, are emerging, which we review in detail in the next chapter. Debugging and maintenance is a major obstacle that hinders the adoption of energy-harvesting platforms by application developers. This problem motivated specialized debugging and instrumentation tools to be part of the system support proposed in this thesis.

Chapter 3

Related Work

Energy-harvesting is an emerging field with problems that span research domains. In this section we review the prior research on problems most relevant to our work on intermittent computing. In Section 3.1, we begin at the hardware level with a review of energy-harvesting devices that have been built. We focus on hardware mechanisms for managing energy and keeping time in Section 3.2. We trace the evolution of computation on intermittent power in Section 3.3. Since some approaches rely on estimating energy at compile time, we review research in that area in Section 3.4. Section 3.5 discusses the work in debugging and maintaining software on energy-harvesting devices.

3.1 Energy-harvesting platforms

An energy-harvesting platform is a battery-free alternative to a battery-powered sensing mote, such as Telos [133], Epic [53], Synergy [5], Amulet [65], Eco [129] and many others, including commercial products, e.g. iBeacon [6]. Hybrid battery-powered platforms use energy-harvesters to recharge batteries. ZebraNet collars [81] were one of the earliest devices to use a solar panel to charge a battery. Heliomote [135] is a generic solar-harvester with a permanent battery that informs an application of instantaneous battery and panel voltage. Trinity [189] is an HVAC sensor that harvests energy from air flow to charge a thin-film

battery. DoubleDip [109] water usage monitor re-charges a small lithium battery using a thermoelectric harvester attached to hot and cold pipes and uses a capacitor to extend the life of the battery. Similarly, Prometheus [78] increases rechargeable battery lifetime by operating from a supercapacitor when solar energy is available. In the Everlast [156] mote, the efficiency of charging a (large, battery-scale) supercapacitor is improved with a custom charging circuit that tracks the maximum power point of the panel. EnHANTs [108] are indoor solar-powered tags with a thin-film battery and active bi-directional radios capable of forming a network. The stacked 1 mm^3 sensing platform [95] charges a thin-film battery in the bottom layer with a solar panel in the top layer, and provides image and temperature sensors and an optical programming interface.

Battery-powered devices or hybrid devices with an energy-harvesting battery charger have some advantages over battery-free devices. Ultimately, application constraints determine which is plausible and which of the plausible ones is best. For example, only battery-powered devices can be deployed in places without an ambient energy source. On the other hand, batteries are not attractive for deployment in places where they cannot be easily accessed for replacement when their lifetime inevitably expires, e.g. inside a structure or a body. Compared to battery-only or hybrid power supplies with battery chargers, energy-harvesting power supplies eliminate the battery to extend the lifetime of the device, reduce weight and size, and improve robustness to harsh operating conditions.

Energy-harvesting devices available today harvest light, RF, motion, heat flow and other sources surveyed in the literature [165, 154, 161]. WISP [149], WISPCam [116], SolarWISP [61], and Moo [196] are sensor nodes with a general-purpose microcontroller, an accelerometer, and a CCD camera (on WISPCam only) that are powered by an RFID reader (915MHz) and a solar panel (SolarWISP only). These devices receive from and transmit to the reader using the backscatter method, driven by a software implementation of the RFID protocol. Bistable e-paper tags [49] are updated using an NFC transmission from a smartphone and display the content without further energy input. A sensor node powered by

ambient RF energy [130] continuously samples temperature and light level and transmits the data wirelessly. Self-powered human-machine interfaces, e.g. a push-button [127], a rotary knob [179], or touch-sensitive paper [86], send a control command over radio in response to a user interaction. Leaf nodes in the Sensornet Tree [193] are powered by indoor light, include output power conditioning (cf. Section 2.3), and communicate with battery-powered nodes in the tree using an off-the-shelf radio IC. Eternal camera [119] takes pictures with a matrix of photo-sensitive elements each of which is powered by the light that it images. Monjolo architecture [47] interprets the energy harvester output as a signal to infer a physical quantity of interest, e.g. electric load.

KickSat [194] is a battery-free solar-powered board-scale satellite capable of sensing the magnetic field, temperature, and inertial quantities, and processing the data, and transmitting packets to Earth using an off-the-shelf radio IC. The original KickSat design is effectively unbuffered, i.e. it will successfully transmit radio packets only as long as the solar panels source more current than the radio consumes, which may not be the case if the satellite is oriented at an obtuse angle towards the sun. Chapter 8 presents an alternative implementation of a satellite in the KickSat form factor, using the same communication stack, but with extended energy management capabilities, and a wider range of ambient energy conditions.

A notable subset of energy-harvesting devices, e.g. the energy-harvesting cell-phone [164] and video camera [115], operate without an energy buffer, i.e. the device is either on at a 100% duty-cycle or not on at all. This is a special case possible when harvested power exceeds the power consumed by the load (cf. Section 2.2). Devices in this category can be viewed as wireless transducers (sensors), rather than as computing nodes. Because computation on such devices usually does not span power failures, the intermittent execution model (Section 2.6) is a valid but not a necessary description. In this work we focus on energy-harvesting devices with on-board compute capability powered by harvesters weaker than the load.

To have a chance on the market, the design of an energy-harvesting computer should be

specialized to a particular application. However, applications can be prototyped on generic versatile platforms that support the combination of harvesters and features needed by the particular application. Flickr [67] is a modular development platform that supports multiple types of harvesters as well as multiple sensors and processors, as pluggable daughter boards with an intelligent power distribution mechanism (cf. UFoP in the next section). Energy-harvesting modules already on the market, e.g. from Powercast [134] and EnOcean [55], accelerate development of energy-harvesting platforms by providing a complete power system on a mini-board for attaching to a device as a single component with a simple interface. Chapters 6 and 8 expand this set of platforms with another versatile prototyping platform that offers a choice of two harvesters – solar or RF (an implementation of the harvester design on the WISP [149]) – and a power system with novel energy-management capabilities.

3.2 Hardware for energy and time management

Besides the type of supported harvesters, energy-harvesting platforms are distinguished by the energy-management capabilities of their power system. The power system determines how much of the input energy ultimately becomes available to the application and when it is made available. Chapter 2 explained how power conditioning circuits and different capacitor technologies affect the energy that becomes available to the application. In this section, we review hardware mechanisms proposed for managing how energy is accumulated and consumed as well as for providing services such as time keeping to the application.

An energy-harvesting device has one or more harvesters (energy producers) and one or more loads (energy consumers). The energy management mechanism inside the power system must combine the energy flow from each harvester, buffer the energy, and then distribute the buffered energy among loads. Focusing on the energy producer side, Ambimax [128] proposes to dedicate a separate capacitor to each harvester, in order to charge each capacitor at the maximum power point of the respective harvester. Recent work, discussed in detail below,

has addressed energy distribution on the load side, across peripherals with disparate energy demands.

Starting with the observation that allowing all peripherals unrestricted access to a shared energy buffer may degrade application performance, the federated energy storage design (UFoP) [66] dedicates separate capacitors to the MCU and each peripheral. The power distribution circuit charges the capacitors in a priority order, at expense of losing some energy in the resistor-based implementation of the circuit. Federation partitions energy is hardware and offers limited control to the software. By dedicating a separate energy buffer to each peripheral, federation trades off the ability to re-direct energy stored in those buffers at runtime. Alternatively, the dedicated assignment of capacitors can be relaxed using a specialized DC-DC converter [24] that supports bidirectional energy flow, including flow from a capacitor to the DC-DC converter into another capacitor. The trade-off of dedicating energy buffers to different components is a manifestation of the classic conflict between a shared and a partitioned resource.

The opposing side of the trade off has been explored in the Dynamic Energy Burst Scaling (DEBS) [57] hardware design that allows the software to programmatically request an energy burst at runtime. By provisioning an energy burst of a controllable magnitude, with DEBS software can avoid task fragmentation across power failures, minimizing the save/restore overhead, and thus reducing the total energy required to execute a sequence of tasks. DEBS is implemented on a platform that uses the partially-software open trigger (cf. Section 2.4) with a programmable capacitor voltage threshold (V_{top}). Chapter 6 presents an alternative mechanism for letting software control the amount of buffered energy, which does not rely on the partially-software open trigger and has a lower start time from an empty buffer.

Energy management across multiple storage elements also arises in mobile devices (e.g. laptops), which may feature multiple batteries. However, unlike in the energy-harvesting context with multiple loads, in the mobile device context, multiple batteries are used to ultimately emulate a single battery but with better properties. Software defined batteries [12]

used this approach to increase the effective capacity under varying load currents. The problem of distributing energy across storage elements located on different nodes in a network has been explored in [199], which supported energy sharing *via wires*.

Energy management mechanisms within the above categories are complimentary to hardware and software mechanisms for reducing power consumption with or without sacrificing performance, e.g. Dynamic Voltage and Frequency Scaling (DVFS) [157] and Dynamic Power Management (DPM) [20], automatic power-gating and scheduling in hardware or in software [96, 50, 200]. Since energy-harvesting devices benefit from these techniques in the same way as battery-powered or even general-purpose plugged-in computers do, this thesis does not investigate such energy-saving techniques, but focuses on the challenges unique to intermittently-powered devices.

The service provided by the hardware to the software application may extend beyond energy management to time keeping. Time keeping is a useful service that is readily available on battery-powered nodes, but is a challenge to provide on energy-harvesting devices [68]. When an energy-harvesting device has no incoming power, and has exhausted its own stored energy, it cannot power a clock and therefore cannot keep time. Although this was not investigated in the original publication, the federated energy storage approach [66] could be applied to dedicate a capacitor to a Real-Time Clock (RTC) component. Alternatively to an actively-powered RTC, time can be estimated by measuring the self-discharge of a (dedicated) capacitor [69] or by counting the cells in SRAM memory whose value has decayed since losing power [137]. While the time keeping problem is out of scope of this thesis, it is of utmost importance for energy-harvesting platforms, since this capability is a prerequisite for a large class of applications, e.g. collecting timeseries of samples and synchronizing clocks across devices to enable communication.

3.3 Computation on energy-harvesting devices

The intermittent execution model defined in Section 2.6 implies that to compute on an energy-harvesting device, the computation task must either be small enough to finish within one active period or the system must preserve execution state in non-volatile memory. DewDrop [28] was the first system to support intermittently-powered devices for programs in the former category, composed of single-shot self-contained tasks. DewDrop's scheduler analyzed available energy and the expected cost of executing a task, and scheduled the task only if the analysis predicted it would complete before energy was exhausted.

Mementos [141] later extended support to programs of arbitrary length, written in the C language. Mementos runtime preserved state across failure periods by periodically copying volatile state into a *checkpoint* saved in non-volatile memory and restoring the state from the last completed checkpoint upon each reboot. A compiler pass part of Mementos inserts energy-state checks at loop backedges and function returns. An energy-state check triggers a checkpoint copy operation if the energy in the capacitor is below a threshold. If introspection hardware for sensing the capacitor voltage (which costs area and energy overhead, cf. Section 2.4) is unavailable (e.g. on the WISP [149] device), Mementos saves a checkpoint at each energy check. Mementos includes an optional timer-based mechanism for skipping energy checks, which effectively spaces checkpoints at a particular time interval. This time interval does not have a direct relationship with energy consumption, but energy consumption determines whether checkpoints are close enough for the program to make progress. If checkpoints are throttled to a period that generates code segments too large to complete on one capacitor charge (cf. Section 2.6), the program may never complete, i.e. it would become a Sisyphean task, in Mementos terminology.

DINO [101] later observed that applications that write non-volatile state as part of the program, risked executing incorrectly under Mementos, because Mementos could leave non-volatile program state inconsistent with the (volatile) program state saved in the non-volatile checkpoint. This memory consistency problem arises in any system that (1) ignores non-

volatile state when saving checkpoints *and* (2) does not guarantee that execution will reach the next checkpoint before power fails (and volatile state is lost). The memory consistency problem induced by intermittent execution is an instance of the more general crash consistency problem studied in data bases [58], file systems [25], and non-volatile memory systems [145, 36, 180, 198, 131, 178, 117, 52] and file systems for non-volatile memory [191]. Crash consistency is ensured by one of many *logging* mechanisms, implemented in software [145] or hardware [198, 122], that fall into the general category of a *transactional* system [58]. Transactional memory (TM) in hardware [64] and software [155, 63, 32] also aims to make code atomic, but targets systems with volatile memory, not a mix of volatile and non-volatile memory, and with parallelism among access requests to the memory, which does not exist in simple low-performance processors in the energy-harvesting scale. Some systems for intermittent computing, including our approach in Chapter 4, places all code in a transaction (i.e. task), which is a concept applied in TCC-like memory models [62, 93].

Unlike in server, desktop, or mobile battery-powered systems, in intermittent systems failure is the common case, and the recovery procedure is on the critical path, since it executes every charge-discharge cycle. Much of the work in non-volatile memory systems outside of the intermittent computing domain, investigates designs that eliminate the performance bottleneck of the persistence barrier, analogous to a synchronization barrier, by allowing non-volatile accesses to be overlapped [131, 198]. Energy-harvesting systems are much lower performance, e.g. there is only one thread of execution and access to non-volatile memory is synchronous, through one port, with no re-ordering, and no cache or a write-through cache. The barrier-induced performance bottleneck does not arise on such systems, rendering much of the work on improving performance of non-volatile (transactional) systems not applicable to this domain. The relative overheads to recover and to save state, lack of parallelism, availability of a load/store interface to persistent storage, lack of hardware support, among other differences, motivate the design of specialized transactional systems for intermittently-powered devices undertaken in this thesis and in the cited related work. Chapter 4 explores

the memory consistency problem in more detail and illustrates the two vulnerability criteria introduced above with an example.

DINO [101] resolves the consistency problem by neutralizing the first clause of the vulnerability criteria: DINO saves a version of non-volatile state along with a copy of the volatile state into the non-volatile checkpoint. Non-volatile state that needs to be versioned, i.e. that is modified between two checkpoints, is identified and added to the checkpoint by a compiler pass. However, DINO inherits the challenge of checkpoint placement from Mementos. Like with Mementos, with DINO, programs may fail to complete if checkpoints are throttled aggressively or may incur a high overhead (for checking the energy state and copying data) if checkpoints are not throttled at all. Chapter 4 presents an alternative approach to supporting computations based on statically-defined *tasks* instead of checkpoints. The task-based approach requires neither checkpoints, as Mementos, DINO, and Ratchet [177] (discussed below) do, nor introspection hardware for measuring the energy-state, as Mementos and DINO do.

Our task-based approach was adopted in later work, Alpaca [106], that introduced a compiler analysis for *privatizing* non-volatile state into local volatile variables and committing the writes to non-volatile memory at task boundaries. Unlike our approach, Alpaca supports shared access to non-volatile variables by multiple tasks, which simplifies the programming model, at the cost of privatization and commit overhead. In an alternative approach taken by Ratchet [177], tasks are defined implicitly by *idempotent regions* [46, 45] identified by a program analysis, and an unconditional checkpoint is inserted at the end of each region. Clank [70] is another approach based on idempotent-regions, which identifies the regions dynamically, at runtime, using a special-purpose custom hardware module proposed to be added to the microcontroller. Dynamic tracking of loads and stores eliminates the conservatism of the compile-time idempotence analysis due to pointer aliasing, at the cost of the tracking overhead, either hardware, as in the case of Clank, or software, as in the case of accesses to arrays in Alpaca. Idempotent region construction has been revisited with condi-

tional checkpoints [190], taken only when a compile-time estimate of the energy consumption of a region exceeds the energy in the capacitor measured at runtime. This system predicates its correctness on the assumption that energy consumption of a code region executing on a complete embedded system can be estimated using a measurement-free model based on assembly instructions alone. We discuss why this assumption does not hold for a practical energy-harvesting device, and revisit the open problem of energy estimation in Section 3.4 and Chapter 5.

The Variable-Grained system [14] uses compile-time dominator tree analysis to automatically decompose a program into atomic regions such that each region has a single entry. Consistency of program state is ensured by maintaining two logs: (1) a non-volatile undo log with old values of overwritten non-volatile memory locations, which is replayed in reverse upon each recovery, and (2) a redo log with updates to volatile state so that these updates can be applied atomically at the end of each region to the checkpoint of volatile state kept in non-volatile memory. Our task-based approach presented in Chapter 4 ensures that programmer-defined tasks are idempotent without requiring any logging overhead, by restricting access to persistent state. HarvOS [21] statically estimates memory consumption (and consequently the size of checkpoints) at different points throughout the program and optimizes the placement of checkpoints to minimize the checkpointing overhead.

Unlike our task-based approach and Alpaca, the systems Ratchet, Clank, and Variable-Grained form regions automatically without accounting for their energy consumption and the energy storage of the device. In the Variable-Grained system region size can be constrained by the number of dynamic instructions, but not by energy cost. The size of such energy-agnostic regions depends on load/store patterns in the code, which may either bound them to a small size, producing a larger checkpointing overhead than necessary, or let them grow without bound, creating a risk of non-termination due to exceeding energy storage capacity. Furthermore, unlike statically-placed task boundaries in DINO and unlike our task-based approach, such implicitly-constructed regions do not readily support programmer-defined

atomic tasks that should not be resumed mid-way after interruption by a power loss mid-way, even if the system guarantees that program state would remain consistent (cf. Section 2.6).

An alternative approach to avoiding the inconsistency problem is to neutralize the second clause of the vulnerability criteria: guarantee that the execution will always reach a checkpoint before power fails. There have been several approaches to fulfill this guarantee. Just-in-time checkpoint systems [17, 16, 111, 76] trigger a checkpoint when an energy-sensing hardware circuit detects the capacitor voltage drops below a threshold. Assuming the worst-case energy cost of the checkpoint can be reliably bounded, and the voltage threshold is set such that it triggers while there is at least that much energy remaining, then the checkpoint will be guaranteed to complete. Hibernus [17] was the first system to propose the just-in-time checkpointing approach with fixed thresholds. Hibernus was later extended in Hibernus++ [16] with a mechanism for measuring worst case checkpointing cost to provide a less conservative estimate of the checkpointing trigger threshold, and a mechanism for measuring the power output of the energy source (by periodically pausing the execution) to adapt the turn-on threshold to the energy source. In terminology from Section 2.4, the turn-on threshold is the open trigger threshold that determines the amount of energy to accumulate before beginning computation. QuickRecall [76] is a just-in-time checkpointing approach for systems which allocate *all* program state into non-volatile memory. Checkpoints in QuickRecall only need to save registers, and thus are very light-weight, however all memory accesses consume more energy and may take more than one cycle because they access non-volatile instead of volatile memory.

A further optimization to the Hibernus [102] just-in-time checkpointing system proposed to enter an SRAM-retaining sleep state if capacitor voltage approached the critical last-chance checkpoint voltage. Sleep states are not a general solution because they assume positive input power at all times, as explained in Section 2.4. As an optimization, the sleep state mode might actually *increase* execution time and total energy consumption, in cases where harvester output is near or below the power consumption of the SRAM-retaining sleep

state, because the capacitor discharge or charge very slowly while the device is sleeping, at a rate strictly worse than the charging rate in fully off state (with a hardware threshold-based open trigger, defined in Section 2.4).

The just-in-time checkpointing systems save fewer checkpoints than systems that collect multiple checkpoints or log memory accesses throughout the execution, however not without disadvantages. To determine the voltage threshold, which marks the last chance to save a checkpoint, the worst-case energy cost of a checkpoint must be estimated for each platform, accounting for all possible states of hardware, including all components in the device. For the estimate to remain valid over time, the capacitor degradation due to aging [35, 153] must be taken into account. Inevitably, the estimate must be conservative, larger than the average, but the more conservative it is, the bigger share of stored energy must be reserved and the less work the device can do on each capacitor charge cycle. By reserving energy for a checkpoint, a just-in-time system requires a capacitor that is at least as large as the size of the checkpoint, i.e. the size of the volatile program state. Even for capacitor above but near this bound, a just-in-time system may use most of the energy available in each charge cycle on saving the checkpoint instead of on executing the program.

In contrast, a task-based system can work with capacitors that are smaller by choosing a task size smaller than the size of a just-in-time checkpoint. The task size in task-based systems is not constrained by the total size of the volatile state in the program. A software solution to supporting smaller capacitors is valuable for space-constrained devices, even when it comes with a performance trade-off. Furthermore, a just-in-time system may be extended to support application-defined atomic sections that must not be resumed in mid-way, however this feature is not a given. One approach is to force a checkpointing before entering the section and disable checkpointing while in the section. However, none of the existing just-in-time systems offer any such support. In contrast, tasks in task-based systems naturally define atomic sections.

Checkpointing systems differ in the size of their checkpoints. Since the stack is a poten-

tially large share of all volatile state (e.g. compared to the set of registers), eliminating it from the checkpoint reduces the time and energy cost of each checkpoint significantly. Unlike Mementos and DINO, Ratchet avoids copying the stack from volatile memory into the non-volatile checkpoint. However, unlike our approach, Ratchet accomplishes this optimization by allocating the stack in non-volatile memory, which is only possible on microcontrollers which map on-chip non-volatile memory into the address space, and not on systems with non-volatile memory in external ICs. Allocating the stack in non-volatile memory, also increases the energy cost and time of each push/pop and load/store operations on local variables that spill onto the stack, because non-volatile memory is slower and consumes more energy than volatile (SRAM) memory. Our measurements on the MSP430FR5969 microcontroller showed an energy difference of up to 1.5x between accesses to FRAM and accesses to SRAM; and, at 16 MHz, SRAM is accessed in a single cycle while FRAM requires 2 cycles [173]. The Variable-Grained [14] system uses a volatile stack, but amortizes the stack copying overhead by maintaining a log of call contexts in non-volatile memory that can be used to repopulate stack frames during recovery. Our approach presented in Chapter 4 avoids the cost of persisting the stack by employing the task-based programming model, which restricts the stack to be local to each task.

The software systems reviewed above and the one presented in this thesis make it possible to run programs intermittently on the same processors used for battery-powered embedded systems. Prior work, however, has explored specialized hardware processor architectures that are fully non-volatile [105, 104, 146, 124], implemented with non-volatile logic [87, 183] instead of CMOS. A fully non-volatile processor lets the execution be interrupted and resumed at instruction granularity, without checkpointing volatile state. Non-volatile logic, however, increases the energy and time for all operations relative to conventional CMOS logic. Furthermore, preserving all execution state *inside the processor* may not be beneficial if that same processor must first run peripheral initialization code upon each boot before resuming the interrupted execution. As is the case for systems with automatically constructed regions,

fully non-volatile hardware may also hinder the programming language from providing an abstraction for atomic regions that must restart from the beginning instead of be resumed mid-way after power failure. Techniques at the hardware design level [111, 110] have been proposed for checkpointing algorithms implemented in custom hardware within Application-Specific Integrated Circuits (ASICs). Research into alternative non-volatile memory technology seeks to bring its access cost closer to volatile SRAM memory [136], which may reduce checkpointing cost without introducing non-volatility within the processor core.

3.3.1 Language-level abstractions

As explained in Section 2.6, software written for a battery-powered device might not execute correctly on an energy-harvesting device. In part, this incompatibility is a consequence of the language used to write the software being agnostic to the implications of the intermittent execution model. Prior to the work in this thesis, to the best of our knowledge, programming models for intermittent computing were not investigated at the language level. Chapter 4 proposes language constructs for expressing programs that execute correctly across power failures. Later work has proposed the Mayfly language [68] for intermittently-powered devices with the power to express timing properties, such as freshness of sensor values. The implementation of Mayfly runtime relies on time-keeping hardware (cf. Section 3.2). The trigger abstraction [29] allows the program to specify actions in response to power conditions and external events.

Language-level features for writing energy-aware programs that originate in battery-powered systems may be applicable to energy-harvesting devices as well. Language abstractions (partially) delegate the detailed timing and mode of I/O to the system, so that it can optimize the energy consumed on that I/O. In Eon [159], tasks are associated with abstract energy states and executed when the system is in the corresponding state. The LAB abstraction [85] lets the programmer declare the required quality of sensing data and leaves it to the system to activate sensors to provide the required data at the minimum

energy cost. Real-time sensor streams [4] provide an abstraction for declaring streams of sensor data that are then scheduled by the system to maximize the device sleep intervals. Type systems can catch energy-related bugs by preventing code that uses high-energy types from running while the device is in a low-energy state. Energy Types [37] attribute energy to application phases via a type system. ENT [31] introduces dynamic types that resolve to different instances based on device energy state. Energy-awareness provided by these abstractions, however, is not sufficient for intermittent computing, because the language must also be aware of intermittent execution.

An emerging class of application-level power-reduction techniques is based on decreasing the workload through approximation in response to changes in the available energy [72, 13, 84, 150]. These systems rely on a mechanism for estimating available energy at runtime, which requires some hardware support (cf. Section 2.4). Changing the workload in order to save energy is an option on energy-harvesting devices whenever it is an option on battery-powered devices, and solutions that exploit this possibility are complementary to the system support developed in this thesis.

In Chapter 4, we define a computational model and a language for intermittent systems as Actors did for concurrent distributed systems [3]. The Actor model has inspired languages for computationally constrained devices, such as embedded systems [18], sensor nodes [94], and spacecraft [23], but not for intermittently-powered energy-harvesting devices. In contrast to the goals of the Actor model, our aim is to extend the language with support for intermittent execution. Actors differ from static tasks introduced in Chapter 4 in that Actors are concurrent, are created dynamically, and are triggered by messages, whereas tasks compose a sequential program, are defined statically, and execute in a fixed order specified by the task graph. The differences in the models have a direct effect on their implementations: an actor system requires a communication network and a dynamic actor instance manager, while a task system performs most of its work at compile time.

3.4 Energy estimation

A capability for estimating how much energy a program will consume on a given platform without running it is useful and sometimes required for (1) decomposition into tasks and non-termination checking addressed in Chapter 5, (2) some checkpointing mechanisms reviewed earlier in Section 3.3, and (3) provisioning energy storage capacity for the application investigated in Chapter 6. However, accurate and reliable energy estimation is a challenging open problem [113] that has been researched previously and as part of the work in Chapter 5.

Energy estimators compute the energy of a program analytically in terms of more basic parts. The energy of each basic part is determined from measurement by the system designer or the hardware manufacturer. Estimators differ in the granularity of the basic parts, which may be counts of fine-grained micro-architectural events [143, 27, 22, 75], instructions [175], basic blocks [99], statements, or processor frequency and voltage alone [158]. Unlike the parts present in the binary, micro-architectural event counts require a simulation and describe the energy of a particular dynamic execution of the binary [27, 22]. To produce absolute estimates, useful beyond a relative comparison, an architectural simulator must be calibrated to a specific platform, which may not be practical for each proprietary embedded processor and other hardware components in each device. Instruction-level current consumption at micro-second scale can be measured using a current-sense resistor and an instrumentation amplifier or using a switching DC-DC converter that can count the number of switches [170]. Unlike instruction energy, basic block energy must be measured for each application. However, this application-specific measurement may be unavoidable for obtaining the full system power, including all hardware components in the device, configured into their application-specific configurations. Basic blocks have the advantage of accounting for the context of each instruction in a sequence [99], which constrains the actualizable energy cost of each instruction. In Chapter 5 we present an energy model based on measurements at the basic-block granularity.

Prior work on this problem has produced either *worst-case* estimates [99, 188], i.e. a

theoretical upper bound on the actual energy consumption, or average-case estimates [90, 158]. Worst-case bounds overestimate energy because they span all possible (and even some impossible) behaviors of the code, including across different inputs, and across different hardware configurations. To account for input-dependent variation in energy consumption, prior work [99] has proposed to profile basic blocks under a subset of different inputs and use an evolutionary algorithm to find the maximal energy. Proposals for computing total program energy generally rely on a static analysis that expresses the total energy in terms of the lower-level building blocks, e.g. instructions or blocks. Work inspired by implicit path enumeration (IPET) program analysis technique [77, 182] expressed the program energy as the sum of energies of basic blocks multiplied by the (unknown) number of times each executes. This expression was used as the objective function in an Integer Linear Program, which encoded control flow and loop bounds (provided by the programmer or profiler) as constraints, and optimized over the unknown execution counts to find the maximum energy. Other approaches [59, 99] represented the energy of a program as a recursive cost relation parameterized on inputs, that can then be solved. If the hardware description at the register-transfer level (RTL) is available for the processor, prior work [34] has proposed to estimate worst-case energy for the processor (but not for the whole device) by simulating program execution and counting the number of times transistors switch.

A worst-case estimate calculated by the analytical models reviewed above may conservatively overestimate the energy that can be actualized in the real achievable worst case, and may be much greater than the average case energy. Furthermore, models based on instruction-level estimates measured outside the context of the application and the rest of the device hardware cannot estimate energy of the *full* system, which is a prerequisite for energy-harvesting use cases listed in the beginning of this section. To describe the energy consumption in more detail than a scalar number and to support full-system estimates, our approach in Chapter 5 is *distributional*, i.e. it estimates the probability distribution of the energy consumption on a device.

3.5 Debugging and maintenance

Research into debugging and maintenance of software on energy-harvesting platforms has resulted in simulators for design exploration [61], tools for reproducing energy traces in the lab [197], and for deploying binaries wirelessly [166, 2, 195]. Prior to these efforts, tools such as Clairvoyant [192], T-Check [98], TinyTracer [162], and KleeNet [151], were focused on battery-powered motes in a wireless sensor networks. We review this work in this section. Complementary to debugging techniques are practices that avoid introducing bugs in the development process, e.g. by using a safe subset of C [79] and an OS with isolation between processes, e.g. enforced by the Rust language in TockOS [97].

Application behavior on an intermittent energy source can be partially inferred from a simulation. The Computational RFID Crash Test Simulator (CCTS) [61] can produce a voltage trace representative of a solar harvester with a specified capacitor size and load. CCTS is useful for exploring the design space for a new energy-harvesting application, but not for *in situ* debugging tasks that Chapter 7 is dedicated to.

Ekho [197] is a device that records the amount of energy harvested by a harvesting circuit and reproduces the trace as power input into the target device. Ekho can reproduce problematic program behavior, but it cannot offer insight into this behavior. Complementary to Ekho's features, in Chapter 7 we develop debugging mechanisms for inspecting the program state and correlating program events to the energy level.

Wisent [166] and Stork [2] offer a protocol and a bootloader that allow software to be reliably downloaded onto RF-powered energy-harvesting devices via a wireless link from an RFID reader. Live updates have also been proposed using in-place code replacement [195]. WINDWare [186] is a middleware for managing data streams from a large set of sensor tags. This support is valuable when the device is in the field, without a debugger, like the one proposed in Chapter 7, attached.

3.5.1 Debugging embedded wireless sensor nodes

Prior to recent interest in energy-harvesting systems [149, 95], there was considerable interest in battery-powered wireless sensor nodes [82, 71]. Sensor nodes necessitated programming [56] and operating system [60, 96] support, which in turn created a need for development and debugging support. Clairvoyant [192] is the closest work from this area, because it provides interactive debugging capabilities. The system tries to minimize its effect on the program being debugged, in terms of memory use, network traffic, and system lifetime. However, because Clairvoyant targets powered nodes, it is not concerned with energy interference. Additionally, Clairvoyant does not focus on debugging primitives specialized to energy-harvesting devices, such as energy compensation, energy-aware assertions and breakpoints, introduced in Chapter 7. Sympathy [138] provides support for debugging *networks* of sensor nodes. The scope of Sympathy is restricted to determining why data collection nodes stop sending data to "sink" nodes in the network. This work uses a series of metrics and an inference step to isolate failures and is largely orthogonal in purpose to debugging within a single energy-harvesting device.

TinyTracer [162] supports lightweight event tracing for sensor node programs written in nesC [56]. Its traces enable execution replay and manual failure analysis. Like TinyTracer, our tool presented in Chapter 7 provides event tracing using watchpoints and I/O bus monitoring. Unlike TinyTracer, our tool traces energy in addition to program events and allows the developer to correlate the two together. T-Check [98] and KleeNet [151] use model checking and symbolic execution (respectively) to *expose* failures in sensor node programs. Both are orthogonal but potentially complementary to our work, as they do not support monitoring or interactive debugging on an intermittently-powered device, but may be useful for diagnosing bugs early, prior to running the application.

Chapter 4

Chain: A Programming Model for Reliable Intermittent Programs

Intermittent execution may cause a program that is correct in a continuous execution to exhibit unpredictable behavior. When a device experiences a power failure, its volatile state (program counter, registers, SRAM) clears, while its non-volatile state (FRAM, flash) persists. As a result, intermittence can impede forward progress and cause data corruption or crashes. Prior work attempted to address these issues, primarily by checkpointing [141, 101]. However, Checkpointing does not scale to large memory sizes because its time and energy overhead is proportional to the amount of program state. Checkpointing overheads deprive the application of the scarce storage and energy on the energy-harvesting computer (EHC). Furthermore, the energy cost of the checkpoint increases the energy storage capacity that the device must be provisioned for.

We propose to make software that runs intermittently reliable without resorting to checkpoints. We introduce the Chain programming and execution model for writing intermittent programs. In a Chain program, the programmer decomposes the computation into a sequence of *tasks*, each of which can perform arbitrary computation and I/O. Tasks are sequenced according to a *task graph* expressed by the programmer.

Chain executes tasks sequentially according to the static control flow specified as part of each task. In case of a power failure, the currently executed task is restarted from the beginning. Chain can restart tasks because it enforces *atomic* all-or-nothing semantics for each task. Program state in volatile and non-volatile memory visible to a task is always consistent. To ensure consistency, Chain uses a novel memory access model for the volatile and non-volatile memory on the EHC. A task has full access to volatile memory, but Chain requires all volatile variables to be local to a task. To pass inputs to and outputs from tasks, the programmer uses Chain's *channel-based* non-volatile memory access model. A task can send a named value to another task (or to a future instance of itself) via a channel dedicated to a pair of tasks (or its own 'self' channel).

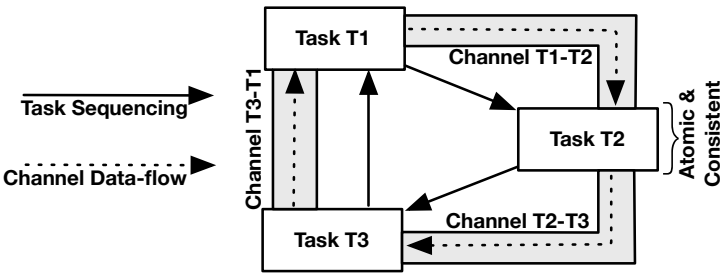


Figure 4.1: A Chain program. The program has three tasks that execute in sequence and pass data to one another via channels.

Chain's channel mechanism guarantees that a task's inputs and outputs are stored in distinct memory locations. The separation of inputs and outputs ensures that a task with any mixture of accesses to volatile and non-volatile memory is arbitrarily restartable after a power failure with *essentially no resume cost*. From the perspective of the task, its inputs are always immutable, consistent, and available in the channel. Figure 4.1 shows a schematic representation of a Chain program with three tasks that are sequenced by explicit control-flow transitions that form a task graph and that exchange data via channels.

In the following sections of this Chapter we define the task and channel abstractions and present their implementation as a runtime library. We also show with experimental comparisons to prior work that Chain more effectively preserves progress, keeps data consistent, and significantly improves on run time performance of checkpointing systems with similar goals, namely DINO [101] by 2-7x and Mementos [141] by 10-150x.

	Feature	Function
Tasks	`task` T, f	Create a task T implemented by function f
	`origin` T	Specify task T to run on first power up
	`self`	Refer to the current task
	`NextTask` T	Transfer control to task T
Channels	`channel` (T_1, T_2), $\{F : type, \dots\}$	Define channel from task T_1 to task T_2 with a set of fields and specify a type for each field F
	`ChIn` F, T	Read field F from channel (T, `self`)
	`ChOut` $\{F \leftarrow v\}, T$	Write value v into field F in channel (`self`, T)
	`ChSync` F, $\{T_1, \dots, T_n\}$	Read the most recent value of field F in channels (T_1, `self`), ..., (T_n, `self`)
	`MultiOut` $\{F \leftarrow v\}$, $\{T_1, \dots, T_n\}$	Write v into field F in channels (`self`, T_1), ..., (`self`, T_n)
Modules	module $M, T_{\text{in}}, T_{\text{out}}$, $\{T_1, \dots, T_n\}$	Create module M with entry task T_{in}, exit task T_{out} and member tasks T_1, ..., T_n
	`ModEnter` M, T	Transfer control to the entry task in module M and make task T the successor of module M
	`ModLeave`	Transfer control to successor of current module
	`ModPut` $\{F \leftarrow v\}, M$	Write value v to field F in the input channel of M
	`ModGet` F, M	Read field F from the output channel of M
	`ModIn` F	Read field F from current module's input channel
	`ModOut` $\{F \leftarrow v\}$	Write v to F in current module's output channel

Table 4.1: The Chain language keywords.

4.1 Task and channels programming model

The Chain programming model uses *task-based control-flow* and a *channel-based memory model* to ensure progress and consistency for intermittent executions without any of the overheads of checkpointing. Task-based control-flow provides a strong notion of progress in the presence of power failures. Channel-based memory allows tasks to exchange data values with guaranteed consistency. Channels ensure that non-volatile data remain consistent by separating a task's inputs from its outputs by construction.

We next describe Chain as a set of extensions to a typical embedded systems base language (e.g., C). Table 4.1 summarizes the Chain language, and Figure 4.2 shows an example Chain program that we use to illustrate Chain's features.

4.1.1 Task-based control-flow

A Chain program is written as a collection of *tasks*. The `task` keyword labels a C function as a Chain task. A task can perform arbitrary computation and is free to define task-local volatile variables (e.g. `s` in task `Sense` in Figure 4.2) and access peripherals (e.g. call to

`sensor()`). The set of tasks in an application form a *task graph* that determines how control flows into and out of each task. Each task has at least one *predecessor task* and at least one *successor task*. One task in the graph is marked as the *origin* task and is the task that executes when the device powers up for the first time.

Each task has a single *entry point* and one or more *exit points*. The entry point is at the top of the task function. The programmer syntactically represents an exit point with a `NextTask` statement. Each `NextTask` statement takes the name of another task as an argument and when the statement executes, control is transferred to the entry point of that task. The programmer can include a `NextTask` statement along any control-flow path in a task, terminating that path. The program in Figure 4.2 has three tasks that are linked into a task graph using `NextTask` statements.

Chain provides the *language-level* guarantee to programmers that tasks are *progress-preserving*. Progress preservation means that control flows from one task to another at a task's endpoint only. Control never jumps discontinuously back to an earlier task and never flows non-deterministically, even in the presence of arbitrarily-timed power failures. For example, after a reboot anywhere in the code in Figure 4.2, the application resumes from one of precisely three locations in the code: the first instruction in `Sense`, `Alert`, or `CmpAvg`. In addition, once execution advances into `CmpAvg`, it cannot enter `Sense` before going through `Alert`; execution follows the task graph. This intuitive control flow behavior cannot be relied on in a system that is not progress-preserving but can experience power failures. In Chain *forward progress* is guaranteed as long as the energy demand of each individual task never exceeds the total energy storage capacity of the device. The programmer can control the energy demands of tasks by choosing the total number of tasks in the application and the amount of work in each task.

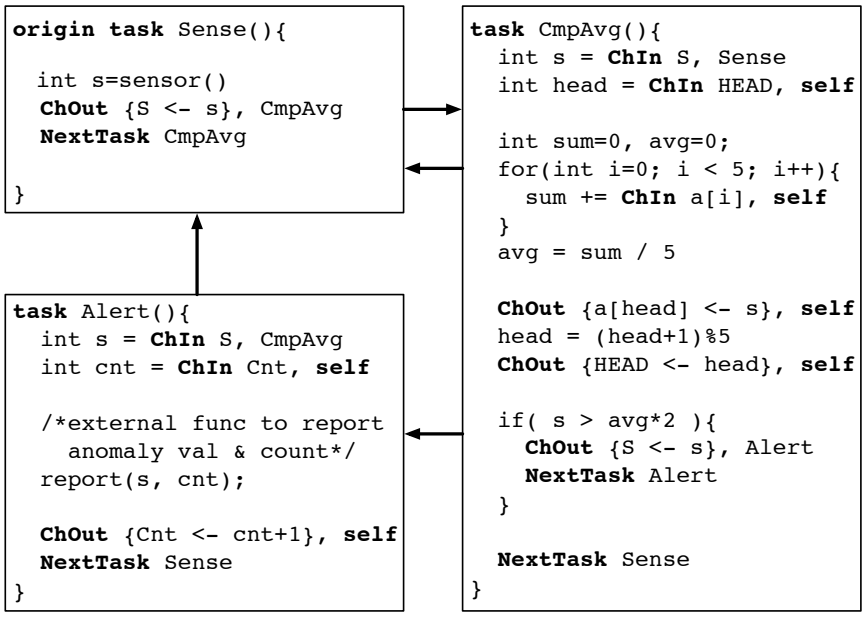


Figure 4.2: An example Chain program. The program has three tasks, `Sense`, `CmpAvg`, and `Alert`. Sense is the origin task. It reads a sensor and sends the result to `CmpAvg`. `CmpAvg` compares the current sample from `Sense` to twice the average of 5 past samples. If the current sample is greater, `CmpAvg` sends the anomaly to `Alert`. `Alert` counts and outputs anomalies. The code assumes channel fields are statically initialized to zero. All non-channel state (i.e., `int s`) in Chain is task-local.

4.1.2 Channel-based memory model

A Chain task has unrestricted access to volatile, task-local variables but is not allowed to directly access the system's non-volatile memory. Instead, Chain exposes non-volatile memory to programmers through *channels*. A channel is a named region of non-volatile memory controlled by Chain. Each channel holds a collection of individually-accessible named, typed *fields*. A channel may be declared from any task to any other task using the `channel` statement. A channel is identified by a tuple of its endpoints: the *source task* and the *destination task*. The source task can write a named value into the channel and the destination task can read that named value from the channel. We refer to the channel as a *self-channel* when the source and destination are the same task.

Channels are the sole mechanism to move data into and out of tasks. The programmer passes data through a channel using the `ChIn` and `ChOut` operations. `ChOut` takes a named

value and the name of a channel and writes the value into the matching field in the channel. `ChIn` takes the name of a channel and a field name and returns the value that was most recently written to that field in the channel by another task's `ChOut`. For example, with channel (T_1, T_2) declared, task T_1 can `ChOut` its output values for T_2 to use as inputs via `ChIn`. A self-channel (T, T) allows an instance of task T to send values to future instances of itself. Note that the programmer need only explicitly specify the destination task of a `ChOut` statement and the source task of a `ChIn`: the other task in the channel's name is the task containing the `ChIn` or `ChOut`. To refer to a self channel, the programmer can use the `self` keyword in place of a task name. Figure 4.2 shows how three tasks exchange data using `ChIn` and `ChOut` statements. `CmpAvg` and `Alert` both use `self` channels to maintain data across instances.

4.1.3 Multi-endpoint channel communication

While communication between tasks in a simple program may be expressed with basic channel operators presented in the preceding section, communication patterns in complex programs require generalized operators. Chain defines *multicast channel write* for channeling data to more than one destination and *synchronized channel read* for channeling data from more than one source. Figure 4.3(a) and (b) illustrate multicast write and synchronized read schematically.

One-to-many writes with multicast channels

Chain allows one task to produce the same value for many other tasks at the same time, using a *multicast channel write*, which is written `MultiOut` in Chain syntax. The semantics of `MultiOut` is identical to the semantics of a consecutive sequence of normal `ChOut` operations. Conversely, a `ChOut` is a special case of the generalized `MultiOut`. Including both `MultiOut` and `ChOut` in the language benefits implementations for two reasons. By retaining `ChOut` in the language, `MultiOut` becomes an optional feature that can be omitted by a compliant but

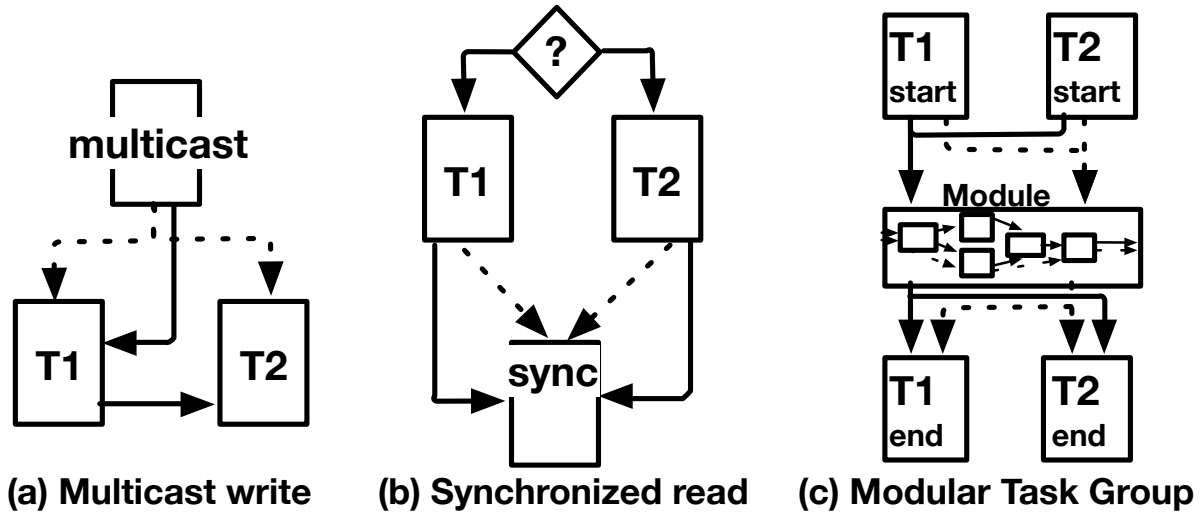


Figure 4.3: Schematic view of advanced Chain features. Dashed lines are channels and solid lines are task graph edges. (a) A one-to-many multicast channel write to T1 and T2. Multicast enables use of a single, shared channel buffer. (b) A many-to-one synchronized channel read from T1 and T2. Sync enables consuming values conditionally produced in one of many tasks. (c) A modular task group enabling reuse of an encapsulated task sub-graph. T1 and T2 can enter/exit and channel data into/out of the module, which need not refer explicitly to T1 or T2.

minimal implementation of the core language. An implementation that supports `MultiOut` can leverage it to reduce the channel memory footprint. As we discuss in detail in Section 4.2, `MultiOut` allows the Chain implementation to use a single channel buffer for the multicasted values, rather than using multiple channel buffers, one per sequential `ChOut`.

Many-to-one reads with channel sync

Chain allows one task to consume a value that may come from any one of a set of tasks using a *synchronized channel read*, which is written `ChSync` in Chain syntax. A `ChSync` takes the name of the value to read and a list of channel names, rather than a single channel name like `ChIn`. `ChSync` returns the named value from the channel that *most recently* had a value with that name written into it.

`ChSync` is an essential Chain language feature, because a value-consuming task may sometimes need a named value produced by one task and other times need the same named

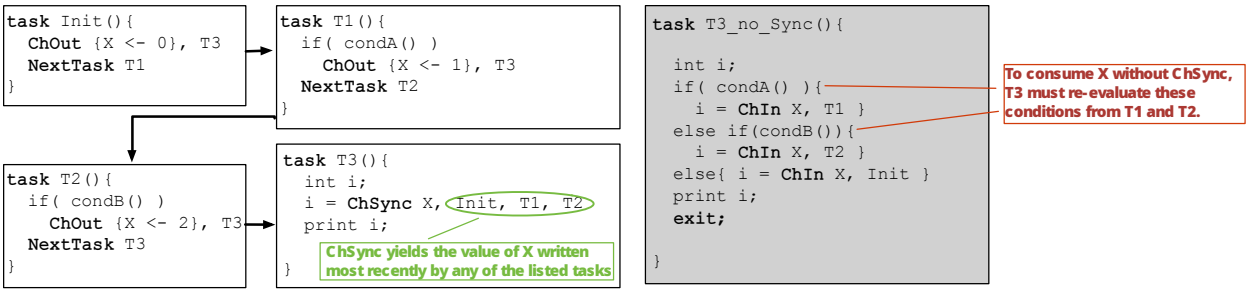


Figure 4.4: `ChSync` simplifies channel logic. Any of the tasks, `Init`, `T1`, and `T2`, may provide a value to `T3`. `T3` is trying to read the freshest value of `X`. We show two versions of `T3`, one with and one without `ChSync`. The version without `ChSync` must re-evaluate the conditions from `T1` and `T2` (in either order, assuming the conditions are exclusive), to decide which of the three task to get the value from. Such logic may require additional values referenced in the conditions to be channeled from `T1` and `T2` to `T3`. `ChSync` is a concise, robust, and efficient solution to this problem.

value produced by another task. Similarly, a value-producing task may produce a named value only if some condition is met. Figure 4.4 shows an instance of this situation. Tasks `Init`, `T1` and `T2` each potentially produce a different value for field `X`. The task consuming that named value, `T3` in this example, has no way of knowing which of the tasks, `T1` or `T2`, actually produced a value for `X` in a particular execution of the program, unless the programmer adds dedicated logic that would generate that information. The extra logic would need to either replicate the code evaluating the condition in the value consuming task (`T3`), as sketched in Figure 4.4b, or unconditionally channel the result of the condition evaluation to that task (from `T1` and `T2`). With `ChSync` such unsustainable logic duplication is avoided.

`ChSync` eliminates the need for a value-consuming task to reason about which of a set of possible value producers produced the value it needs. Instead, `ChSync` lets the consuming task observe the value most recently produced by any of the potential value-producing tasks. Yielding the most recently produced value at a `ChSync` is reasonable because there is a global, total order on tasks: yielding any other value would be contradictory to the sequential task order and would be equivalent to reordering executed tasks. Section 4.2 describes how the

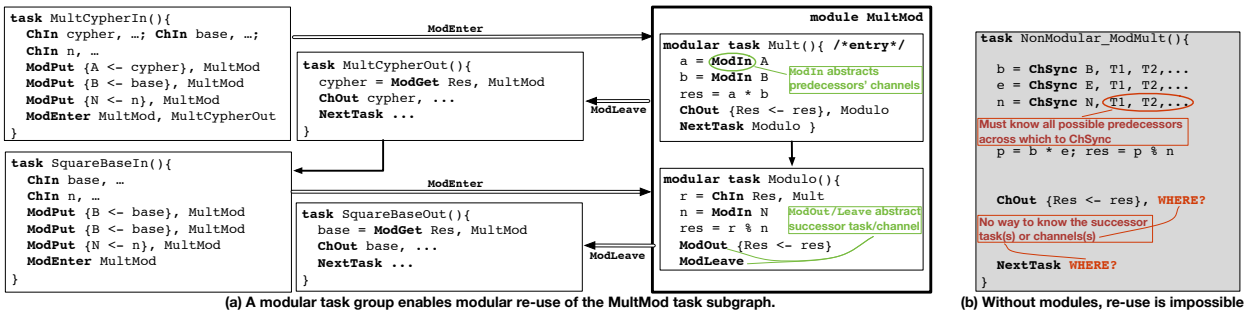


Figure 4.5: Modular task groups encapsulate code. The `RSA` implementation contains a reusable task graph that computes a product modulo a number. With modules, reused tasks need not explicitly name all predecessors (to get the inputs from) and all successors (to pass the outputs and transfer control to). `NonModular`_ModMult shows why it is impossible to properly encapsulate without Chain's modules.

Chain runtime ensures that a `ChSync` always sees the latest value.

4.1.4 Encapsulation of reusable functionality

Chain supports encapsulation of reusable functionality with *modular task groups*, or "*modules*." Figure 4.3(c) schematically illustrates Chain's module support. Like callable functions, modules allow the same code to be executed at many points in a Chain program without requiring the code inside the module to include any information about what those arbitrary points of use are. Without modules, code cannot be parameterized for reuse, because the code would need to include information about each point in the program at which its parameters would be instantiated with values. Specifically, without module support, the channel operations in the code must explicitly refer by name to each task that could produce parameter values for the code and to each task that could consume the result computed by the code. The module interface abstracts tasks that produce parameters and consume results, making it possible to write reusable code that does not contain any information about the sites where it is used.

A set of tasks may be grouped into a module using the module keyword. One task is designated as the *entry* task and one as the *exit* task. The encapsulated functionality is the

behavior of any execution from the entry task to the exit task. We refer to any task that transfers control to a module as one of the *predecessors* of that module and any task that receives control from a module as one of the *successors* of that module. To transition to a module, a predecessor task uses a `ModEnter` statement. A `ModEnter` takes the name of a module and the name of a successor task and transfers control to the module's entry task. A module's member tasks then execute until the module's exit task executes a `ModLeave` statement. The `ModLeave` statement transfers control to the successor task specified to `ModEnter`.

A module takes its input from its own input channel and produces output into its own output channel. Both channels are allocated as a result of the module's declaration and dedicated to that module. A modules's member tasks can read values from the input channel using `ModIn` and the exit task can write values into the output channel using `ModOut`. Only the exit task is allowed to write into the module's output channel, because it is never safe for more than one task to write into the same channel (cf. Section 4.1.5). For the same reason, only a predecessor task can put values into the module's input channel. A predecessor task channels values into the module using the `ModPut` statement. `ModPut` takes the name of a module and a named value, and associates the value with that name in the module's input channel. A module's successor can read a value from the module's output channel using a `ModGet` statement. A `ModGet` statement takes a module's name and a field name and returns the value from the module's output channel. To eliminate a potential source of bugs, only a successor task is allowed to access a module's output channel.

Figure 4.5 illustrates modules in a simplified snippet from our implementation of RSA. The figure shows a partial RSA task graph (left) that in two places computes a product modulo a number. The tasks inside the bold box, `Mult` and `Modulo`, compose a module called `MultMod`. `Mult` is the module's entry task. The `MultCypherIn` task `ModPut`s the two factors and the modulus into the `MultMod` module and enters the `Mult` task. `Mult` uses `ModIn` to receive the inputs. `ModIn` is key to modularity, because it eliminates the need for `Mult` to

`ChSync` across all possible predecessor tasks to find its inputs. Such a `ChSync` would break encapsulation by requiring all predecessors of the module to be explicitly enumerated. `Mult` `ChOut`s the multiplication result to `Modulo`, another member of the module, which computes the remainder. `Modulo` uses `ModOut` and `ModLeave` to yield output and transition to the module's *current* successor, which may be either `MultCypherOut` or `SqBaseOut`. `ModOut` and `ModLeave` are also essential to reusability, because they abstract the successor. These operations send output and transition to the successor registered in the module by `ModEnter`. Without modularity support, `Modulo` would be forced to break encapsulation by enumerating all successors and conditionally choosing among them based on a control value channeled into the task by its predecessor. Figure 4.5(b) shows an unsuccessful attempt to reuse code for the same calculation through complex control logic instead of Chain's modularity support.

We also note that Chain allows reusing code by encapsulating it into a conventional function and calling that function from within a task. However, this method severely limits the maximum size of a reusable component (i.e., the function). Computation that requires more energy than can be stored in the capacitor on the device cannot be encapsulated into a function, because any task that would call that function would *always* run out of energy before completing. This limit does not apply to a Chain module, because computation encapsulated into a module is decomposable into as many member tasks as necessary.

4.1.5 Correctness

Chain is correct because its execution model ensures progress and its memory access model ensures that every task is *atomic* and *idempotent*. Together these properties imply a task can arbitrarily lose power and reboot without compromising progress or consistency. Chain's progress guarantee follows trivially from the task-based execution model definition, assuming no task's energy demand exceeds the maximum energy storage of the capacitor on the device. Chain's consistency guarantee follows from the way Chain constrains accesses to non-volatile and volatile state.

Task atomicity and isolation

Channel exclusion for non-volatile memory consistency. Chain channels are strictly, statically subject to *access control*. A task may not write into any channel for which it is not the source and a task may not read from any channel for which it is not the destination. The key property that Chain guarantees about channels is *input/output channel exclusion*: by construction, a single task cannot both read and write the same non-volatile memory location. Channel exclusion occurs trivially for non-self channels. Chain statically requires that when a task executes a `ChIn` on a channel, the executing task must be the channel's destination. Likewise Chain statically requires that a task executing a `ChOut` on a channel be the channel's source. For self channels, channel exclusion is enforced by the Chain runtime (Section 4.2).

Channel exclusion guarantees that a task's accesses to non-volatile memory are *idempotent*. The task's input values are always available in channels for which it is the destination and the task cannot alter those inputs. The task's output values are always written into channels for which that task is the source and it cannot read those outputs. The inability for a task to read a non-volatile value *after* a failure that it wrote *before* a failure precludes visibility of partial or repeated non-volatile data structure updates.

Task-locality for volatile memory consistency. Chain's volatile memory model requires that all volatile variables are task-local. This ensures that a volatile variable is initialized in the task before it is used. Task-locality ensures that a task can arbitrarily reboot from its entry point without losing any volatile state: any path through the task must include a re-initialization assignment. Task-locality of the Chain volatile memory access model ensures that a task's accesses to the volatile memory are idempotent.

I/O in Chain tasks. A Chain task can interact with the world by using I/O operations to manipulate sensors and actuators. The Chain tasks with I/O behave differently from

normal Chain tasks. An I/O operation in a Chain task may execute repeatedly as a task re-executes due to intermittence. Repeated executions of an I/O operation may produce different behavior if the repeated operation is non-idempotent. Without careful programming, repeated, non-idempotent input operations can violate task atomicity and repeated, non-idempotent output operations may repeat external behavior that should not repeat.

A non-idempotent input operation violates a task's idempotence, because the input operation may produce a different value each time the task re-executes. Exposing the input operation's non-idempotence to the program is desirable, because the re-execution of an input operation (e.g., a sensor read) should always produce the latest available values. A task containing a non-idempotent input operation is safe *only if* the task does not write to a channel *conditionally*, based on the value produced by the input operation.

In the absence of an input-conditional write to a channel, every re-execution of the task performs writes to the same fields of the same channels. Note that these operations may write different *values* into these fields, depending on the result of the input operation. After the task completes, channels contain the consistent result of its last successful execution.

In contrast, a task that conditionally writes to a channel, depending on the value produced by an input operation may leave channel data inconsistent. The programmer must leverage Chain channels to eliminate this possibility. When control flow involves conditional channel output operations and a task is interrupted, it may leave its output channels in an inconsistent state that may be observed by a successor task. A program that may exhibit this behavior is illustrated on the left in Figure 4.6. Consider the execution where task `T1` reads a positive value from the sensor and is interrupted after writing `S` but before writing `SS`. Then, `T1` executes from the beginning, reads a negative value from the sensor, and transitions to `T2`. Task `T2` will observe the value of `S` written by `T1`, but the value of `SS` written by `T0`, which may be mutually inconsistent.

The problem arises because control-flow is affected by an input into the task that does not come from a channel and, therefore, is not covered by Chain's atomicity and idempotence

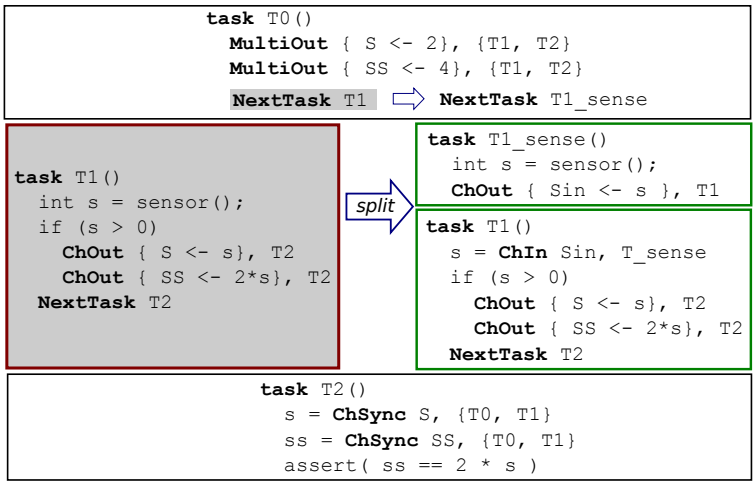


Figure 4.6: I/O in Chain tasks. On the left, in task `T1`, an input operation (`sensor()`) is followed by a conditional output into a channel to `T2`. Because the input operation is non-idempotent, the condition may evaluate differently when `T1` is interrupted and re-executed and `T2` may observe `S` written by `T1` and `SS` written by `T0`, which violates atomicity of `T1`. On the right, the program is re-written according to a programming pattern to perform I/O safely: `T1` is split such that the input operation is confined to a dedicated task (`T1_sense`).

guarantees. This observation suggests a straightforward programming pattern, which prevents I/O non-idempotence from affecting control-flow. The pattern for safe I/O is to confine the input operation to a dedicated task that acquires and channels the value to its successor(s). All downstream tasks have to obtain the sensor value from a channel, which ensures their idempotence. Applying the pattern to the above example, we obtain an equivalent program, shown on the right in Figure 4.6, which uses sensor input safely.

Once an output operation has activated an actuator, its physical effect cannot be undone. Chain does not attempt to provide at-most-once semantics for output operations. We assume that in an application destined for an environment where power is intermittent, the interfaces to actuators, the actuators themselves, or the physical world can tolerate incomplete and repeated output operations from the software.

Chain tasks are idempotent and atomic. Channel exclusion ensures that a task's non-volatile memory accesses are idempotent. Making all volatile variables task-local en-

sures that a task's volatile accesses are idempotent. Together, these facts guarantee that a task's computation and memory operations are completely idempotent. Idempotent tasks can fail repeatedly and be arbitrarily re-executed from their start without any risk of inconsistency. Furthermore, Chain guarantees forward progress at the granularity of tasks, as long as each task can complete on one capacitor charge. The progress guarantee, combined with the idempotence guarantee implies that Chain tasks are also *atomic*, exhibiting all-or-nothing behavior with respect to observable memory state. We emphasize that Chain imparts idempotence and atomicity guarantees to the application *by construction* at language level. Section 4.2 describes how our Chain implementation provides these strong guarantees without the need for costly checkpoint/restart mechanisms.

Generality of the channel-based memory model

For any program written in a sequential language with a conventional memory model, e.g. C, there is a program written using Chain task and channel abstraction that has equivalent behavior.[1] We define *program behavior* as a sequence of observed variable values, similar in spirit to the model in [103]. To simplify the discussion, we only discuss operations that manipulate non-volatile memory. This simplification is reasonable, because Chain-specific operations (`ChOut` and `ChSync`) only manipulate non-volatile memory, and there is a trivial correspondence between volatile variable manipulations in a conventional execution and in a Chain execution. Without loss of generality, we consider `ChSync`, but not `ChIn`, because `ChSync` is a generalization of `ChIn`, and `ChOut`, but not `MultiOut`, because `MultiOut` can be expressed in terms of multiple `ChOut` statements.

In a conventional execution, a memory write assigns a value to a named variable, and a memory read observes the value most recently written to a named variable. The sequence of variable values observed by the reads in an execution defines the program behavior. In a Chain execution, a `ChOut` operation assigns a value to a named field in a specified channel. A

[1] We argue about the generality of uninterrupted execution, because a conventional program does not complete in the presence of intermittence.

`ChSync` operation observes the most recent value of a named field in a collection of specified channels. The sequence of variable values observed by the `ChSync` operations in an execution defines the program behavior. We show that for all conventional executions, there exists a Chain execution that has the same behavior. We start from an arbitrary execution trace generated by an arbitrary conventional program and we construct a Chain execution trace that has the same behavior.

We consider an arbitrary conventional execution trace, E_{conv}, defined by a sequence of operations, $\mathsf{write}_i(M[v], x)$ and $\mathsf{read}_j(M[v])$, where indexes i and j denote positions in the sequence and the operations are respectively a write of value x to memory location v and a read from memory location v. We begin constructing a Chain execution trace, E_{Chain}, by initializing it with a copy of E_{conv}. At this point, E_{Chain} is not yet a valid Chain execution trace, because it contains direct accesses to non-volatile memory. We assume an arbitrary assignment of operations to tasks in the Chain execution and define a convenience function, $\mathsf{Task}(c)$, that reports the task containing operation c. In E_{Chain}, we replace each $\mathsf{write}_i(M[v], x)$ with a `ChOut` of value x to field $F[v]$ from task $\mathsf{Task}(\mathsf{write}_i)$ to each subsequent task that reads that value before it is overwritten by another, later write. That is, the `ChOut` associated with write_i accesses channels

$$\{(\mathsf{Task}(\mathsf{write}_i), \mathsf{Task}(\mathsf{read}_j)) \mid j > i \text{ and } \nexists \mathsf{write}_k,\ i < k < j\}.$$

Similarly, we replace each $\mathsf{read}_j(M[v])$ with a `ChSync` of field $F[v]$ from each preceding task that wrote to that field. That is, the `ChSync` associated with read_j accesses channels

$$\{(\mathsf{Task}(\mathsf{write}_i), \mathsf{Task}(\mathsf{read}_j)) \mid i < j \text{ and } \exists \mathsf{write}_i\}.$$

Note that in the case of `ChSync`, we consider *all* prior writes, because, as we note in Section 4.1.3, a task that reads a field using `ChSync` cannot know which `ChOut` had produced the observed value and must rely on `ChSync` to retrieve the most recent value. We next argue that the constructed E_{Chain} has the same behavior as E_{conv}.

To show behavioral equivalence, we show that the sequence of variable values produced by the memory read operations in E_{conv} is the same as that produced by the `ChSync` operations in E_{Chain}. A $\mathsf{read}_j(M[v])$ in E_{conv} observes value x written by the *unique* $\mathsf{write}_i(M[v], x)$ that most recently precedes that read — i.e. $i < j$ and $\nexists \mathsf{write}_k(M[v], y)$ for $i < k < j$. The write is unique, because a conventional execution is sequentially totally ordered. In the constructed E_{Chain}, each `ChSync` accesses a *collection* of channels, each of which may contain a value for field $F[v]$. As observed by the `ChSync`, the channels are not sequentially ordered, unlike writes with reads in E_{conv}. However, by the semantics defined in Section 4.1.3, a `ChSync` on field $F[v]$ observes the value of the `ChOut` to $F[v]$ that was the *most recent* according to an explicitly maintained timestamp. By our construction, this `ChOut` operation corresponds to $\mathsf{write}_i(M[v], x)$ in E_{conv}, where $i = \max\{k \mid k < j \text{ and } \exists \mathsf{write}_k(M[v], \cdot)\}$. This correspondence implies that the value of the `ChOut` operation to field $F[v]$ is also x. Extending this argument to all read operations in E_{conv} and the corresponding `ChSync` operations in E_{Chain}, we conclude that the sequence of variable values produced by the execution traces is identical, implying that the behavior of programs producing these traces is equivalent.

4.2 Implementation

We implemented the Chain programming language primitives described in Section 4.1 using a combination of compile-time macros and a runtime library. The compile-time features declare and allocate memory for tasks and channels. The runtime features implement task sequencing and channel operations. Figure 4.7 depicts the state that Chain uses internally to implement execution context, tasks, and channels. Our implementation of Chain is distributed as a static library with headers. It amounts to 644 lines of C code, which compiles to 412 bytes or 0.6% of program memory on the WISP platform.

Hardware Assumptions. Our implementation makes few assumptions about the underlying energy-harvesting hardware. We assume some (but not necessarily all) memory is

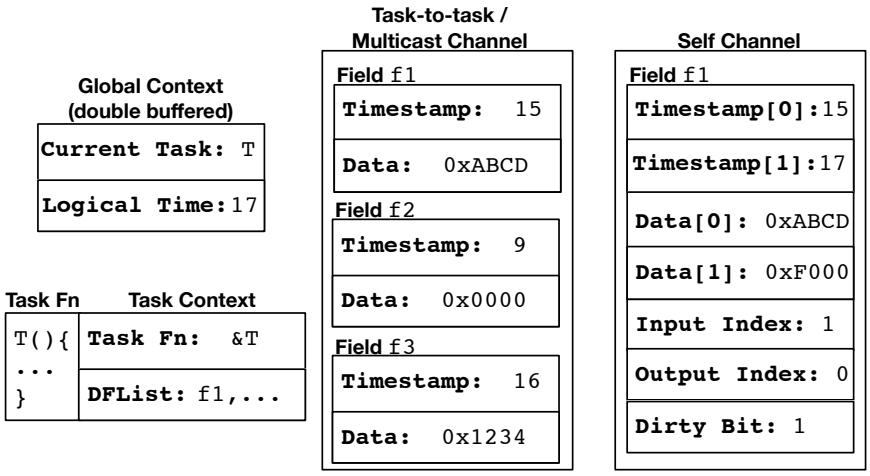


Figure 4.7: State used in our Chain implementation. The double-buffered execution context tracks time and the current task. The task context keeps a pointer to the task code and a "Dirty Field List" (DFList) containing updated fields in the task's self channel. Task-to-task channels and multicast channels have the same representation and each of their fields contains a timestamp and data. A self channel field contains two timestamps and data buffers, one for input and one for output. A self channel field tracks which timestamp and data buffer is input and which is output using the input/output indices; the dirty bit is set if the field was updated.

non-volatile. This assumption matches existing energy-harvesting devices (e.g., the Wireless Identification and Sensing Platform (WISP) [149]). Chain runtime implementation assumes single-word writes to non-volatile memory are atomic. This assumption is reasonable for the single-cycle 16-bit microcontrollers and the FRAM memory technology common in energy-harvesting hardware. While atomicity of memory writes given arbitrarily-timed power failures is not explicitly guaranteed by manufacturers, we have never observed partially written words in non-volatile memory. For the runtime library, the compiler must not be allowed to re-order writes to non-volatile memory. This requirement does not concern application code, because all Chain operations are sequencing points.

4.2.1 Tasks

A task is composed of a *task function* that contains its code and a *task context object* that contains its runtime state. A task function is a C function with no arguments and no return

value. A task function can contain calls to arbitrary C code (i.e., legacy/third-party code), but Chain's consistency guarantee does not extend to any such code that writes to non-volatile memory. A task's context consists of a pointer to its function and state related to maintaining its self channel, which we describe in Section 4.2.3. Each task object is statically allocated and initialized in non-volatile memory.

4.2.2 Task sequencing

The Chain runtime maintains a non-volatile *global execution context* that stores the pointer to the current *task execution context* and the current logical time; both objects are depicted on the left in Figure 4.7. The `NextTask` control-flow directive updates the current task context pointer in the global execution context to point to the context object of the next task. Chain must atomically update the multi-word global execution context despite intermittence. Atomicity is ensured by *double buffering* the global execution context and indirecting accesses to it through a pointer. While the current global execution context is in one buffer, the `NextTask` routine sets up the updated context in the other buffer. To commit the transition, Chain sets the context pointer to point to the updated buffer atomically using a single instruction. After a reboot, the runtime transfers control to the task pointed to by the current global execution context, which is retrieved from non-volatile memory. On each task transition, but not on reboot, the runtime increments the current *logical time* in the global execution context, which clocks application progress and is used to implement channel operations described in the next section.

A key property provided by the Chain language implementation is that all state visible to a program after a task transition is *exactly the same* as after a reboot. This property frees Chain from the need for costly restore operations after reboots that are characteristic of checkpointing systems. After a transition, the Chain runtime invokes a *task prologue* that idempotently sets up a task's channel structures. Section 4.2.3 provides a detailed explanation of channel setup in the prologue. Importantly, Chain guarantees that the prologue

completes exactly once for each task transition. To guarantee a single successful prologue execution the runtime saves the logical time at the end of the prologue into the task context. Chain only executes the prologue if the current time exceeds the saved timestamp. After the prologue, Chain jumps to the task's function entry point.

4.2.3 Channels

In our implementation of Chain, channels are defined statically and accessed dynamically. A channel is defined by specifying its two end-points and a set of named, typed data fields. We refer to a channel between different tasks as a *task-to-task* channel to distinguish it from a *self* channel (Section 4.1.2). A channel's data field may hold a scalar value or an array value. Chain implements array fields as a collection of scalar fields, each of which can be referred to by an index. The fields of each channel are specified by the programmer at compile time using syntax defined in the Chain library header.

Each channel definition translates to a C structure type. Internally, each member field of a channel is a nested structure. The nested field structure in a task-to-task channel has a buffer to hold the data value of the channel field and a member for channel metadata. The channel metadata field consists of a last-modified timestamp, which is used by the `ChSync` implementation described in Section 4.2.6.

The implementation of a self-channel is different from a task-to-task channel. A self-channel field has *two* buffers for data values — one for incoming and one for outgoing data — and a member for metadata. The duplicated data buffers in a self-channel field are used to implement the channel exclusion principle introduced in Section 4.1.5. The metadata field of a self-channel holds the timestamp of its last update and the state Chain needs to decide which data buffer is its input and which is its output (Section 4.2.4).

A channel declaration statically allocates a channel as a non-volatile C `struct` (in FRAM). A channel's symbol name is the concatenation of the names of the channel end point tasks. Consequently, `ChIn` and `ChOut` can resolve a channel's name using tasks' names

at compile time.

4.2.4 `ChIn` and `ChOut`

A task writes a value into a field of a channel using `ChIn` and reads a value from a field of a channel using `ChOut`. These directives resolve the channel's memory location at compile time by concatenating the names of the source and destination tasks into the channel structure's symbol name. A `ChOut` to a field in a task-to-task channel writes a value into the field's data buffer and sets the field's last modified timestamp to the current logical time (from the global execution context). A `ChIn` from a task-to-task channel's field returns the value in the field's data buffer.

A self-channel field has an input and an output data buffer. Its field metadata consists of two timestamps, an *output buffer index*, an *input buffer index*, and a *dirty bit*. A `ChOut` to a field of a self-channel writes the given value to the data buffer identified by the output index, sets the dirty bit, and adds the field offset to a *list of dirty fields* in the task object. A `ChIn` from a self-channel returns the contents of the data value buffer identified by the input index. On the next transition the roles of the input and output value locations are reversed for all fields that were marked as dirty. This takes place in the prologue routine that runs once after a task transition, as explained in Section 4.2.2. For each field in the dirty field list with its dirty bit set, the prologue does an atomic *swap-and-clear* that *swaps* the input index with the output index and *clears* the dirty bit. We pack the index and dirty bits into a single 16-bit word, making the atomic *swap-and-clear* a single write instruction. Even if the prologue executes repeatedly, each field undergoes exactly one swap, because a swap only occurs if the dirty bit is set and the dirty bit is cleared by the swap.

4.2.5 `MultiOut`

With a `MultiOut` primitive a task can channel a value to multiple recipients using as much memory as a single task-to-task channel. A *multicast channel* is like a task-to-task channel

in its compile-time declaration and field structure. The channel's name is the concatenation of its source with a *destination ID* that uniquely identifies its destination list.

A `MultiOut` statement can refer to a multicast channel only if its source task is the calling task, because `MultiOut` constructs the channel's name using the name of the calling task. The `ChIn` and `ChSync` statements use the name of the calling task and the multicast channel's destination ID string to refer to the channel. Our prototype does not prohibit reads from a multicast channel by tasks that are not members of the destination set. This limitation may be unintuitive, but does not jeopardize Chain's correctness.

4.2.6 `ChSync`

A `ChSync` operation reads a value that may reside in one of a set of channels. The `ChSync` primitive accepts a field name and a set of sources. A source can be a task name, `self`, or a multicast destination ID. At compile time `ChSync` resolves each source into a channel name and locates the field's last-modified timestamp by the field's name. At runtime, `ChSync` compares the timestamps associated with the fields and returns the data value of the field with the latest timestamp.

4.2.7 Modular task groups

A *modular task group*, or a "*module*", encapsulates a group of tasks for re-use. A module contains an entry task, an exit task, an input channel, and an output channel. A module's input and output channels are implemented as augmented task-to-task channels. The input channel stores the name of the module's successor task. `ModEnter` saves the name of the successor task into the input channel and transfers control to the entry task. A module's entry task's name is constructed at compile time from the module's name. `ModPut` and `ModIn` translate into `ChOut` and `ChIn` on the module's input channel. `ModGet` and `ModOut` translate into `ChIn` and `ChOut` on the module's output channel.

Any task in the module can `ModIn` from the input channel, because the input channel symbol name does not include the calling task's name. The output channel's name includes the name of the exit task, allowing only the exit task to `ModOut` to the output channel. Our prototype implementation of modules has a limitation that is not fundamental to Chain's design. We do not check that only a module's member tasks access its channels, and the programmer is responsible for correctly using `ModIn`, `ModOut`, `ModPut`, `ModEnter`, and `ModGet`.

4.3 Evaluation

We compare Chain to state-of-the-art runtime systems in terms of correctness, performance, memory profile, and developer effort. We deployed each application described in the next section on the WISP [149] and ran it on harvested-energy.

4.3.1 Experimental setup and applications

We used Chain to implement four applications, each representative of a practical domain with varied control flow, compound data types, and complex data structures. We built our systems using the WISP5 energy-harvesting platform, which has a TI MSP430FR5969 16-bit MCU with one core clocked at 8 MHz, 2KB of RAM, 64KB of non-volatile FRAM, an accelerometer, and an RF energy-harvesting power system [149]. In addition to implementing the applications using Chain, we also implemented each using DINO [101] and Mementos [101], which are state-of-the-art runtime systems for intermittence. We used the publicly released DINO implementation. We wrote two variants for Mementos. One variant, Mem-NV, uses volatile and non-volatile memory, but may experience data corruption because Mementos does not keep non-volatile memory consistent. The other variant, Mem-V, restricts mutable state to volatile memory, which is kept consistent by Mementos, but limits the total size of the program state to the small capacity of the volatile memory.

Activity Recognition (AR). AR is a machine-learning physical activity classification

system used in prior work [101]. AR collects accelerometer samples into a sliding window and filters out samples below a noise threshold. AR converts the 3-axis samples into feature vectors and classifies the window as moving or stationary using a nearest neighbor classifier. After classifying, AR updates the classification statistics for each class and stores them in non-volatile memory for later inspection. AR trains its model by having the user generate reference activity for each class. In a correct execution, the classification statistics must be mutually consistent: the class counts must sum to the total count.

Cold-Chain Equipment Monitoring (CEM). A CEM system continuously monitors a temperature-controlled environment (e.g., vaccine storage), logging temperature over time. Our CEM system collects a stream of temperature sensor readings and compresses them using LZW compression [185] to maximize the capacity of the log. The compressed stream is recorded in non-volatile memory for later decompression and inspection. In a correct execution, the resulting log is a valid, LZW-compressed data stream.

Data Encryption (RSA). This application encrypts a message using RSA [142]. The public key of up to 2048 bits (configurable at compile time) is stored in non-volatile memory, and can be changed after deployment. To the best of our knowledge, ours is the first RSA implementation on an energy-harvesting device using such a strong (large) key. Our implementation thus enables an energy-harvesting device to securely communicate with any base station without the need to share a secret ahead of time.

Cuckoo Filtering (CF). A cuckoo filter is a general-purpose data structure that approximately encodes set membership and supports element deletion. This data structure is well-suited for filtering out redundant samples from a sensor. Like a Bloom filter, a cuckoo filter may return a false positive but not a false negative when queried for a value. Our CF implementation inserts a fixed-length sequence of pseudo-random values into a large filter. CF then looks up the sequence of values in the filter. In a correct execution, the count of affirmative lookups matches the sequence length.

We used the following experimental setup for each application. The input to AR was

generated by the accelerometer as we flipped the orientation of the WISP from vertical to horizontal throughout the experiments. The AR model was trained in the vertical and horizontal orientation for each class, respectively. The input to CEM originated from the temperature sensor without any deliberate manipulation of the surrounding temperature. The plaintext for RSA was a fixed 11-byte string stored in non-volatile memory. The input values in CF were produced by a simple pseudo-random-number generator with a fixed seed. We used the Energy-interference-free Debugger (EDB) [38] to record the output without affecting the energy state of the device. In our lab setup the WISP harvested energy from a ThingMagic Astra-EX RFID reader from a distance of 20 cm (10 cm for CEM).

4.3.2 Correctness

An application may produce an incorrect result on an intermittently-powered platform if the runtime system does not guarantee memory consistency. Table 4.2 summarizes the outcome of running each application on harvested energy. Applications written in Chain and DINO always produced correct output. This result follows from the memory consistency guarantee made by these systems.

The Mem-NV version of every application either returned an incorrect output or failed to complete on at least one trial. AR generated percentages for moving and stationary classes that did not add up to 100%. CEM entered an infinite loop or produced compressed text with more compressed indexes than uncompressed samples, which violates an invariant in LZW algorithm. RSA produced undecryptable cyphertext. CF reported false negatives to membership queries. Mem-V guarantees correctness for an application only if its state (stack and global variables) completely fits within *volatile* memory. We discuss the implications of restricting state to volatile memory in Section 4.3.4.

App.	Chain	DINO	Mem-NV	Mem-V
AR	✓	✓	✗	∗
CEM	✓	✓	✗	∗
RSA	✓	✓	✗	∗
CF	✓	✓	✗	∗

Table 4.2: Correctness of observed application output. Legend: ✓= correct, ✗= incorrect, ∗= correct if application fits in RAM.

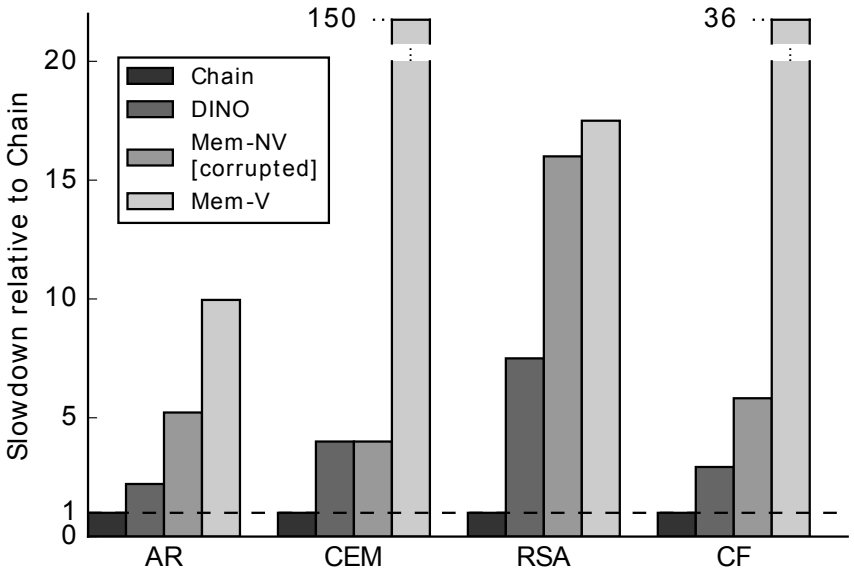


Figure 4.8: Application performance with Chain and state-of-the-art.

4.3.3 Performance

We ran each application in Section 4.3.1 on harvested energy and measured the time it took to complete a fixed amount of work. The amount of work was defined by application-specific parameters that control the number of classifications in AR, the size of the compressed log in CEM, the size of the plaintext (equal to key size) in RSA, and the number of buckets in the cuckoo filter in CF. We configured AR to 128 classifications, CEM to a dictionary of 280 entries, RSA to 128-bit keys, and CF to a filter with 256 buckets. These are the largest workloads that Mem-V can handle due to its memory size limitation. In our trials each application completed its workload within several seconds. We present detailed results from a representative trial run in Figure 4.8.

App.	Memory Consumption (KB)				Multicast Benefit		
	Chain	MemV	MemNV	DINO	Ops.	Avg. Dests.	Savings
AR	2.5	4.1	4.2	4.2	1	2	50%
CEM	9.4	4.1	5.8	5.8	5	2.2	37%
RSA	4.9	4.2	4.4	4.4	11	3.6	90%
CF	16.2	4.1	4.6	4.6	5	3.4	73%

Table 4.3: Non-volatile memory usage (KB) with Chain and state-of-the-art, measured when deployed on TI MSP430FR5949 MCU that features 2 KB of SRAM (volatile) and 64 KB of FRAM (non-volatile). Right-hand columns show the benefit of Chain's channel multicast feature: the number of multicast operations used, the average number of destinations, and the memory saved on multicast channels relative to an equivalent set of per-task channels with `ChOut` statements.

Figure 4.8 shows the slowdown relative to Chain, defined as a ratio of the respective execution times. Chain outperforms DINO, the state-of-the-art, by 2x to 7.6x. We include Mem-NV performance results for completeness, but emphasize that its output is *incorrect* in trials that generate any output at all (cf. Section 4.3.2). Mem-V running time is one or more orders of magnitude above the alternatives. Mem-V spends most of the time in saving and restoring its disproportionately large checkpoints, each of which must include *all* program state. Applications run faster with Chain, because Chain does not use checkpoints. Chain eliminates the cost of saving and restoring checkpoints as well as the work wasted on checkpoints that are started without sufficient energy to complete them.

4.3.4 Memory profile

We characterized how Chain utilizes memory and showed that Chain is always comparable with DINO and Mementos, and in some cases Chain makes better use of memory. The memory footprint of the runtime is not prohibitively high for any of DINO, Mementos, or Chain. Chain's footprint was 412 bytes (0.6% of memory on the WISP), compared to 582 bytes for DINO (30% larger than Chain) and 340 bytes for Mementos (21% smaller than Chain). The most significant memory cost for all three systems is the non-volatile memory consumed by checkpoints and channels. Table 4.3 summarizes the non-volatile memory footprint that we collected from the application binaries. For Chain, the footprint consists

of the channel memory buffers. For checkpointing-based systems (Mem-V, Mem-NV, and DINO) the footprint includes the space reserved for a *double-buffered* checkpoint, as well as non-volatile variables in Mem-NV and their versioned copies in DINO. Mementos and DINO both conservatively use a checkpoint buffer that can accommodate a checkpoint of all of RAM (i.e., 2KB), amounting to double-buffered checkpoint storage of 4KB.

The data show that Chain's memory use is sometimes less than, and sometimes more than that of the other systems for the same applications. Chain uses less memory than checkpointing systems when the size of channel state is less than the size of checkpoints and versions. Chain allocates exactly as much memory as it needs, since its channel declarations are available at compile time. In cases where Chain uses more non-volatile memory (CF, CEM), the overhead is due to replication of data in channels. In these cases, Chain trades non-volatile memory consumption for the often disproportionately large improvement in throughput and energy efficiency that comes with eliminating checkpoints.

We also analyzed volatile memory consumption. Volatile and non-volatile memory usage affect deployment cost disproportionately. Non-volatile memory is orders of magnitude cheaper than volatile memory, e.g., the largest MCU in the MSP430FR family has 128 KB of FRAM but only 2KB of SRAM. Chain does not change a system's volatile memory consumption. DINO requires enough volatile memory to hold versioning data for non-volatile variables on the stack before adding them to a checkpoint. Mem-V requires enough volatile memory to hold all program state, making it impossible to use for non-trivial applications on some MCUs.

Mem-V's exclusive dependence on volatile memory compromises application performance, constrains the choice of MCU, and limits the maximum distance to the energy-source. For example, in CEM the compression rate is a function of the size of LZW dictionary. The 2048-bit RSA would require an MCU with at least 2KB of SRAM. However, purchasing more RAM is not a solution if the RAM contents is part of every checkpoint, as it is in Mem-V. Once the checkpoint becomes large enough, the energy required to reach the next

App.	Chain					DINO	
	Tsk. / Mod.	LOC				Tsk. Bnd.	LOC
		Decl.	Flow	Chan.	Tot.		
AR	11 / 0	61	19	49	519	8	435
CEM	12 / 0	82	19	63	412	13	264
RSA	20 / 2	103	28	119	831	34	644
CF	14 / 1	109	20	74	432	13	262

Table 4.4: Size of application implementations in Chain and DINO.

checkpoint will exceed the energy available during one capacitor charge-discharge cycle and application progress will halt. We directly observed this failure mode for our CEM system at >10 cm from its energy source, where Mem-V stopped working completely.

To illustrate the value of Chain's multicast channel feature, we calculated the memory saved in Chain applications by using multicast channels instead of standard channels. As explained in Section 4.2.5, our implementation of the multicast channel shares one memory buffer between all destination endpoints. Table 4.3 shows that using multicast channels is valuable in all of our applications and using multicast channels on average consumes less than half as much memory as a collection of standard channels would use to serve the same purpose.

4.3.5 Developer Effort

To implement an application in Chain, the developer decomposes it into tasks and connects the tasks with control flow statements and channel statements. A decomposition follows naturally from a modular application design. A loop may need to be converted into a task with the loop body and a transition to itself. To reuse code in a decomposed application, with moderate effort tasks can be encapsulated into modules (cf. Section 4.1.4). Decomposing into Chain tasks is similar to placing DINO boundaries. We list the number of tasks and modules in Chain and task boundaries in DINO for each application in Table 4.4.

We compare the amount of additional code each application requires in both Chain and DINO implementations in Table 4.4. For Chain, lines are categorized into channel

declarations, task transition statements, and writes to and reads from channels. Across our applications Chain code is larger than DINO code on average by 42%, of which 60% are straightforward declarations of tasks and channel fields. A task declaration specifies the name of the task and its implementation function, and a channel declaration specifies the channel's endpoints and the types of each of its fields.

Although they burden the programmer, explicit specifications of channels are self-documenting and provide the necessary information to statically check the usage of `ChIn`/`ChOut` statements for correctness. An implementation alternative could trade off the advantage of explicit specifications in favor of reducing the amount of code required. Such an implementation could infer channel declarations from their usage in `ChIn`/`ChOut` statements. A graph of inter-task data exchange could then be constructed from the `ChIn`/`ChOut` statements, and a channel allocated per each edge. The type of channel fields can be inferred from the type of values that are being written and read from the channel, with the exception of sizes of array fields, which would be specified by explicit type declarations. Channel field, type, and structure inference is an especially compelling direction for future work on Chain.

4.4 Summary

Chain is the first programming model to provide intermittence-safety without the need for costly checkpoints. Chain provides a task-granular progress guarantee and its channel memory model keeps data consistent. Channels dispatch with the need to save and restore checkpoints on reboots. Chain ensures consistency and progress with 2-7x higher throughput than prior systems. Such a throughput increase *enables* compute-intensive applications that demand correctness, like 1024-bit RSA and LZW compression, on energy-harvesting devices.

Chapter 5

CleanCut: Task Decomposition for Intermittent Programs

A task-based intermittent programming models like Chain (cf. Chapter 4) and checkpoint-based systems like DINO [101] asks the programmer to decompose the program into smaller regions, which we will refer to as *tasks* for both classes of systems. After a power failure, a task re-executes from the beginning. However, the task has a chance to finish executing only if the energy storage capacitor can buffer sufficient energy to sustain the workload of that task. Whether a task completes depends only on the storable energy and not on the energy harvestable from the environment, because the energy incoming during the (brief) time that the capacitor takes to discharge is negligibly small for weak harvesters.

When a system delegates the decomposition of the program into tasks, i.e. the placement of task boundaries into the program code, to the programmer, it burdens the programmer with the energy estimation challenge. Reliance on the human programmer introduces the risk that tasks will either incur a high overhead or be impossible to complete. An overly cautious programmer may place more task boundaries in code than necessary, wasting energy and imposing a time overhead. If the programmer uses too few boundaries, the program may have a *non-terminating path* that requires more energy than the device can buffer. A

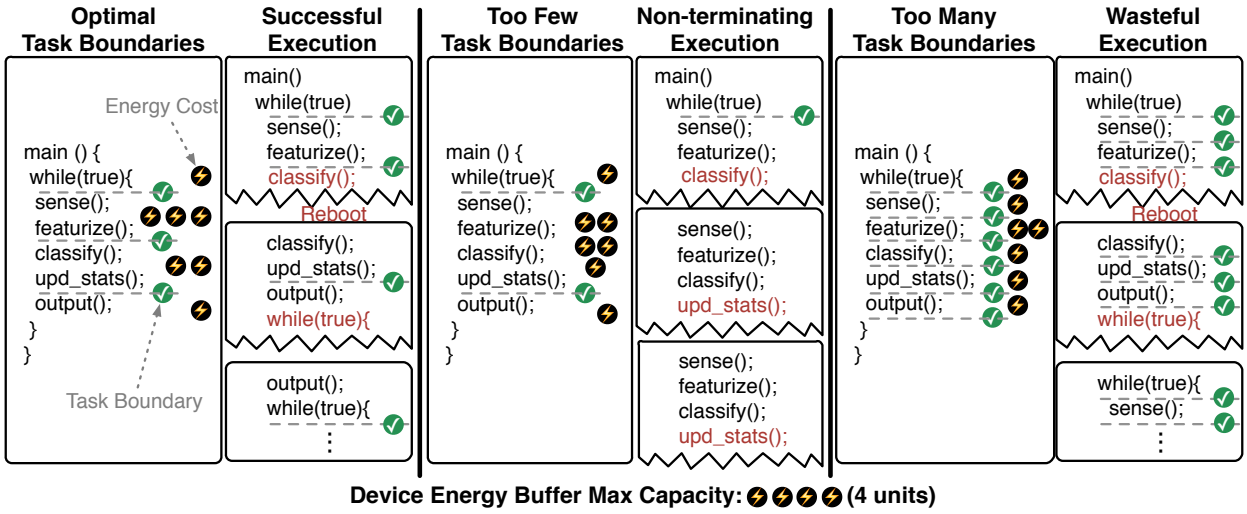


Figure 5.1: Different task decompositions cause different execution behavior.

non-terminating path consumes more energy than will ever be available, causing the task to repeatedly restart and fail forever. Code including such a non-terminating path represents a new type of software bug that is unique to intermittent applications. There is currently no system support to help find these bugs by assessing whether a task boundary placement includes non-terminating paths, nor for helping place task boundaries.

Figure 5.1 shows how different static task boundary assignments lead to different behavior with three variants of an activity recognition application from prior work [101, 39]. The code featurizes and classifies data from a sensor, maintains statistics, and produces output. The energy cost of a task is illustrated in terms of abstract energy units, represented by the lightning bolt circles between the task's initial and terminal boundary. The figure assumes a device that can buffer at most four energy units. The left variant of the program is decomposed into tasks using three boundaries. The energy consumption of the resulting tasks does not exceed the device's energy capacity and the depicted execution makes progress with little boundary overhead, despite periodic reboots. The middle variant has a non-terminating path bug because it is decomposed with *too few* boundaries. This variant's most costly task consumes more energy than the device can buffer. Consequently, the application can never make progress, rebooting and re-executing the task indefinitely. The right variant

is inefficient because it uses *too many* task boundaries because the energy cost of each task fits within the device capacity and boundaries execute more often than necessary, wasting time and energy.

Despite the importance of placing task boundaries, doing so remains a difficult, manual process for which there is no system support, regardless of the system for intermittent execution. There is no direct correspondence between a code span and its energy cost, because energy consumption varies per execution and depends on the *full system*, including peripherals. Code paths have a *distribution* of energy costs that is opaque to the programmer. There is no connection between code and the energy capacity of the device. Programmers get no feedback from the compiler about their task boundary placement. Instead, the programmer is left to guess whether tasks will terminate, or if excessive boundary overhead will throttle throughput. To port to another platform, the programmer must decompose the code again. Adoption of task-based intermittence models is impeded by the lack of support to *check* that a decomposed program is free of non-terminating path bugs and to *place* task boundaries to avoid these bugs by construction. Our objective is to fill both of these gaps.

In this chapter we propose CleanCut, a system for *finding* non-terminating paths in intermittent programs and *eliminating* such paths by generating terminating task boundary assignments automatically. CleanCut's *checker* checks a task boundary assignment and reports non-terminating paths that need refinement. CleanCut's *placer* subdivides a program into tasks free of non-terminating paths. CleanCut minimizes overhead by approximately bisecting paths and preferring boundaries unlikely to be executed frequently.

Both the checker and placer use CleanCut's statistical model of the energy consumption of each program path. The energy cost of a path is calculated from the energy cost of basic blocks that compose that path, including paths with loops bounded by annotations from the programmer. CleanCut's path energy model is compatible with block energy models based on direct measurement and analytical models [99].

We implemented CleanCut's analyses in LLVM and used it to analyze applications from

prior work [101, 39, 106]. We show that CleanCut's checker identifies task boundary assignments with non-terminating path bugs, demonstrating its value as a debugging tool. We show that CleanCut's placer produces boundary placements that are free of non-terminating paths and have lower overhead than manually- or randomly-placed boundaries. We measure the additional compilation time taken by CleanCut to confirm its practicality.

In the next section we give a system-wide overview of CleanCut. Sections 5.2—5.4 describe CleanCut's energy model, checker, and placer. Section 5.5 provides implementation details. Section 5.6 evaluates CleanCut. We summarize in Section 5.7.

5.1 Overview

CleanCut is both a debugging tool and a program transformation analysis that helps a programmer place task boundaries in a program written for a task-based intermittent execution model to avoid non-terminating path bugs. CleanCut has two modes of use, as a *checker* or as a *placer*. Both the checker and the placer rely on CleanCut's energy model, which is described in detail in Section 5.2. Sections 5.3 and 5.4 describe the details of the checker and placer themselves.

CleanCut's checker is a *debugging tool* that examines a task boundary placement and checks for non-terminating path bugs. A non-terminating path bug stems from a misuse of task boundaries that allows a path through a task to consume more energy than the maximum amount of energy that the target device can buffer. The program's source and the energy buffering capacity of the target device are inputs to the checker. If the checker finds a path that consumes more energy than the device can buffer (i.e., a non-terminating path bug), the checker reports the path to the programmer, along with the boundaries of the task containing the non-terminating path. The programmer can then adjust the task boundaries — by moving existing boundaries or adding new ones — to eliminate the bug. The programmer can re-run the checker after each adjustment to the task boundaries

eventually eliminating all reported non-terminating path bugs. The checker is most useful to a programmer that prefers fine-grained manual control over boundaries to ensure that, for example, related I/O operations execute in the same task.

CleanCut's placer is a *program transformation* that adds task boundaries to a program to avoid non-terminating path bugs. The goal of the placer is to produce a task boundary assignment that is free of non-terminating paths and that minimizes the overhead of executing task boundaries. The placer works iteratively and each iteration evaluates the current task boundary assignment to identify non-terminating paths. The placer selects the non-terminating path of highest energy cost to subdivide, and inserts a new task boundary along the path to divide the path into two sub-paths of approximately equal energy cost. To minimize task boundary overhead, the placer avoids placing boundaries in loops with a high iteration count and in functions that are called from many call sites. The placer is most useful to a programmer that has fewer platform-specific requirements in their application, and benefits more from a fully-automated workflow.

5.2 Energy model

CleanCut's non-termination bug checker and task boundary placer both rely on a statistical model of the energy consumed by each control-flow path from one task boundary to another. The model computes a path's energy by combining the energy of its constituent basic blocks. We chose to model path energy based on basic block energy, as opposed to single instruction energy, to produce estimates closer to the observable average case rather than the theoretical worst-case, following the insights in [99]. In addition, since profiling is part of the programmer's workflow in CleanCut, we avoid relying on high-resolution measurement hardware to collect per-instruction estimates. With a block-based model, as with a *detailed* instruction-level model, energy estimates must be recomputed as code changes.

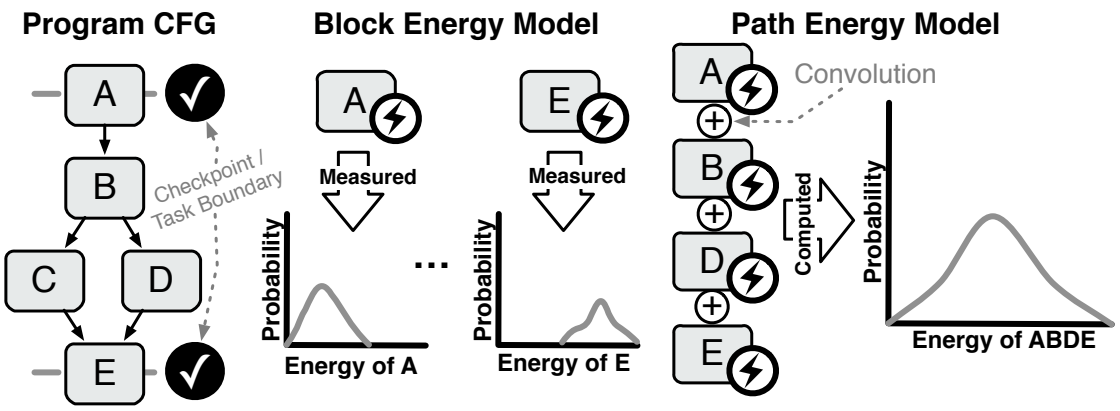


Figure 5.2: CleanCut models the energy of each path.

5.2.1 Block energy model

A basic block energy model compatible with CleanCut represents the consumed energy as a *probability distribution* that indicates how likely the block is to consume different amounts of energy. A distributional model captures the range of possible energy costs of a block, which is necessary to estimate the probability of a non-termination bug manifesting. Figure 5.2 on the left illustrates that CleanCut measures an energy distribution for each block in a program's control-flow graph, using the procedure described in Section 5.5.2.

We chose to make CleanCut's energy model *distributional* to avoid losing information about possible path energy consumptions. A distributional model captures more information than a scalar worst-case energy model about whether a non-termination bug will manifest. A distributional model lets CleanCut's non-termination bug checker report the likelihood that a path will not terminate to the programmer. Reporting non-terminating path bugs with their manifestation likelihood enables the programmer to prioritize potential non-termination issues.

CleanCut is designed to accept any distributional block energy model that can represent the distribution as a discrete histogram. Our prototype implementation of CleanCut uses a measurement-based block energy model, because it accounts for the total energy of the board, including sensors and radios, does not rely on any models of low-level circuit power

behavior, and was effective in our evaluation (Section 5.6). The potential drawback of this measurement-based prototype is that it may not capture all of a block's energy behaviors, potentially underestimating the block's worst case energy as the maximum energy *observed* during measurement.

The potential drawbacks of the measurement-based model in our prototype are *not* inherent to CleanCut's path modeling approach, however, and CleanCut could instead use an analytical block model derived from device characteristics and application analysis [34, 99, 90, 188]. Using an analytical model has the advantage of being able to estimate *theoretical* worst-case energy, and can provide estimates that cover all program inputs. However, analytical models, too, have drawbacks: analytical models typically capture only processor power since other board components like sensors and radios require fundamentally different modeling methodologies. As better energy models arise, CleanCut can incorporate them.

5.2.2 Path energy model

CleanCut uses the block energy distributions to compute the energy distribution of each path in the program, as illustrated in Figure 5.2 on the right. CleanCut's path energy model accumulates the cost of blocks along a path from its *initial* task boundary to its *terminal* task boundary. A path is a non-branching sequence of basic blocks (Section 5.2.1), loop blocks (Section 5.2.2), or opaque blocks (Section 5.2.2). CleanCut's target programming model does not support recursion, which is uncommon in embedded software where predictability and static resource bounds are often required.

To compute path energy, CleanCut must aggregate the energy of the various types of blocks that comprise the path. If CleanCut represented block energy with a scalar, then it could calculate the energy of a path by simply adding the energy costs of the blocks that make up that path. However, CleanCut represents the energy of each block as a distribution, which precludes simple addition. To accumulate block costs, CleanCut *convolves* the energy distributions for the blocks along the path. Convolving the distributions for two random

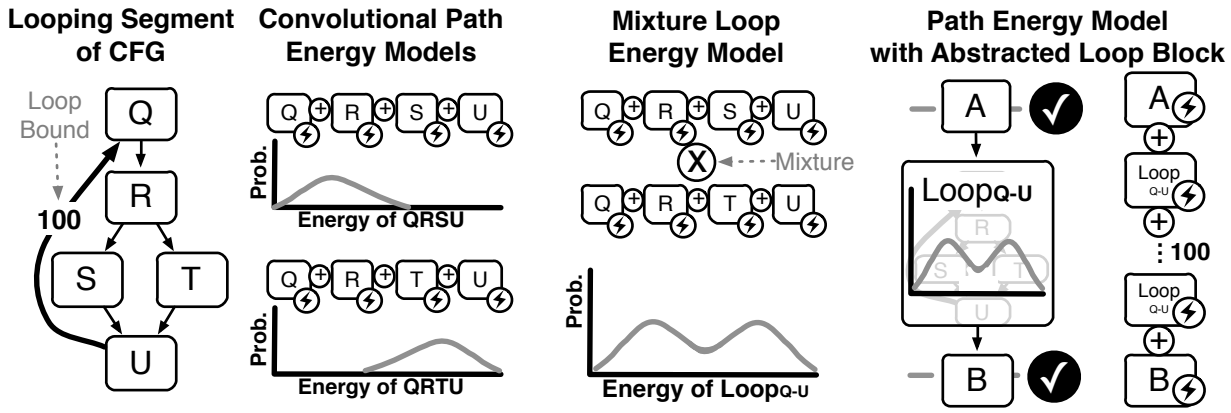


Figure 5.3: CleanCut's loop model mixes path energy models. The figure assumes uniform path likelihood in the loop, but CleanCut can weight paths in a loops mixture using a path profile.

variables (i.e., two block energy distributions) produces a distribution for the random variable that is their sum. Any two arbitrary distributions can be convolved. CleanCut sequentially convolves blocks on a path yielding a distribution representing the energy cost of the path.

Loops

CleanCut handles loops by encapsulating their energy cost in an *abstract loop block*, as shown in Figure 5.3. CleanCut abstracts a loop's body by using a single distribution to represent the energy cost of all paths from the head of a loop to its back edge. A nested loop is recursively abstracted and incorporated into a path through the parent loop. Along a path with a loop, CleanCut convolves the loop body's energy distribution once per loop iteration along with the distributions of the other blocks on the path.

A loop body with many control-flow paths has a *modal* energy distribution, with a mode at the expected energy cost of each path. As illustrated in Figure 5.3, CleanCut computes this modal distribution by *mixing* the distributions for each of the paths through the loop body. To produce an energy model for a path containing a loop, CleanCut convolves the loop body's distribution with the path energy distribution a number of times equal to the estimated loop bound. By default, the loop body's mixture model *uniformly* combines the

intra-loop distributions, treating each path through the loop as equally likely. CleanCut provides an implementation of Ball-Larus path profiling [15] that can determine the likelihood of each path by monitoring representative executions to use as weights in the mixture.

Loops present two main challenges to any analysis. First, the iteration count of an unbounded loop is statically unknowable. Second, an efficient analysis must not unroll the loop. To account for the iteration count of a loop, CleanCut requires the programmer to provide a *bound estimate*, as depicted on the arc (U, Q) in Figure 5.3. For unbounded loops, CleanCut gives the programmer a choice of either providing an annotation statically bounding its iteration count (similar to k-bounded [48] profiles and often simple for embedded applications) or forcing a task boundary inside the loop, which effectively eliminates the loop from the task. For the former choice, to help the programmer determine the loop iteration count, CleanCut has a loop iteration count profiler that can measure the histogram of a loop's iteration counts. The accuracy of the profile for dynamically-bound loops is limited by the sensor inputs during profiling.

I/O operations

CleanCut accounts for the energy cost of I/O operations. The energy of I/O that is contained within a basic block is accounted within the energy for the containing block. Composite multi-block I/O operations (e.g. polling a peripheral) are abstracted into *opaque blocks*. CleanCut measures the energy distributions for opaque blocks *in-place* during a dedicated instrumented run of the application.

5.2.3 Evaluating the energy model in the compiler

CleanCut's compiler uses the recursive procedure shown in Algorithm 1 to calculate each path's energy probability density function (PDF). Before the algorithm runs, a preliminary pass splits any basic blocks with a call instruction and inlines the callee's blocks, recursively. The traversal starts at the entry block and recursively descends along each path until a task

Algorithm 1 CleanCut path energy estimation algorithm.

```
1: function CalcPathEnergies(CFG G, block b)
2:     if IsLeaf[b] ∨ IsBoundary[b] ∨ IsLoopSucc[b] then
3:         return {0}, ∅
4:     E ← ∅,   S ← ∅                                  ▷ Path energies and successors
5:     if ¬ IsLoopHead[b] then                          ▷ Add energy of a block
6:         for s ∈ Successors[b] do
7:             E_s, S_s ← CalcPathEnergies(G, s)
8:             E ← E ∪ E_s,   S ← S ∪ S_s
9:     else                                             ▷ Add energy of a loop
10:        e_l ← 0,   S_l ← ∅                           ▷ Loop energy and successors
11:        for s ∈ Successors[b] do
12:            E_s, S_s ← CalcPathEnergies(G, s)
13:            E_l ← {e ∈ E_s : EndsAtBackedge[Path[e]]}
14:            e_l ← e_l ⊗ e for e ∈ E_l
15:            E ← E ∪ (E_s \ E_l)
16:            S_l ← S_l ∪ S_s
17:        e_l ← e_l × LoopIters[b]
18:        for s ∈ S_l do                               ▷ Add loop to paths after the loop
19:            E_s, S_s ← CalcPathEnergies(G, s)
20:            E ← E ∪ {e_l ⊕ e : e ∈ E_s}
21:            S ← S ∪ S_s
22:    if E ≠ ∅ then return {e ⊕ Energy[b] : e ∈ E}, S
23:    else   return {0}, ∅
```

boundary or a program-terminating block (Line 2). A recursive call (Line 7) returns a list of energy distributions, E_s, for paths that start from the intermediate node and a list of entry blocks into successor tasks, S_s. Each frame adds the current block's energy to each sub-path that starts at a block's child by convolving ($\oplus$) the distributions (Lines 8, 22) and the current block's successors list, S, is extended with its children's successors, S_s (Line 8). To add the energy of a k-iteration loop to a path, the pass recursively computes the energy of each loop body path (Line 12), mixes them ($\otimes$) (Line 14), and convolves the resulting block with itself ($\times$) k times (Line 17). The loop energy is then convolved with each path starting after the loop (Line 20). The set of loop body paths E_l excludes paths that descend into the loop body but reach a task boundary before a backedge (Line 15).

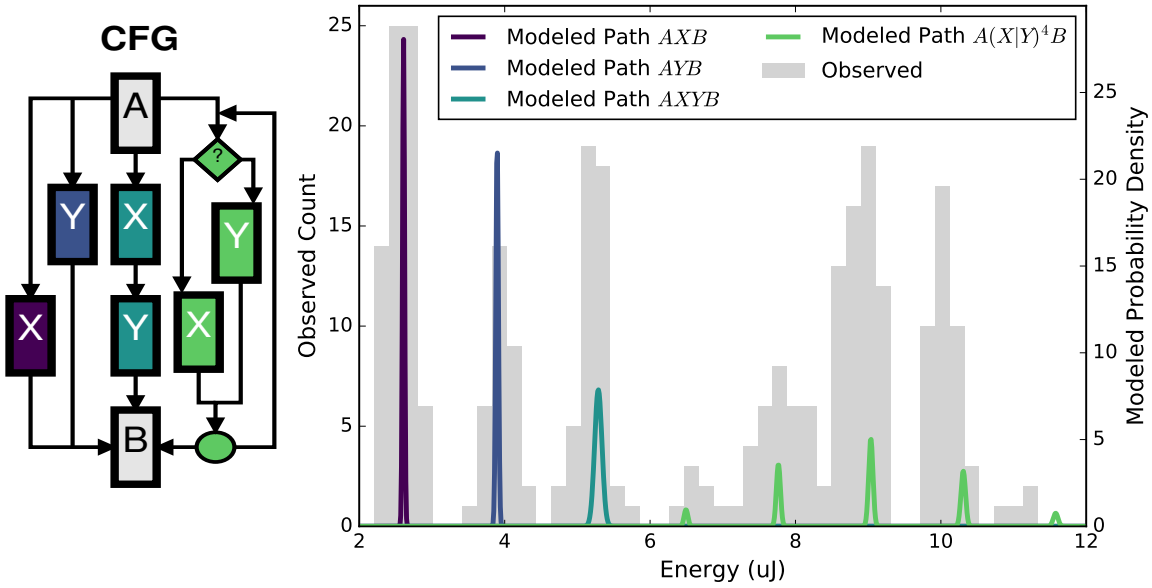


Figure 5.4: Modeled and observed distributions for energy of four paths through a benchmark application (left). The match between locations of the modes on the x-axis validates that CleanCut modeling abstractions correctly represent energy behavior.

5.2.4 Energy model validation

We validated the path energy computation using a microbenchmark to show that the distribution computed by recursive convolutions and mixtures matches the measured energy of the path. Figure 5.4 shows a CFG with four paths comprising simple sequences of blocks and a loop. Each path is composed of three or more blocks of four types, labeled A, B, X and Y, that differ in energy cost. Branches are decided uniformly randomly. The probability density function (PDF) curves in Figure 5.4 show each path's estimated energy distribution.[1] The bars in the plot show path energies measured on the WISP [149] during the 294 independent executions of the program over 5 minutes. There is no correspondence between the scales of the left and right y-axes beyond the relative heights of modes *within data for a single path.*

The key result is that the x-axis position of peaks in a path's modeled distribution corresponds to the path's peak in the observed energy values. The match for path $AXYB$ shows that the energy cost of a sequence of blocks, XY, is correctly modeled by the convolution

[1]For a PDF f, $f(x)$ may exceed 1, but $\int f(x)\,\mathrm{d}x \leq 1$.

of energy distributions for X and Y. The match for each of the 5 modes in the distribution for path $A(X|Y)^4B$ shows that the cost of a loop is correctly modeled by a mixture of energy distributions of the paths through its body. The data also show that CleanCut underestimated path energy variance and overestimated values in the upper range.

5.3 Non-termination checker

CleanCut's non-termination checker evaluates a task decomposition to report non-terminating paths to the programmer if any exist. CleanCut compares an estimate of the energy of each path to an estimate of the energy storage capacity of the device. If there is a non-zero probability that a path energy exceeds the capacity, then CleanCut reports the paths along with the non-termination probability.

Energy available to execute a path is determined by the size of the capacitor installed on the device. To estimate the *effective* energy capacity, which excludes energy spent on initialization after each boot, CleanCut measures energy consumed from first application task until power failure, as described in Section 5.5.1. The estimate is the minimum observed sample. We assume that variations in capacity at runtime induced by temperature or degradation are negligible.

To identify non-terminating paths, CleanCut uses the energy model from Section 5.2 to represent the energy of each path as a probability density function (PDF). CleanCut then integrates the PDF to obtain the cumulative density function (CDF).

Figure 5.5 illustrates how CleanCut evaluates the CDF to estimate the probability that the path energy will exceed the device capacity. The vertical dashed line shows where the device's energy level (E_{dev}) intersects the paths' CDFs. The completing path's CDF is flat at and after E_{dev}, which indicates that it is not a non-terminating path bug. The horizontal dashed line at the intersection of E_{dev} and the non-terminating path's CDF point's y-value shows P^p_{complete}: the likelihood that the non-terminating path p completes using E_{dev}

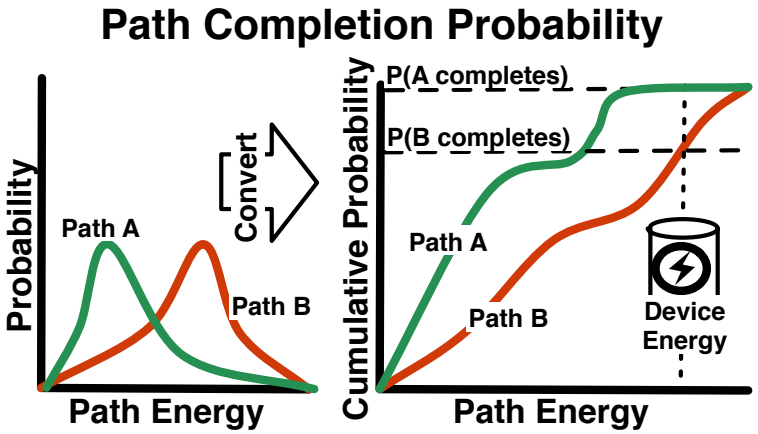


Figure 5.5: CleanCut detects non-terminating path bugs by evaluating the cumulative density function (CDF) of the path energy distribution.

energy or less. The figure shows only two CDFs (one terminating, one non-terminating), but CleanCut's path checker applies this reasoning to all paths' CDFs and reports non-terminating paths and their non-termination probabilities to the program in probability-ranked order.

5.4 Task boundary placer

The CleanCut task boundary placer inserts boundaries into a program to eliminate non-terminating paths while minimizing boundary overhead. The placer's core is the greedy algorithm listed in Algorithm 2. The main loop in DECOMPOSE repeatedly divides the path with the highest energy cost by placing a boundary along the path. Each iteration begins with estimating the energy for all paths through the program (Line 5) according to the energy model (Section 5.2) and storing the estimate as a distribution in the Energy[] field of each path object. For the division and comparison operations (but not addition), the distribution is reduced to a scalar value. The reduction operator is configurable to either the expectation or the maximum observed value; to model worst-case behavior we use the latter with the energy model from Section 5.2.

The algorithm then selects the highest energy path (Line 6) and, if its energy cost exceeds

Algorithm 2 CleanCut program decomposition algorithm.

1: **function** DECOMPOSE(CFG G, device model D) ▷ program G on device D
2: $B \leftarrow \emptyset$ ▷ Initialize set of boundary locations
3: **do**
4: ▷ Evaluate the energy model and return max e s.t. Prob(energy $= e$) > 0
5: $P \leftarrow$ CALCPATHENERGIES(G, B) ▷ Stores energies into field Energy[]
6: $p \leftarrow \arg\max_{p \in P}$ Energy[p] ▷ Pick path of maximum energy
7: **if** Energy[p] $>$ Capacity[D] **then** ▷ Is path predicted to exceed capacity?
8: **if** $|p| > 1$ **then** ▷ Only splits at block granularity are supported
9: $b \leftarrow$ SPLITPATH(p, D) ▷ Place a boundary
10: $B \leftarrow B \cup b$ ▷ Add the boundary to the decomposition
11: **else**
12: **return** "NO PLACEMENT EXISTS"
13: **while** Energy[p] $>$ Capacity[D]
14: **return** B
15: **function** SPLITPATH(path p, device model D)
16: $m \leftarrow 1 + \arg\max_k \sum_{i=0}^{k}$ Energy[p_i] $<$ Energy[p]$/2$ for $0 \leq k < |p|$
17: **for** $i \leftarrow 0$ to m **do**
18: **if** IsLoop[p_i] $\wedge$ Energy[p_i] $>$ Capacity[D] **then**
19: $L \leftarrow$ BodyPaths[p_i] ▷ Block p_i is loop head, get loop body paths
20: $l \leftarrow \arg\max_{l \in L}$ Energy[l]
21: **return** SPLITPATH(l, D)
22: **return** $\arg\min_{s \in [1,m]} 2 \left| \frac{\sum_{k \in [1,s]} \text{Energy}[p_k]}{\sum_{t \in [1,m]} \text{Energy}[p_t]} - \frac{1}{2} \right| + \frac{\text{DYNBOUNDARIES}(p_s, p)}{\max_{t \in [1,m]} \text{DYNBOUNDARIES}(p_t, p)}$
23: **function** DYNBOUNDARIES(block b, path p) ▷ Estimates dynamic transitions
24: **return** $\sum_{b \in \text{InlinedInstancesOfBlock}[p_s]} \Pi_{\text{loop}L | b \in L} \text{LoopBound}[L]$

the device energy capacity (Line 7), the algorithm calls SPLITPATH to choose a location on the path for a boundary (Lines 8-10) using criteria explained in Section 5.4.1. The set of paths P is recomputed on the next iteration, because the new boundary affects not only the path being split but also all paths with a call to the function that contains the new boundary. The placer completes when the costliest path is within the energy capacity of the device (Line 13).

The algorithm must divide looping paths with a high energy cost, even if those looping paths are contained within an abstract loop block (Section 5.2.2). If the traversal over blocks in a path encounters an abstract loop block (Line 18), the algorithm *descends* into

the abstract loop block if the energy cost of the loop exceeds capacity (Line 18) and inserts a boundary along the most costly path in the loop body (Lines 19-20). A boundary placed along a path through a loop, invalidates the energy estimate for that loop until it is recomputed in the main loop (Line 5).

5.4.1 Minimizing task boundary overhead

The location of a boundary determines its run-time energy and time overhead. Given a path p, SplitPath finds the location in p where a boundary will have the least impact. The algorithm identifies the *energy midpoint* of the path, i.e., the block at which energy accumulated from either end of the path is below half of the total path energy (Line 16). SplitPath places the boundary at one of the *candidate* split points between the start and the midpoint of the path. The algorithm could consider the split points between the midpoint and the end of the path but does not in order to save time.

SplitPath assigns each candidate split block a score and chooses the candidate with the lowest score (Line 22). The *split score* measures the impact of a boundary using two components: the relative energy of the two segments after the split and the expected number of dynamic task boundaries. A static task boundary at block b leads to as many dynamic task boundaries as there are calls to b's parent function and iterations of (nested) loops that include b at runtime. Function DynBoundaries (Line 24) estimates the dynamic calls and loop iterations from the inlined version of the program CFG — where each call is recursively replaced with the callee's blocks — and from loop bounds. The placer algorithm assumes that the best candidate split point for a boundary is the one that leads to the fewest dynamic boundaries.

5.4.2 Placer algorithm analysis

Correctness. *If a placement exists that is free of non-terminating paths according to Clean-Cut's energy model, then the placer algorithm will find such a placement; otherwise it will*

report failure. A *valid placement* is a placement for which CleanCut's model indicates that all path energy costs are below the device energy capacity. If the loop in DECOMPOSE terminates, then placement B is valid, because the negation of the loop condition (Line 13) implies that the maximum-energy path p is below capacity, which implies that all paths are below capacity. The loop in DECOMPOSE terminates if the least upper bound on the energy of a path in set of paths P^i (set P in iteration i) strictly decreases, i.e. $\max_{p \in P^i} \text{Energy}[p] > \max_{p \in P^{i+1}} \text{Energy}[p]$, because the right hand side of the inequality in the loop condition (Capacity[D]) is constant. The least upper bound on energy of P decreases in iteration i, if SPLITPATH is called on the maximum-energy path in iteration i and if SPLITPATH decreases the least upper bound.

SPLITPATH is called on all but the last iteration, because on the iteration in which SPLITPATH is not called, either the condition in Line 7 is false, which implies the loop condition is false, or the condition on Line 8 is false which violates the premise that a valid placement exists (i.e., some block exceeds the energy capacity). SPLITPATH decreases the least upper bound on energy of paths in P, because SPLITPATH inserts a boundary at the block at index $s \in [1, m]$ in maximum-energy path p (maximum selected in Line 22), which excludes at least block p_0 from the maximum-energy path in the next invocation of CALCPATHENERGIES (Line 5). That the maximum-energy path is shortened follows from the fact that (1) energy is strictly increasing in the number of blocks in the path, regardless of the type of the block, (2) boundaries are strictly appended to the set of boundaries B, and (3) adding a boundary to program CFG G with boundaries B cannot increase the length of any path in G.

Complexity. Let $W(n, e)$ be the number of blocks traversed by the greedy placer algorithm for a program with n paths and the costliest path of energy e. At each iteration of the outer loop, the algorithm splits one path, which may create boundaries on every path in the worst case, doubling the number of paths for the next step, but cutting the maximum energy in half (since the split is done near the energy-midpoint). That is, $W(n, e) = n + W(2n, e/2)$

with $W(n, e) = n$ for $e < C$, where C is the device capacity. The recurrence is bounded by $O(n \cdot 2^{\log e+1})$.

5.5 Implementation

The CleanCut toolchain is organized as a tree of dependent analysis phases in GNU Make, with the checker and placer results near the root and requisite models and profiles at intermediate and leaf nodes. Independent phases run in parallel.

5.5.1 Energy measurement

CleanCut programmatically controls the Energy-interference-free Debugger (EDB) [38] connected to the capacitor on the target device to measure energy. For each measurement, CleanCut places two voltage watchpoints in the application code and EDB records the capacitor voltage at the watchpoints. Energy consumed between the watchpoints depends on the watchpoint voltage measurements, V_{from} and V_{to}, and device capacity, C, as $E = \frac{1}{2}C(V_{\text{from}}^2 - V_{\text{to}}^2)$. Using EDB, CleanCut directly measures full-system energy, not system components.

Using our energy measurement setup, we measure the energy storage capacity on the device and block energy costs. Assuming V_{on} is the voltage when the initialization completes and the first application task begins and V_{off} is the MCU's brown-out threshold, CleanCut computes the *effective* capacity using $V_{\text{from}} = V_{\text{on}}$ and $V_{\text{to}} = V_{\text{off}}$. V_{on} is measured by running the application binary with an EDB watchpoint after power-on code. V_{off} is set from the MCU's specification (we validated that $V_{\text{off}} = 1.8 \pm 0.002$V for our MSP430FR5969 using an EDB watchpoint).

5.5.2 Block and path energy

To measure a block's energy cost, CleanCut extracts assembly generated by LLVM's backend for the target architecture, translates the instruction arguments to make the block runnable outside of its context, replicates it, and inserts it into a harness binary for measurement. To make the block safe to execute repeatedly outside of its context, CleanCut replaces register references with a designated "scratch" register and memory references with random addresses in a designated range. CleanCut generates harness code with the application's clocking and peripheral configuration to reflect true energy consumption. After running the harness binary on the device for 20 s and tracing watchpoints, CleanCut calculates the block energy from watchpoints as described in Section 5.5.1. CleanCut replicates the block being measured in the harness, to ensure that the measured energy is above EDB's watchpoint measurement resolution. The block's energy cost is the energy cost of the sequence of replicas, divided by the replication factor. After a code change, CleanCut compares hashes of canonicalized blocks to the hashes of the existing profiled blocks and only measures block energy for non-matching blocks.

To estimate the path energy distribution (PDF) described in Section 5.2.2, an LLVM pass first traverses the CFG according to Algorithm 1. The pass assembles an expression that symbolically represents the path energy distribution as a sequence of convolutions and mixtures of block distributions. To evaluate the resulting expression to a numerically-represented probability density function (PDF), CleanCut computes convolutions using NumPy [51] and mixtures as an element-wise linear combination of input PDFs.

5.5.3 Checker and placer

The checker computes a cumulative distribution function (CDF) by integrating the PDF that represents path energy using Simpson's method in SciPy [80]. CleanCut uses the CDF to determine a path's failure likelihood for a given device energy capacity C by finding the probability value at the closest index below C in the CDF's array representation. By accept-

ing capacity as a parameter, the checker can validate a program on a range of capacities, using the same CDF.

We implemented the placer (Algorithm 2) in an LLVM pass that incorporates the path energy model. The pass selects the blocks at which to place task boundaries according to the traversal of the CFG in Algorithm 2. The placer invokes the DINO [101] LLVM passes to insert checkpointing and versioning code at each boundary marker.

5.6 Evaluation

In this section, we evaluate CleanCut to show that the checker's validations are useful, the placer is flexible and its task decompositions are efficient, and CleanCut's analysis time is practical for a real developer. Recall that Section 5.2.4 validated CleanCut's energy model. We applied CleanCut to real code on real energy-harvesting hardware. We used the WISP [149] energy-harvesting device, which has an 8MHz MSP430FR5969 MCU with 64KB of non-volatile memory and a 47 μF capacitor. We powered the WISP wirelessly using a ThingMagic Astra-EX RFID reader at 16 dBm. We fixed the WISP 45 cm from the power antenna, parallel to its surface.

5.6.1 Benchmarks

We evaluated CleanCut on four energy-harvesting applications from prior work [101, 39]. Activity Recognition (AR) classifies 8 windows with 8 accelerometer samples each into two activity classes based on a pre-trained model. RSA encrypts an 11-character plaintext with a 32-bit public in non-volatile memory. Cuckoo Filter (CF) exercises a Bloom-filter-like set membership structure that supports deletion. CF inserts 64 pseudo-random keys and then queries for each. The Cold-chain Equipment Monitor (CEM) records 64 temperature readings from a sensor, LZW-compresses them, and stores the result into non-volatile memory.

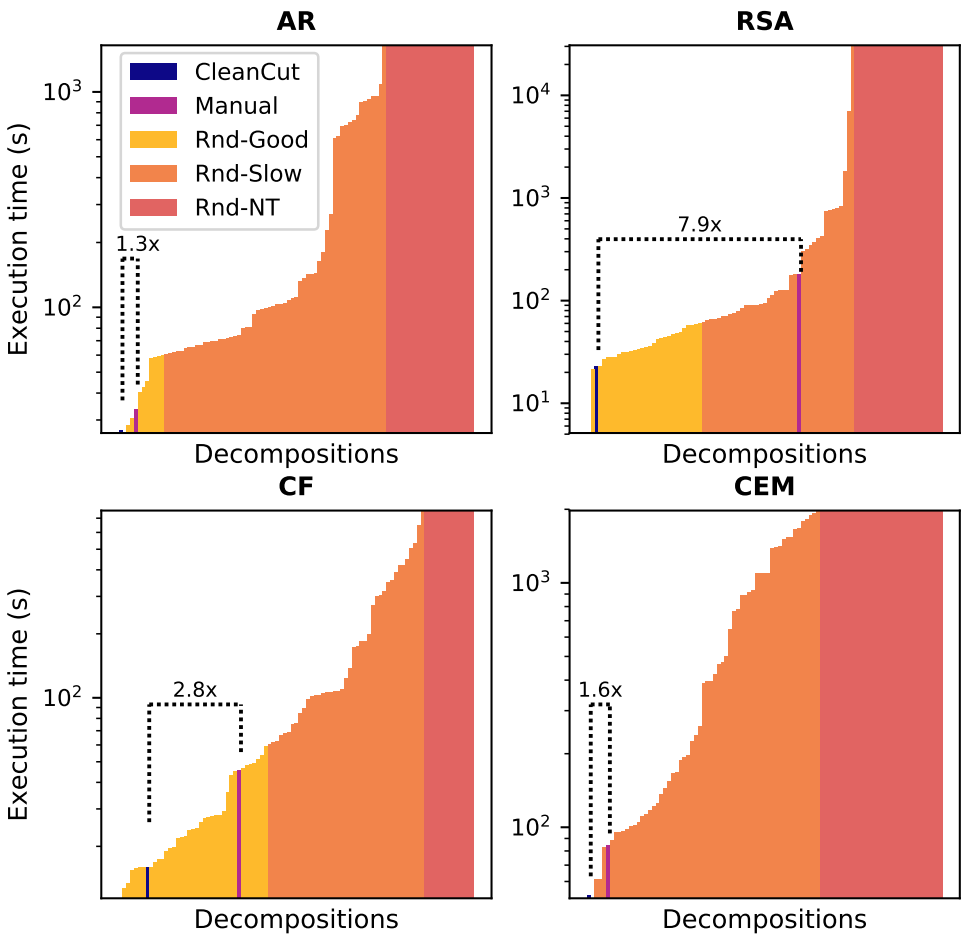


Figure 5.6: Application execution time when decomposed by CleanCut, manually, or randomly. Random decompositions are grouped into completing within one minute (**Rnd-Good**), completing after a long time (**Rnd-Slow**), and not completing (**Rnd-NT**). The speedup of CleanCut relative to the manual strategy is shown in the annotations.

5.6.2 Placer evaluation

We evaluated how well CleanCut's placer helps to insert task boundaries into a program to avoid non-terminating paths. The evaluation shows that CleanCut's decompositions are superior to the programs' original, manually placed boundaries and random placements. Our results also show that CleanCut provides flexibility to changing hardware, while avoiding non-terminating placements.

Performance

The main result of our placer evaluation is that CleanCut produces higher-quality, more efficient placements than a modular, manual decomposition strategy and a large number of randomly-generated potential boundary placements. We assess the quality of a decomposition by measuring the run time of the decomposed application on the real device. Modular decomposition is an intuitive manual approach of placing a task boundary at the entry of each major function or outer loop. Random decompositions place boundaries at basic blocks chosen uniformly at random from the CFG. We generated 10 random decompositions for each possible boundary count between 1 and 10, for a total of 91 distinct decompositions (there is only one one-boundary placement because CleanCut requires a boundary at the top of main). We measured execution time by wrapping the main function with EDB watchpoints that collect timestamps when hit.

Figure 5.6 compares run times for CleanCut, manual modular, and random decompositions. CleanCut consistently outperforms the modular decomposition, with a harmonic mean speedup of 2.45x. CleanCut also outperforms all *terminating* random decompositions for AR and CEM, and is slower only than 2 out of 78 random placements for RSA and 7 out of 68 for CF. Several of the random decompositions that are slower than CleanCut are slower by an order of magnitude or more. The placer's decompositions are more efficient, because they contain fewer boundaries than manual and random decompositions, incurring less checkpointing overhead. The low boundary count is a benefit of CleanCut's energy model: the placer's algorithm splits the path with the highest energy cost maximally amortizing boundary cost across the largest available span of code. In contrast, manual decompositions have many boundaries, because the authors of these applications were conservative and relied on intuition alone to estimate task energy cost. The overly conservative assumptions lead to the high overhead in Figure 5.6.

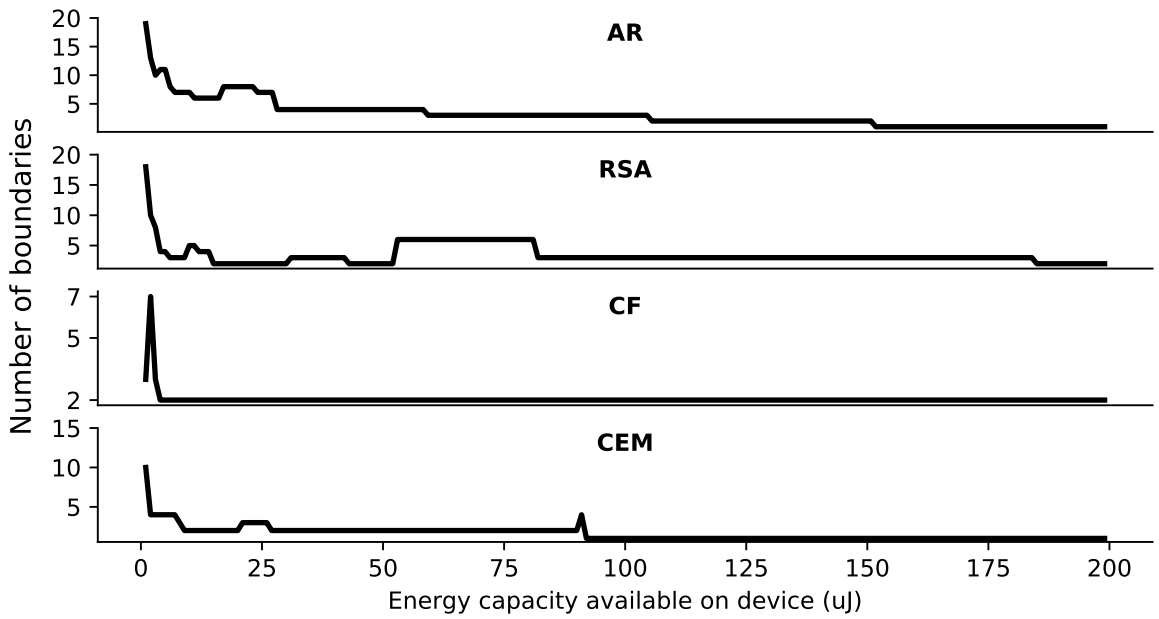


Figure 5.7: Number of task boundaries in CleanCut decompositions. As opposed to a manual decomposition, CleanCut adapts its decompositions to the energy capacity available on the device. The larger the energy capacity, the fewer boundaries are placed.

Adaptation to Changing Hardware

CleanCut is parameterized by the energy storage capacity of the target device. This flexibly lets the user apply CleanCut as hardware specifications change. The manual decomposition strategy lacks flexibility: A decomposition that is valid on one device, may not terminate on a device with a smaller energy buffer, and may be inefficient on a device with a larger buffer. Figure 5.7 shows how CleanCut selects a different boundary count for different target energy capacity. The counterintuitive increase in boundary count with capacity is a result of the placer's greedy algorithm.

5.6.3 Checker evaluation

We evaluated CleanCut's checker by using it to identify non-terminating path bugs in the same pool of random decompositions used in Section 5.6.2. The goal of this evaluation is to show that the checker reliably reports non-terminating paths and rarely reports that a

path is non-terminating when it is terminating. After obtaining the checker's predictions for each path in each decomposition, we executed the decomposition on the WISP energy-harvesting device on RF power. During execution some (unknown) subset of the program's paths executed, depending on the real experimental input from the sensors. The outcome of each execution is either that the decomposition terminated, implying that no path *that executed* had a non-termination bug, or that the decomposition did not terminate, implying that a non-terminating path executed. A non-termination prediction for a *program* may not match the observed behavior because not all *paths* execute in all runs as a result of input variation.

Figure 5.8 plots the predicted energy for every path in any decomposition that terminated (left plots) and that did not terminate (right plots). Groups of paths from the same decomposition are adjacent, of the same color, and sorted by energy. The horizontal line indicates the measured energy capacity of the device. For a terminating decomposition, we expect that *for all* paths that executed – and for most potential paths – the checker predicts the energy to be below the capacity; i.e. the adjacent vertical bars of the same color should be below the red line if the paths that they represent executed during the trial run. Paths in a terminating decomposition that were predicted to be above capacity either did not execute, or their energy was overestimated by the model. For a non-terminating decomposition, we expect that *there exists* at least one path for which the checker predicts the energy to be in excess of capacity; i.e. from each group of bars of the same color, *some* vertical bars (corresponding to the paths that executed) are above the red line. Paths in a non-terminating decomposition that were predicted to be below capacity are expected, because it only takes a single non-terminating path to prevent an execution from terminating. However, if a non-terminating decomposition has no path whose energy was predicted to be above capacity, then CleanCut underestimated the energy cost of at least one non-terminating path.

These expected trends are visible in Figure 5.8. Every non-terminating decomposition had at least one predicted non-terminating path in CF and CEM; and all but one non-termi-

nating instance had such a path in AR. The results for these benchmarks show that CleanCut successfully identified non-terminating paths. In RSA, all but seven of the non-terminating decompositions had at least one path predicted not to terminate. For the remaining seven, we identified the source of the underestimate to be inaccurate loop iteration counts for some dynamically-bound loops (e.g. division, container search) that we obtained by profiling on fixed inputs. CleanCut is likely to perform better with more representative profiling and on applications with statically-bound loops.

For terminating decompositions, CleanCut correctly identified that a majority of paths do not exceed the energy capacity of the device. In CEM, CF, and RSA, only a few instances include paths that exceed the capacity. Such paths either did not execute during the trial run or were overestimated by the model. In AR, at least one terminating distribution has all paths predicted as non-terminating (middle of the X-axis). Since the energy of the paths in this group is similar and above capacity, they likely share a common prefix that was overestimated. This overestimate is likely due to overly conservative loop bounds on the loops that implement arithmetic operations for the pattern matching. The presence of some paths predicted non-terminating in a terminating decomposition, is evidence of CleanCut's conservatism, because these paths are reported to the programmer.

5.6.4 Characterization

We evaluated the practicality of using CleanCut by measuring the time to validate a program decomposition using the checker and find a decomposition using the placer. Table 5.1 summarizes the complexity of each application, showing block count, call depth, and maximum path count across all evaluated decompositions. The table also reports the execution time of CleanCut's costliest phases. The block profiling cost varies with block count and averages 36 minutes. CleanCut incurs this cost only the first time it runs; after incremental changes, only changed blocks are re-profiled.

We measured the time CleanCut takes to check the manual placement and to place

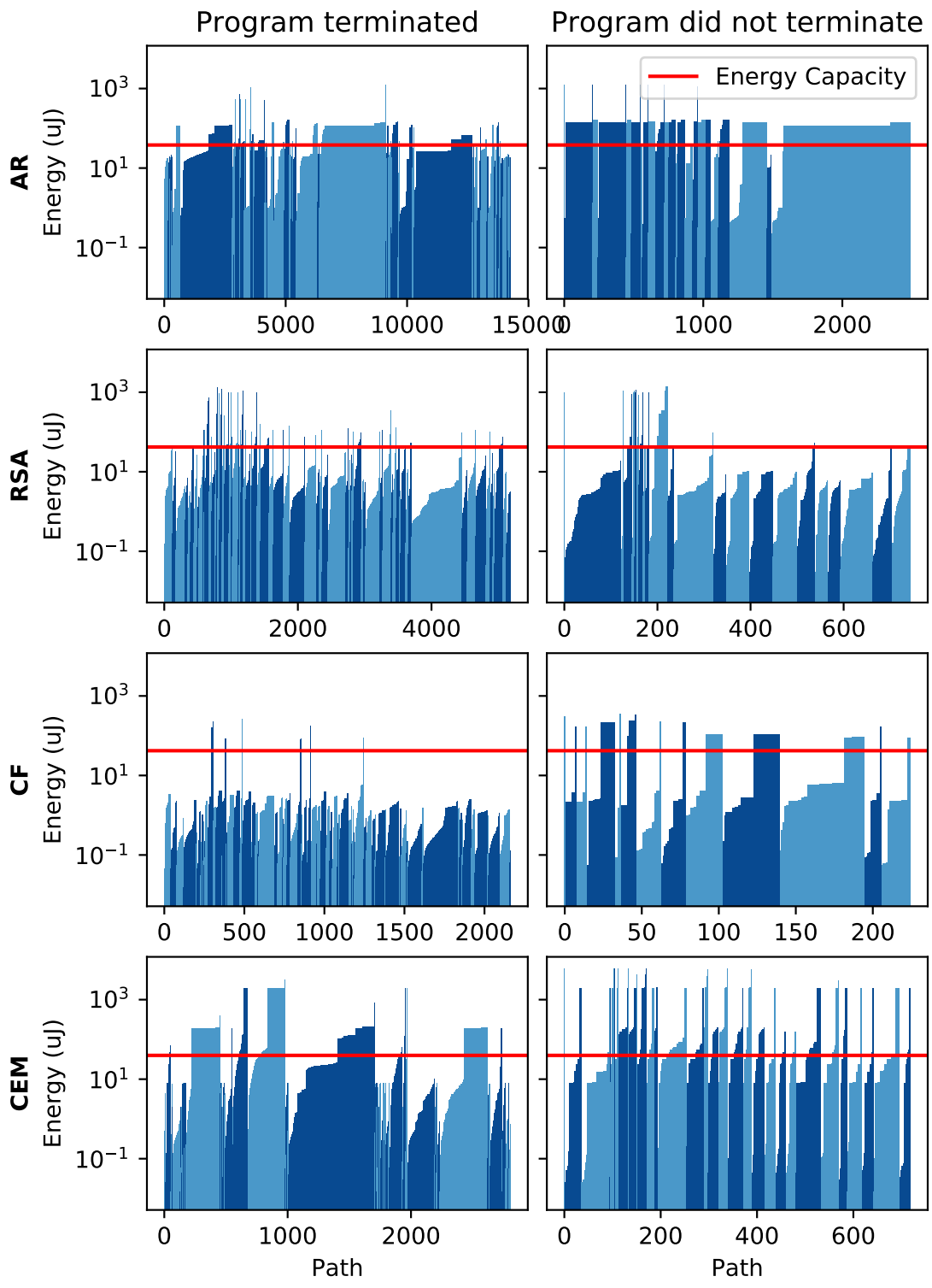


Figure 5.8: Predicted and observed non-termination for random boundary placements.

	App. Characteristics			Analysis Time		
App.	**BBs**	**Paths**	**Call Depth**	**BB Prof.(m)**	**Checker (s)**	**Placer (s)**
AR	187	298	5	45	30	24
RSA	197	326	5	51	57	56
CF	91	217	2	23	36	8
CEM	70	80	2	24	54	11

Table 5.1: Benchmark and analysis time characteristics. Listed are total basic block counts, maximum path count across all decompositions studied (including random), and the maximum call depth. Times are for one-time block profiling (**BB Prof**), checking a decomposition (**Checker**), and finding a valid decomposition (**Placer**).

boundaries in the original uninstrumented code. Evaluating energy expressions occupies the majority of the run time due to numerical operations on distributions. The time cost increases with the application size and number of paths. For example, CleanCut takes longest on RSA, which has 1.5-2.4x lines of code of the other applications. The checker takes longer than the placer, because it computes more detailed information for the bug report, such as the CDF of the energy distributions for each path. The running time of the placer increases with the number of path splits it has to perform, which decreases as the device capacitor size increases. For the benchmarks we evaluated, adding CleanCut analysis to a build system increases the build time in fully-checked mode by under one minute.

5.7 Summary

In this chapter we identified and addressed the problem of validating and generating task decompositions of programs written for an intermittent execution model. Our system, CleanCut, builds a statistical model of the energy cost of paths through a program. CleanCut's checker uses this energy model along with a model of the energy buffer of the target device to report non-terminating paths. CleanCut's placer iteratively generates a *terminating* task decomposition for a program by inserting task boundaries between basic blocks. Having evaluated our CleanCut prototype on a real energy-harvesting device, we showed that its checker is accurate and its placer quickly identifies efficient task decompositions.

Chapter 6

Capybara: A Reconfigurable Energy Storage Architecture

Computing, sensing, and communication tasks in an application place a spectrum of constraints on the energy-harvesting power system of the device. Tasks are sensitive to the length of the intervals during which the device is off to accumulate energy and during which it is on consuming the stored energy. Computational tasks place the fewest constraints, because they can be interrupted and resumed mid-way through their execution. Assuming a task decomposition without non-terminating tasks (cf. Chapter 5), a computation will successfully make progress each time the power system provides some energy to it. In contrast, operations that must execute *atomically*, i.e. will not be successful if interrupted by a power failure, such as sending a radio transmission, constrain the time the device must stay on. Operations that must execute frequently, e.g. sampling a changing physical quantity, constrain the time the device can stay off (to accumulate energy). Sensing tasks that must perform an atomic operation at an unknown time in the future, e.g. send a radio packet in response to a sensor value, place a hybrid constraint that stipulates that the device should have a short off interval while having a burst of energy ready for immediate use. We refer to tasks in the latter three categories as tasks with *temporal constraints* on energy.

Temporal constraints on energy are challenging to meet, because in the intermittent execution model, software cannot control when the device turns on and when it turns off. The execution pattern is a function of input power and energy storage capacity of the device. The larger the energy buffer, the longer the device can stay on but it must also stay off longer to fill the buffer with energy. The smaller the energy buffer, the shorter the off interval and the shorter the on time. A hardware designer may build a power system with a large energy buffer to meet the on-time constraint. However, after depletion, a large energy buffer has a long recharge time. During the recharge period, the device is off and the off-time constraint of other tasks might be violated. Alternatively, a hardware designer may build a power system with a small energy buffer designed to have a short off time. However, the small buffer may store insufficient energy to satisfy all on-time constraints. The application is not fully functional with either choice.

Figure 6.1 illustrates how fixed energy buffering fails to meet application demands. The application attempts to collect a time series of 15 sensor samples to cover a time interval and transmit the data by radio. The figure shows how stored energy (energy buffer voltage) varies with time when the application executes with two different capacities. Blue regions are operating periods and white regions are recharge periods. 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 completely transmit by radio. With a *large* energy buffer (right), the application buffers sufficient energy to transmit. However, the application spends a much longer period of time charging and fails to sample the sensor reactively. There are no samples in the long recharge spans, and many back-to-back samples.

A composite temporal constraint on energy arises in tasks with requirements on both the off-time and the on-time intervals. The dual requirement arises in *asynchronous* tasks that must react to an event at an unpredictable time in the future, such as an environmental signal or an interaction with a user. To handle unpredictable events, a device's power system

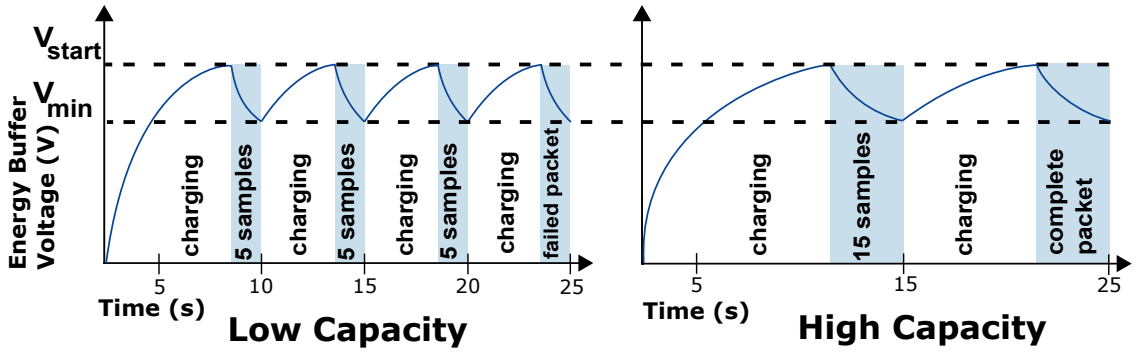


Figure 6.1: Execution with a fixed-capacity energy buffer. Devices are forced to trade short charge cycles for the ability to complete energy intensive tasks.

must provide support to *reserve* energy in advance of a high-energy task that will execute at some unpredictable point in the future.

Without a way to reserve energy before an event, the system has to pause and accumulate energy after the event, but before executing the task that reacts to the event. This recharge period is on the critical path in reactive latency-sensitive applications such as human-computer interfaces. A concrete example of such an application samples a sensor, detects a specific event, and sends a radio alert. The radio alert task has a temporal constraint on both the off time and the on time. Radio transmission demands a high capacity energy buffer that needs to be available immediately after the event. These tandem constraints are beyond the capability of fixed-capacity power systems.

The key capability missing from existing systems to satisfy temporal constraints is software control over when and in what quantity the device accumulates and consumes energy. A platform without this capability must be provisioned with a fixed energy buffer sized for the worst-case, i.e. highest energy consumption. Power systems of state-of-the-art energy-harvesting devices do not support programmatic reconfiguration of energy capacity at run-time. The limited control over charge and discharge timing available to programmers today could be attained indirectly with control code that puts the device to sleep or shuts it down to charge (cf. Section 7.1) at key points in the application code. Such control code expresses high-level energy constraints indirectly and imperatively through *ad hoc* device-specific code.

We propose Capybara,[1] a high-level abstraction for specifying temporal constraints on energy implemented in a hardware/software system that executes tasks according to those constraints. Capybara supports *declarative specification* of temporal energy constraints and eliminates the need for imperative power system control code that entangles application logic with low level hardware configuration. The constraints are specified as annotations to tasks of a task-based system like Chain (cf. Chapter 4). The runtime system uses a novel hardware mechanism to *reconfigure* energy storage capacity as tasks execute. We use our hardware/software prototype of the Capybara platform to show that it enables applications to be more reactive than is possible with a fixed-sized capacitor.

The next section of this chapter provides an overview of Capybara's hardware and software components. Section 6.2 describes the software interface and runtime, and Section 6.3 describes the reconfigurable hardware. We evaluate Capybara on a real energy-harvesting platforms in Section 6.4 and summarize in Section 6.5.

6.1 Design overview

Capybara is a system composed of co-designed hardware and software components that match the energy buffering capacity of the hardware to the energy demand of the tasks in the application. Figure 6.2 shows a high-level system overview of Capybara (top) and a task-based intermittent program with energy mode annotations for Capybara (bottom).

A Capybara system may include general purpose computing components and memories alongside arbitrary peripherals, such as sensors and radios. A motivating insight behind Capybara is that using each of these components to perform a useful quantum of work without a power failure requires a different amount of total energy. Capybara provides an energy reservoir configurable to multiple different energy capacities. The energy buffer configurations correspond to different operating modes that, in turn, correspond to different energy requirements presented by software tasks.

[1] Capybara: **Cap**acitor-based energ**y** **b**anks **a**s a **r**econfigurable **a**rray

Task-based programming model. The application depicted in Figure 6.2 shows software tasks expressed in a task-based intermittent programming model, like Chain introduced in Chapter 4 or Alpaca developed later [106]. In such a model, the programmer decomposes an application into function-like tasks. Control flows from one task to another when one completes, at a `nexttask` statement.

In the figure, the tasks require different amounts of energy. Computing requires little energy, sensor processing requires more energy, and encoding and communicating by radio requires yet more energy. The different tasks are annotated with different modes that express their different energy requirements, and correspond to the energy buffers of different capacity on the Capybara board. Section 6.2 describes how a programmer conveys task energy requirements through Capybara's programming interface. Section 6.3 shows how Capybara's hardware implements a reconfigurable energy reservoir using an array of capacitor banks.

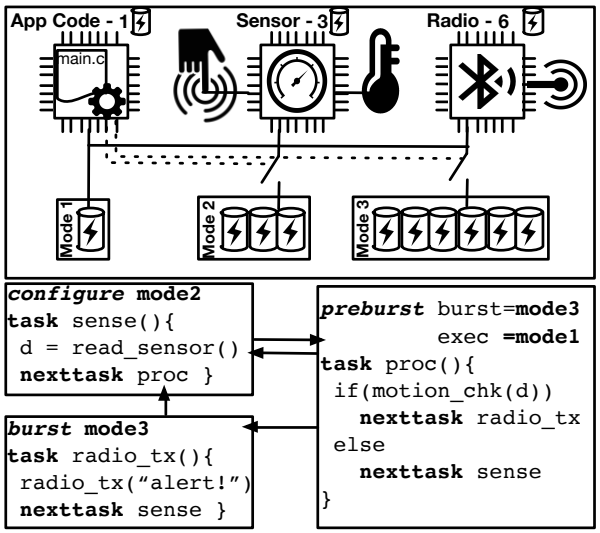


Figure 6.2: Overview of Capybara. The platform has resources for computation, sensing, and communication, and includes three energy storage configurations with different capacities. An example program has tasks annotated with energy mode requirements.

Defining task energy requirements. A pre-requisite to using Capybara to build a system is to measure the absolute amount of energy required by each of an application's tasks and to identify the absolute amount of capacitance required to furnish that amount of energy. From the software perspective, Capybara abstracts the specific amount of energy required by a task, instead allowing software to refer to a task's *energy mode*: an identifier that corresponds to the specific amount of capacitance required to execute the task (discussed

in detail in Section 6.2). Capybara's power system (described in Section 6.3) is designed to allow a hardware designer to partition a set of capacitors into one or more banks such that the capacitance needs of all energy modes can be met by activating some subset of the banks.

A programmer should define energy modes and provision hardware only once an application's code is stable, to avoid re-provisioning as code changes. Energy provisioning requires measuring a task's energy consumption, including initialization and warm-up of peripherals that the task exercises. A simple way to estimate energy consumption is to run the task using an increasingly large energy buffer until the task successfully completes. Another approach is to measure task energy consumption on continuous power using a current sense amplifier and analytically derive the required capacitance and tune it by measuring energy storable in trial capacitor arrays. The translation of the energy estimate into a capacitance value may take into account degradation of the capacitor material over time by the standard practice of *derating*, i.e. overprovisioning by a margin.

Capybara's focus is *not* developing a methodology for measuring the energy of software tasks and this simple measurement-based approach is reasonable (and similar to UFoP [66]). The key insight of Capybara is, instead, the need for *reconfigurability* of hardware energy buffering resources by software to meet *varied* software energy demand. Optimizing the methodology for measuring software energy demand is an important, yet orthogonal problem.

6.2 Software support

Capybara provides a programming interface and runtime software support to reconfigure the power system to meet an application's varied task energy demands. To express the energy demands of a task, a programmer annotates the task *declaratively* with the task's *energy mode*. As the program executes, the Capybara runtime library dynamically reconfigures the power system hardware to execute tasks with their specified energy mode. In hardware,

an energy mode corresponds to a specific configuration of Capybara's reconfigurable energy storage reservoir. Section 6.3 discusses how energy modes are implemented in hardware.

6.2.1 Energy modes

In a Capybara system, an *energy mode* is a property of an application task that expresses a demand of the power system to meet a temporal constraint on energy. Recall from Section 7.1 that a task with a constraint on the on time requires a specific minimum amount of energy to complete without being interrupted by a power failure; a task with a constraint on the off time requires an operation to occur *reactively*, without a long recharge delay; and a task with both types of constraints requires a specific minimum amount of energy to be *reserved* to reactively be consumed at some future point.

The programmer annotates a task with parameterized keywords to associate the task with an energy mode. 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`. As Section 6.3 describes, a hardware configuration concretely corresponds to the activation and de-activation of energy banks by means of a custom switching circuit. When a task with a `config` (`mode`) annotation starts executing, the Capybara runtime issues a command to the power system to configure the reservoir to capacity that corresponds to `mode`. The system then charges the newly configured energy buffer. When the buffer is full, the task executes. The system designer is responsible for ensuring that the hardware configuration that corresponds to `mode` meets the requirements of a task annotated with `config` (`mode`); we discuss the process of doing so in Section 6.1.

The programmer can use a `config` (`mode`) annotation to indicate a constraint on either on or off time. For an on-time constraint, the programmer is expressing that `mode` corresponds to a particular configuration of the hardware energy store that can buffer sufficient energy to execute the task without a power failure. For an off-time constraint, the programmer is expressing that `mode` corresponds to a hardware configuration that buffers sufficient energy

to complete the task, but also that minimizes recharge time for reactivity.

Figure 6.2 shows a high-level schematic of the mapping between hardware energy buffers and software energy modes. The exemplified Capybara-based platform is equipped with three hardware energy buffers connectable through switches in several arrangements. A configuration of the switches, which are controlled by the Capybara runtime system, corresponds to an energy capacity configuration. In the figure, there are three different energy modes, each of which corresponds to a different subset of hardware banks. The `sense`() task in the figure requires the three units of energy provided by the capacitor arrangement inside the `mode`2 box as a result of the `config` (`mode`2) annotation on the task. Before `sense`() can execute, the Capybara runtime requests this arrangement from the power system. After the reservoir charges, the device boots, and the runtime executes `sense`().

6.2.2 Responsive asynchronous bursts

Capybara allows tasks to have an on-time constraint and also to be reactive using its support for *bursts*. The Capybara API includes two additional task annotations that support bursts: `burst` and `preburst`. A task annotated with `burst` (`mode`) requires the specific (possibly very large) amount of energy designated by mode `mode` at a time in the future that is unpredictable, e.g., in response to a specific sensed event. 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`. The key difference between a `burst` task and a `config` task is that Capybara does not need to pause to recharge before executing the `burst` task, because the energy buffer had been filled *ahead of time*.

A programmer can use Capybara's `preburst` task annotation to charge a `burst` task's mode ahead of time. The programmer will annotate a task that is off of the critical path of the `burst` task's operation with the `preburst` annotation. The pause to charge to the `burst` task's mode will then occur before the `preburst` task, well in advance of the time critical

`burst` task. When execution reaches a task annotated with `preburst` (`bmode`, `emode`), the Capybara runtime takes several steps. First, Capybara configures the hardware for the energy mode `bmode` and pauses until the energy buffer for `bmode` is fully charged. Second, Capybara configures the hardware for `emode`, de-activating the energy buffers of `bmode`. A key property of Capybara is that a de-activated mode's energy buffers *retain* their stored energy, except the energy lost to leakage. Third, Capybara pauses to fully charge the energy buffer for `emode`. Fourth, after fully charging, Capybara executes the `preburst` task with the hardware configured for `emode`. The `preburst` task pays the `burst` task's recharge latency in advance when the latency is tolerable, to save the `burst` task from paying its recharge latency on-demand, when the latency is intolerable.

In Figure 6.2, `proc`() is the `preburst` task that charges the energy buffers that the `burst` task `radio_tx` needs to execute. `radio_tx` is a `burst` task because it must be *responsive*: when `proc`() detects a motion event in the data collected by `sense`(), the application should send an alarm immediately. If `radio_tx`() were not a burst task, the system would incur the latency of a full charge of the energy capacity required by `radio_tx`(), which could be tens to hundreds of seconds, depending on incoming power and radio hardware. With `preburst`, Capybara eliminates the latency between the event detection and the alarm delivery by charging ahead of time.

6.2.3 Capybara runtime implementation

We implemented Capybara's task annotations in a runtime software library. The runtime includes a GPIO-based interface to Capybara's power system hardware to reconfigure energy buffers for an energy mode; we discuss the switches and energy buffers more in Section 6.3. The runtime also implements a non-volatile state machine to support `preburst` and `burst`. Our Capybara runtime implementation ensures that all operations are robust to power failures by careful use of non-volatile memory.

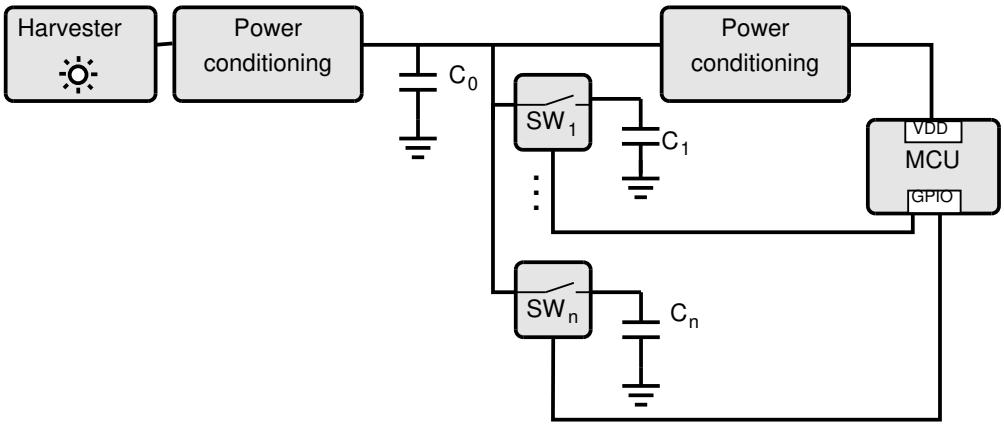


Figure 6.3: Capybara power system with reconfigurable energy capacity using an array of capacitor banks. Each bank of capacitors is connected through a switch circuit. Each switch is controlled by software through a GPIO pin.

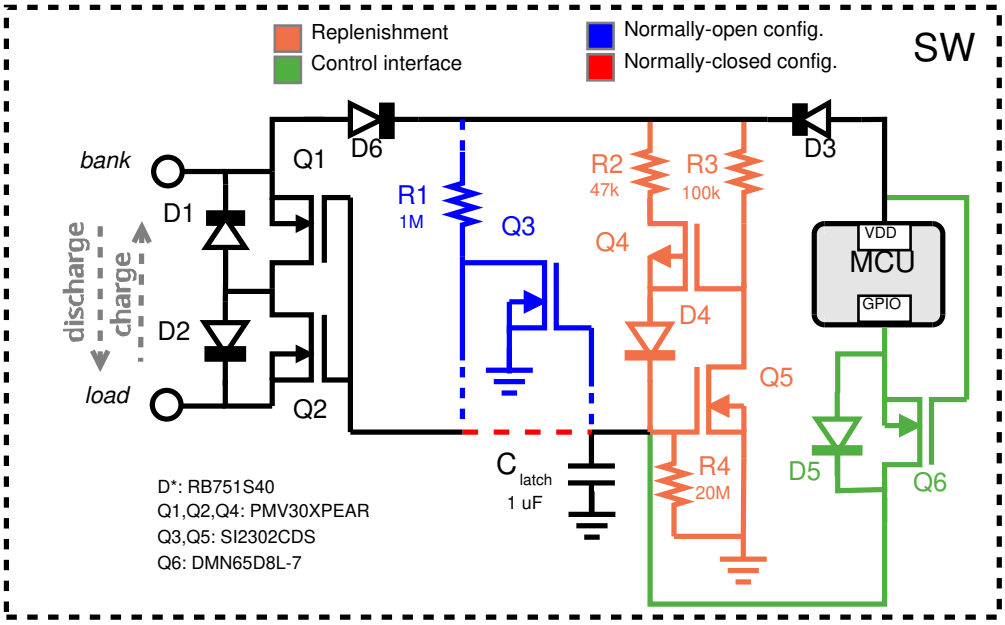


Figure 6.4: Replicable hardware switch module for reconfigurable capacitor banks.

6.3 Reconfigurable power system hardware

Capybara introduces a novel, reconfigurable *power system* architecture with support to programmatically reconfigure the device's energy storage capacity and accumulate energy for asynchronous bursts. The power system hardware architecture is illustrated in Figure 6.4(a).

Energy stored in a capacitor of capacity C that is charged to a voltage V_{top} and discharged

to a voltage V_{bottom} is $E = \frac{1}{2}C(V_{\text{top}}^2 - V_{\text{bottom}}^2)$. To reconfigure the energy storage capacity, the hardware must provide a mechanism for runtime control of one or more of V_{top}, V_{bottom}, or C. We evaluate the merits of each mechanism by comparing the time the device needs to *cold-start* from empty capacitor until boot and the hardware complexity, cost, and durability.

The mechanisms that manipulate either voltage threshold must monitor the voltage on the capacitor with a comparator, either as the device charges (when controlling V_{top}) or as it discharges (when controlling V_{bottom}). To control V_{bottom} the comparator with a resistor network built into the MCU can be used. The built-in comparator is not an option for controlling V_{top}, because the reference must be settable at runtime, must persist while unpowered, and the comparator output must be valid at voltages down to zero. Furthermore, the monitoring overhead while the device is charging increases the minimum incoming power necessary to charge at all. In addition, both voltage-based mechanisms must charge the capacitor to above the minimum for the output booster before any *usable* energy can be accumulated. As a result, cold start is longest for the voltage-based mechanisms. With V_{bottom} control, cold-start time is longer than with V_{top}, because the capacitor must charge to the top threshold even for a short on-time requirement.

The shortest cold-start time is achieved by controlling C. The smaller C is, the quicker the capacitor charges to the minimum boostable voltage. To control C the energy storage must be composed of an array of capacitors connected through *persistent switches* settable at runtime. For Capybara, we chose a mechanism for controlling C for its cold-start advantage and its lower power and space overhead compared to our prototype of a V_{top} mechanism. We prototyped the latter using a non-volatile digital potentiometer based on EEPROM and found that it occupies twice the area and consumes 1.5x the leakage current (according to component specifications). Another advantage of controlling C is its natural wear leveling for capacitors with limited charge-discharge cycles (e.g. EDLC supercapacitors). Taking inspiration from the concept of caching, dense but fragile capacitors can be dedicated to a bank and used only when another bank with less dense but more robust capacitors is

insufficient.

Capybara implements the mechanism for controlling C, with an array of capacitor *banks*, each of which is individually connectable to the device through programmatically-controllable *state-retaining switches*, as illustrated in Figure 6.3. The number of banks and the energy capacity of each bank is provisioned at design time, to match the energy modes that a programmer identifies in a target application. Section 6.1 discussed how to determine an application's energy modes. Figure 6.4 shows the switch circuit that *activates* a bank. The figure includes two design variants, "normally-open (NO)" (blue) in which the switch does not conduct by default and "normally-closed (NC)" (red) in which the switch conducts by default.

The NC and NO switch choice differ in the *implicit* capacity reconfiguration that takes place when the input power is lost for longer than the switch can retain its state. Once power becomes available and the device boots, the runtime system remains unaware of the capacity reconfiguration, because retaining the state loss event is as problematic as retaining the switch state, and an introspection circuit for reading switch state would severely decrease the switch retention time due to leakage. With a NO switch, the energy storage capacity reverts to the (small) default bank, which will charge quickly once power becomes available. However, if the default bank is insufficient for the current task, its first execution attempt will be wasted. Under an adversarial input power timing, the cycle of switch state loss, incomplete task execution, and switch reconfiguration may repeat indefinitely. A NC switch reverts to maximum storage capacity, which takes longest to charge but guarantees successful execution on first attempt after boot.

The *bidirectional* switch interrupting a bank's charge path is implemented using two P-channel MOSFETs connected in series (Q1 and Q2) in a high-side switch configuration. Two MOSFETs are necessary to block current in both directions, since even when a MOSFET is off, it will conduct current in reverse direction, through its internal body diode. When the MOSFETs are off, current is blocked in both directions because at least one MOSFET

does not conduct. When the MOSFETs are both on, the current path flows through one of the MOSFETs and one of the diodes: Q1 and D2 when discharging, and Q2 and D1 when charging. An external diode is added across each switch MOSFET, in parallel with the internal body diode, in order to decrease the forward voltage of the internal diode. A lower forward voltage lets the bank charge to a higher voltage and discharge to a lower voltage. The charge on the latch capacitor (C_{latch}) preserves the switch state while the device is not powered.

To compensate for the leakage of the latch capacitor, the *replenishment* circuit (orange) connects the latch capacitor to the highest voltage source in the circuit whenever the latch capacitor is charged and the device is powered. Due to non-ideal properties of MOSFETs, the replenishment circuit leaks current into the capacitor through Q4, even when the replenishment is not enabled (i.e. when latch capacitor is not charged). If no alternative path for this leakage current is provided, it will eventually charge the latch capacitor, and flip the state of the switch from to closed to open for the NC variant, and from open to closed for the NO variant. Resistor R4 provides a path to ground for this leakage current, at the trade-off of lowering switch retention time.

Software running on the MCU can control the switch using a GPIO pin that charges or discharges the latch capacitor through the *interfacing* circuit (green). The interface isolates the latch capacitor from the MCU pins, to prevent the MCU pin draining the latch capacitor when the MCU loses power and the pin loses its high-impedance state. A GPIO pin is not in a high-impedance state when the MCU supply voltage is zero, because the pin will conduct through the protection diode to the supply rail whenever any positive voltage is connected to it.

Capybara's reconfigurable switch circuit works according to the same principle as a DRAM cell. Both retain state using a capacitor. The key differences are that in the Capybara switch, the logical value is not only stored but always applied to control a gate, the access transistor's purpose is not to multiplex but to isolate, and the replenishment circuit

must be always active and dedicated to each capacitor.

Figure 6.5 demonstrates the reconfiguration of the system capacity using the proposed switch. The trace shows the voltage on the two energy banks in the system: Bank 0 (VBANK0) and Bank 1 (VBANK1). Bank 0 consists of 3 ceramic capacitors of size 100 μF each, and is always connected. Bank 1 consists of 3 EDLC supercapacitors of size 7.5 mF each and is connected through the proposed switch. The trace also shows the voltage supplied to the load (VDD), i.e., the voltage after output power conditioning as described in Section 2.3, and the digital signal (DBG0) that marks the execution of application code immediately after the system boots. The top plot shows the trace for the normally-open variant of the switch, for which the switch starts out open, and the bottom plot shows the trace for the normally-closed variant, in which the switch starts out closed. During the experiment, the switch is closed and opened again, at points indicated respectively by the *first* rising edge and the *last* falling edge of the SWITCH signal. This signal is a manually-controlled GPIO input that is checked by the software on boot to issue the reconfiguration command.

Whenever the switch is closed, both energy banks are part of the energy storage in the system and charge or discharge together. Whenever the switch (to Bank 1) is open, the system can access only Bank 0. In the latter case, the Bank 1 voltage remains unchanged, while Bank 0 charges and discharges and the system operates intermittently. Each cycle, when only Bank 0 is enabled (beginning and end parts of the trace), the system charges for approximately 145 ms and executes for 20 ms, while with both banks enabled (the middle of the trace), the system charges for 5100 ms and executes for 1600 ms.

The top plot in Figure 6.5 highlights the shorter cold start charge time for the normally-open variant. This variant exits the cold start charging phase faster, because it needs to charge a smaller capacitance of Bank 0 only. A shorter cold start allows the system to begin executing useful work sooner. However, the long cold start is still incurred upon *the first* reconfiguration into the larger capacitance, i.e. connecting of the Bank 1 (at time 18 s), since Bank 1 is empty that first time.

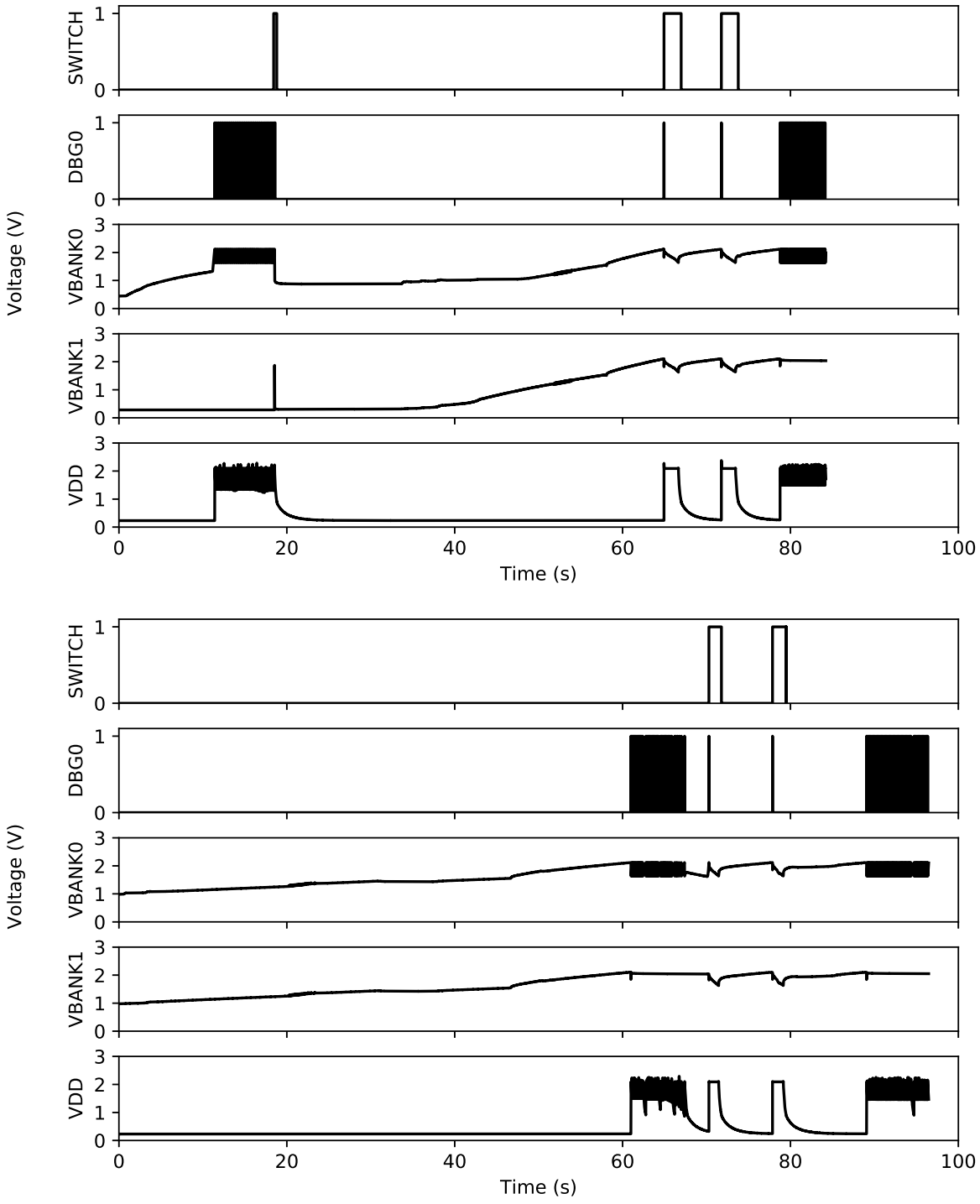


Figure 6.5: Reconfiguration with two capacitor banks and one switch. The system is reconfigured from only Bank 0 ($3 \cdot 100$ μF) to Bank 0 and Bank 1 ($3 \cdot 100$ μF $+ 3 \cdot 7.5$ mF) at the *first* rising edge of the SWITCH signal, and from Bank 0 and Bank 1 back to only Bank 0 at the *last* falling edge. The top trace is for the normally-open variant of the switch and the bottom shows the normally-closed variant.

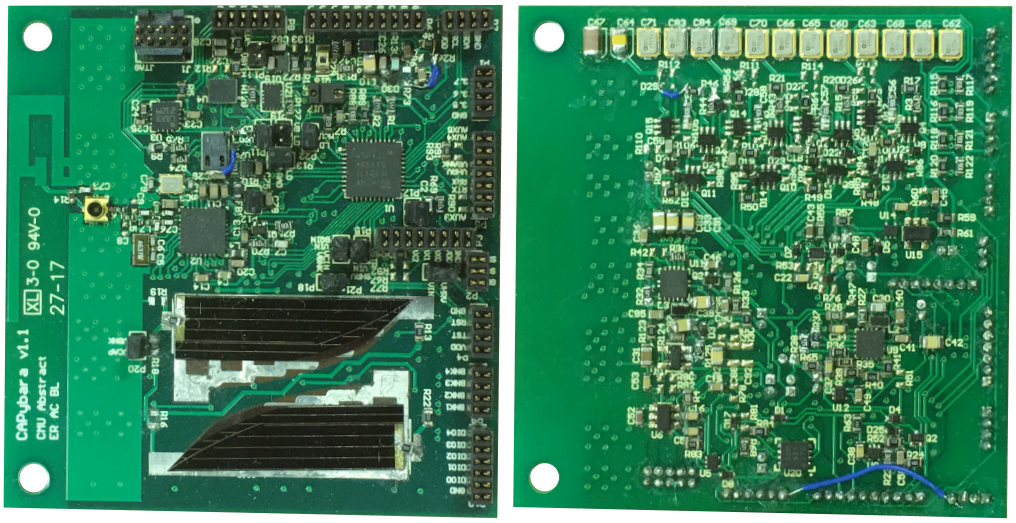


Figure 6.6: Capybara hardware prototype. The solar panels, microcontroller, radio, and, five sensors are on the front side (left), and the power system with five capacitor banks and four switches is on the back side (right).

6.4 Evaluation

We evaluated Capybara to demonstrate that our system enables energy-harvesting applications that detect a higher share of external events and are more responsive. Our experiments on three complete applications running on real hardware compared execution on continuous power (Cont.) to execution on intermittent power under a statically-provisioned fixed energy storage capacity (Fixed) and under two variants of Capybara. Capy-R is a subset of the complete Capy-P. Capy-R excludes `burst` task support and requires recharging after every energy mode reconfiguration.

6.4.1 Hardware platform, applications, and methodology

We built a custom energy-harvesting device with a prototype of the Capybara hardware. The device can be powered by either solar panels or RF and can have up to four energy storage banks implemented using a choice of ceramic, tantalum, or four different models of EDLC supercapacitors. The processor is an MSP430FR5994 microcontroller with ferroelectric nonvolatile memory (cf. Section 2.5). The device features a gesture sensor, a phototransistor, an

accelerometer, a gyroscope, a magnetometer, a barometer, a microphone, and a temperature sensor built into the microcontroller. The device communicates over a Bluetooth Low Energy (BLE) radio. The hardware schematics for this device are provided in Appendix A.

We implemented three complete applications typical of the embedded domain in the Chain programming model, introduced in Chapter 4, and deployed them onto the Capybara board. The applications depend on tasks with distinct temporal energy constraints. We provisioned capacitors for each application through an iterative process. Starting with a pessimistic energy estimate based on load current specified in the datasheets, we ran the task while progressively increasing the capacity on the board until the task completed. In the remainder of this section, we describe each application and the hardware setup used to drive the application with real environmental input.

Wireless Gesture-Activated Remote Control

We implemented a batteryless, wireless, touch-free gesture-activated remote control (GRC) using the APDS-9960 gesture sensor, a photo-transistor and the CC2650 wireless MCU. Each time the MCU turns on, the application samples the photo-transistor to detect if there is an object above the board. If an object is detected, the application activates the APDS sensor for gesture recognition. If the sensor successfully decodes a gesture, the gesture direction is broadcast over the BLE radio.

We implemented two variants of the GRC application. In GRC-Compact the on-time requirements of the application are: (1) acquire one sample from the photo-transistor, (2) keep the APDS sensor on for the minimum duration of a gesture motion (250 ms), and (3) transmit an 8 byte BLE packet. In GRC-Fast, tasks (2) and (3) are joined into a single task with higher on-time requirement equal to their sum. The GRC-Fast variant trades-off peak energy capacity, i.e. device size, to eliminate the recharge latency between gesture recognition and packet transmission. The constraint on off-time in the gesture recognition task is to execute immediately after proximity was detected, before the motion finishes. The

constraint on off-time in the proximity sensing is to minimize inter-sample times to avoid missing proximity events.

For the Fixed-Capacity system, a capacity of 400 μF ceramic + 330 μF tantalum + 67.5 mF EDLC[2] is provisioned to meet the maximum on-time requirement, i.e. (2). For Capybara, two configurations are provisioned, one per energy mode. In both gesture variants Capybara uses a 400 μF ceramic + 330 μF tantalum bank for low energy mode. GRC-Fast provisions 45 mF to meet the high energy requirement for task (2), and GRC-Compact provisions 67.5 mF to satisfy the combined on-time requirements of tasks (2) and (3). In Capy-P, the second bank is pre-charged prior to the energy burst in the gesture task. To produce consistent tap-and-swipe motions for experiments, we use a servo motor to swing a rigid pendulum over the gesture sensor. The board is powered using a harvester built from a voltage regulator and an attenuating resistor that supplies at most 10 mW of power.

Temperature Monitor with Alarm

The Temperature Alarm (TA) senses the temperature of an object (e.g. a pipe) using an external analog sensor (TMP96) and collects a time series of the samples. If the temperature leaves a specified range, the application sends a BLE packet that indicates an alarm and contains the most recent time series. The on-time requirements of the application are: (1) acquire one temperature sample and (2) transmit a 25 byte BLE packet. The off-time constraint of the sampling task is to minimize charging intervals to not miss over- and under-temperature events. The off-time constraint of the transmit task is to send the alarm immediately upon anomaly detection.

The Fixed-Capacity system is provisioned with a single bank of 300 μF ceramic + 1100 μF tantalum + 7.5 mF EDLC capacity.The Capybara systems use one configuration with 300 μFceramic + 100 μFtantalum to support energy mode (1), and another with

[2]Energy capacity is not fungible due to different equivalent series resistance (ESR) of capacitor types that affects the effective extractable energy, as explained in Section 2.2.

1000 μFtantalum + 7.5 mF EDLC to support mode (2). In Capy-P, the second bank is pre-charged prior to the energy burst in the temperature alarm task.

To generate temperature fluctuation, we attach the application's temperature sensor, a 60 W heating element, and a 60 W Peltier cooler to a flat metal heatsink with thermal tape. A control loop on a controller board, which has a temperature sensor attached to the same heatsink, cycles power to the heater and the cooler to maintain the heatsink temperature within a fixed range or push it out of the range to generate an alarm event.

The board is powered via two TrisolX solar panels [176], illuminated with a 20 W halogen bulb with brightness controlled by PWM to 42%. The application on the intermittently-powered board (DUT) is measured with respect to a continuously-powered reference board that runs the same code concurrently and has its sensor attached to the same heatsink.

Correlated Sensing and Report

We implemented a correlated sensing and reporting (CSR) application using a magnetometer and proximity sensor to report the movement of a magnet mounted on our pendulum setup. CSR samples the magnetometer and triggers the proximity sensor to measure distance to the source of magnetic flux. The MCU then lights an LED and sends sensor data by BLE. CSR's tasks are: (1) sample the magnetometer, (2) collect 32 distance samples, (3) power the LED for 250 ms, and (4) send an 8 byte BLE packet. The magnetometer must maintain a consistent sampling frequency to capture field *changes* over time. Tasks (2)-(4) must execute immediately and atomically after a magnetic field event to get accurate distance data and send an alert. The Fixed-Capacity system uses the same bank as GRC-Fast to support (2)-(4). Capybara systems use a 400 μF ceramic + 330 μF tantalum bank for the magnetometer, and the large bank from GRC-Fast for the other mode. The experiment reuses the GRC setup with a magnet attached to the pendulum.

6.4.2 Event detection accuracy

To assess how well applications can detect and react to external events with different power systems, we measure the detection accuracy without and with Capybara. The accuracy in GRC is the number of BLE packets with correctly decoded gesture direction received out of tap-and-swipe motions generated. For TA, accuracy is the number of BLE packets indicating an alarm received from the DUT board out of BLE packets received from the reference board. The CSR accuracy is the number of BLE packets produced to report magnetic events. An event will fail to be reported if the device is charging when the event occurs, or if the device exhausts the energy in its capacitors before the end of an atomic workload (e.g. radio transmission or gesture sensing). Capybara minimizes this cause of undetected events. A secondary cause for failure is the inevitable non-ideal behavior of the hardware that manifests even on continuous power, e.g. BLE packets lost due to interference or gesture sensor inaccuracy. For TA, we only consider events which were successfully reported by the continuously-powered board and count events unreported by the DUT board for any reason as missed. For GRC, we report the imperfect accuracy on continuous power to serve as a point of comparison. Gesture motions are *misclassified* when the proximity detection occurs too late in the pendulum's swing to distinguish the motion direction. *Proximity-only* failures occur when the APDS sensor is activated following a proximity detection but does not report a gesture. Gestures are *missed* if the device is powered off when the pendulum swings by.

Figure 6.7 shows the accuracy each application achieves on an event sequence drawn from a Poisson distribution. The event sequence for TA contains 50 events over 120 minutes, and for GRC and CSR– 80 events over 42 minutes. Fixed-Capacity system correctly detects only 56% of magnetic events, 46% of temperature events, and 18% of gestures, because the charging intervals overlap with events. In contrast, both Capybara variants detect 98% of TA events, at least 89% of CSR events and Capy-P detects 75% and 76% in the two variants of GRC. With Capybara, the device runs the reactive task (i.e., sampling of temperature, proximity, or magnetic field) more frequently, because it charges and discharges only the

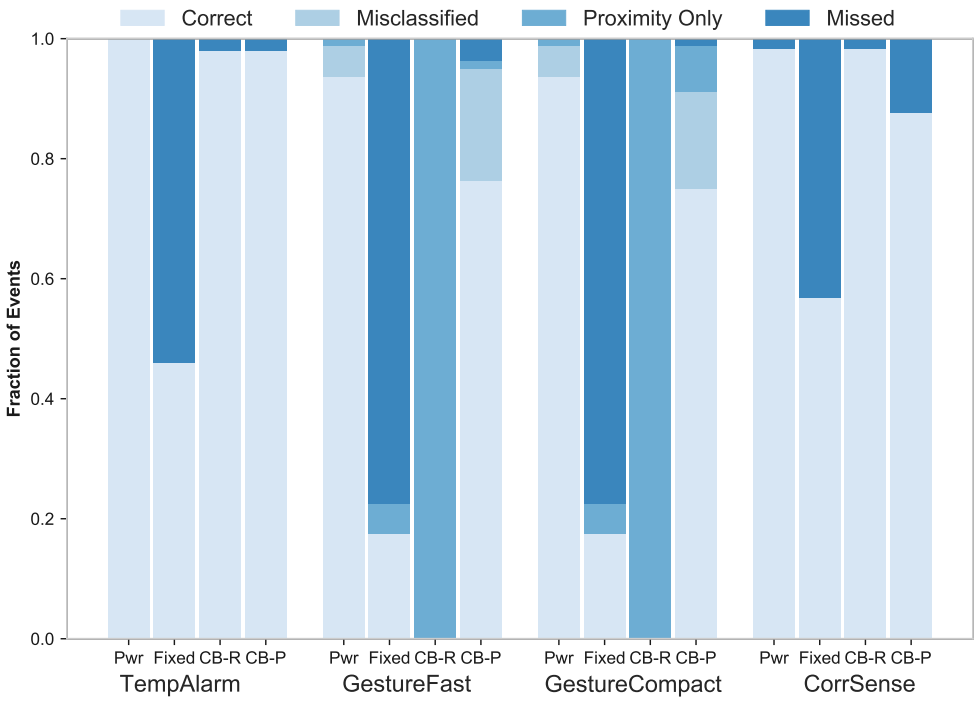


Figure 6.7: Event detection accuracy.

small capacitance between each sample. Capy-R is not suitable for GRC, because it incurs a charging delay between proximity detection and the gesture recognition task, during which the gesture motion completes but the device is off. Capy-P avoids this delay by pre-charging for gesture processing ahead of time.

We assess the sensitivity of accuracy to event inter-arrival times by repeating the measurement for event sequences drawn from Poisson distributions with decreasing means. Figure 6.8 shows that for both applications the farther apart the events are in time the more events are successfully recognized and reported. A lower event frequency, however, does not benefit a Fixed-Capacity system as much as it benefits a Capybara system, because the former exhausts and has to recharge its large fixed capacitor whether or not it had to process an event. For TA, Capy-R achieves an accuracy up to 20% higher than Capy-P on some event sequences, as a result of the lower energy overhead of Capy-R discussed further in Section 6.4.4.

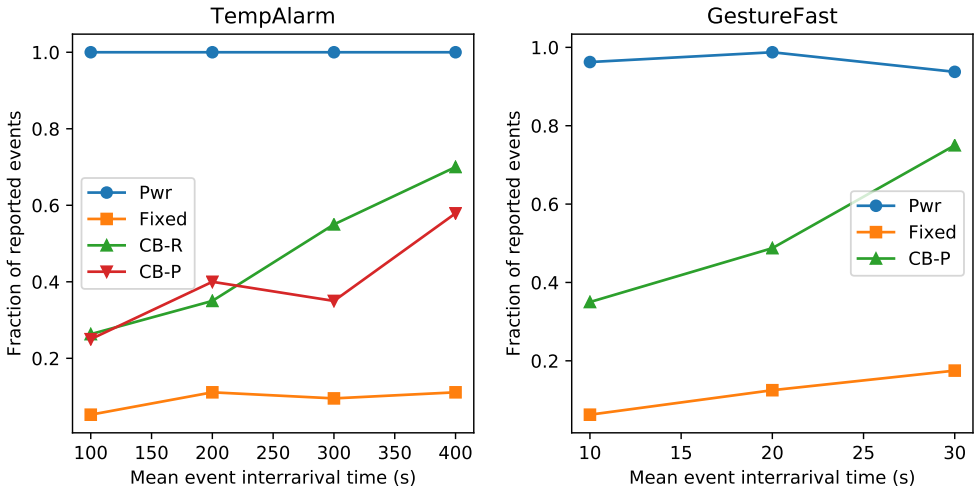


Figure 6.8: Sensitivity of accuracy to event inter-arrival.

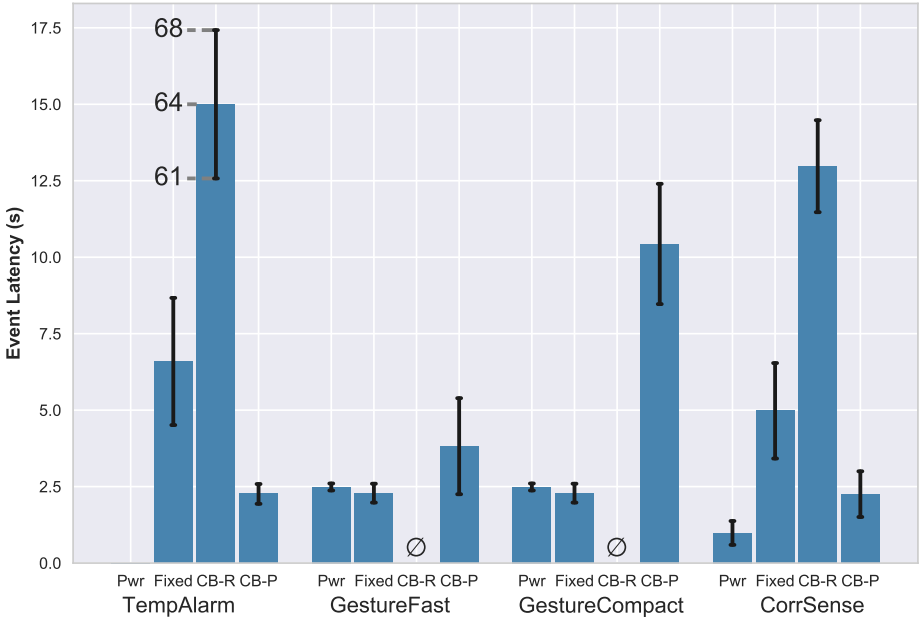


Figure 6.9: Report latency for detected events.

6.4.3 Responsiveness

Since all of our applications are latency-sensitive and react to an event by sending a BLE radio packet, we measured the latency between when the event occurs and when the packet is received on a laptop. For TA, latency is the time difference between the packets from the reference board and the DUT board that correspond to the same temperature alarm event.

For GRC and CSR, latency is the time between the pendulum actuation command and the BLE packet reception.

Figure 6.9 shows the latency of each event that was successfully reported in the experiment of Section 6.4.2. For GRC, while Fixed-Capacity reports few events, the ones it does report, are reported as quickly as on continuous power, because there is no charging between event detection and radio transmission. For TA and CSR, under Fixed-Capacity some packet transmissions fail on first attempt due to insufficient energy and are re-transmitted after a charging interval, which raises the average latency across all events.

The advantage of Capy-P over Capy-R in terms of latency is exemplified by TA. All systems need to charge a large capacity before they can transmit the packet, but only Capy-R charges on the critical path, increasing the latency by the charge time (64 s). By charging the capacitor ahead of time, Capy-P reduces the latency to 2.5 s. By the same principle, in GRC, Capy-P successfully eliminates the charging latency between proximity event and gesture recognition, but not necessarily between gesture recognition and packet transmission. The end-to-end latency differs between the two variants of GRC that demonstrate the trade-off between latency and the maximum required capacity. In all cases, the provisioning is for the average case energy cost, not the worst-case, which causes some events to require charging, despite pre-charge. The increased latency is incurred for 7% of reported events in GRC-Fast and 54% in GRC-Compact, which is reflected in the *average* latency in Figure 6.9. As explained in Section 6.4.2 Capy-R reports no events for GRC.

6.4.4 Reactivity

In sensing applications that record time series, the times at which the samples are sensed matter as much as their total count. For example, batches of back-to-back samples are less valuable than evenly spaced series. In this experiment we quantify improvements in sampling quality achievable with Capybara, by measuring the intervals between temperature samples in the TA application. Figure 6.10 shows the distributions of inter-sample times for three

systems when the input is the same sequence of 20 temperature alarm events. The sub-second intervals between back-to-back samples are colored gray to indicate their limited utility. The remaining inter-sample intervals are broken down into ones during which one or more events occurred and were (necessarily) missed (red), and those without any events (green).

A Fixed-Capacity system forces the application to sample in batches of as many samples as can be taken on the capacity provisioned for the largest atomic workload (i.e. radio packet transmission). An alternative implementation might put the processor to sleep in between samples to introduce a delay. However, the batches will still be separated by the long charge time of the large capacitor, because it will discharge during sampling despite the sleep mode, due to the power overhead of the power system that remains on. With Fixed-Capacity, most non-back-to-back inter-sample intervals are long (110-250 s) and cause the missed events reported in Section 6.4.2.

With Capybara, the large capacity needs to be charged only as many times as there are temperature events to report (32 out 1738 inter-sample periods). All other times, the samples are either separated by the (shorter) charge time of the smaller capacitor (1.5-4 s) or are back-to-back. The back-to-back samples still occur because the small bank is over-provisioned for the temperature sample, since the Capybara power system requires the bank to be no smaller than that needed by the output booster to start up.

Compared to Capy-P, the undesirable (but unavoidable) long inter-sample times with Capy-R have a smaller impact on accuracy, as suggested by the share of long intervals that caused an event miss (23 out of 28 for Capy-P vs 3 out of 31 for Capy-R). This decrease in event misses is explained by a shorter mean charge time (84 vs 220 s), which is a consequence of a subtle power system effect that boosts the charging efficiency of Capy-R relative to Capy-P, for the same energy capacity and input power.

The increase in charge time in Capy-P is a consequence of the lower voltage at which charging has to start for Capy-P compared to Capy-R. The charging starts at a lower voltage for Capy-P, because the discharge starts at a lower voltage. The full-bank voltage is lower

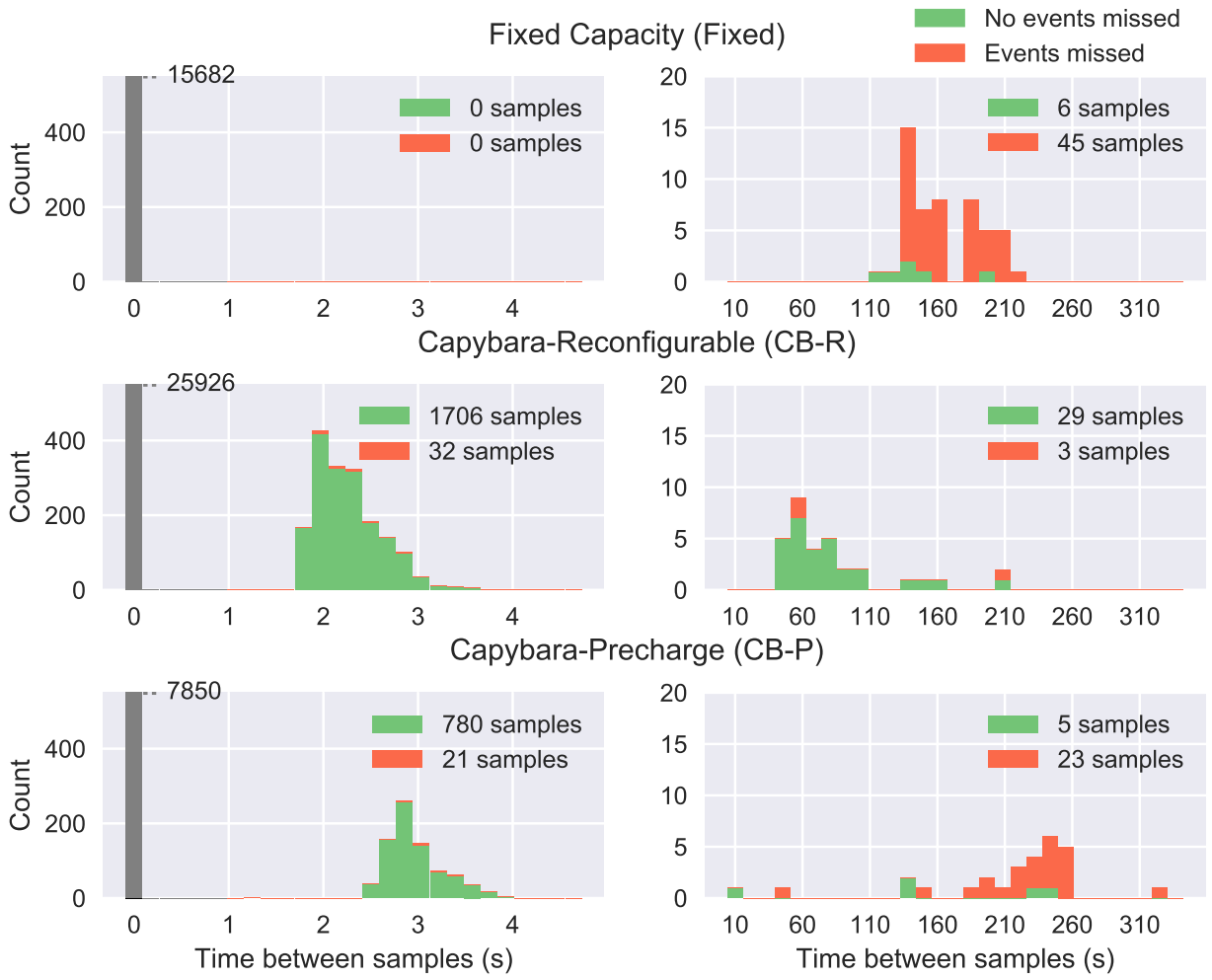


Figure 6.10: Distribution of times between samples in TempAlarm application. Total counts of *non-back-to-back* samples show that sampling is *denser* with Capybara compared to a fixed capacity.

for Capy-P, because Capybara can *pre*-charge a bank only to a strictly lower voltage than it can charge a bank to (by approximately 0.3v), which is a limitation of our particular implementation of the switch circuit. This disadvantage of Capy-P relative to Capy-R is also reflected by the drop in accuracy in Figure 6.8, but is compensated by the order of magnitude improvement in latency in Figure 6.9.

6.4.5 Characterization

Capybara power system hardware is intended to be integrated onto the board of an energy-harvesting device alongside its MCU and sensors. On our 6x6 cm prototype board (with an unoptimized layout), solar panels occupy 700 mm^2, the power conditioning circuits occupy 640 mm^2, and one reconfiguration switch occupies 80 mm^2 with support for both NO and NC configurations and other debugging capabilities that can be omitted from a release version. In our prototype, the switch uses a 4.7 μF latch capacitor and retains state for approximately 3 minutes. The detailed schematics are listed in Appendix A.

6.5 Summary

To handle environmental triggers responsively, software tasks on energy-harvesting devices place temporal constraints on energy availability. The software desires control over when the power system of the device provides energy to the load and how much energy it provides at a time. Both of these parameters are a function of the energy storage capacity of the energy buffer on the device. We observed that a system with a fixed-capacity energy buffer cannot satisfy both capacity and temporal constraints, due to the inverse relationship between capacity and charge time.

Capybara is the first system with a software interface for expressing task energy requirements as energy modes. Capybara provides a hardware mechanism for reconfiguring energy storage capacity and pre-charging capacitors for on-demand energy bursts, and a runtime system that reconfigures the hardware energy capacity dynamically. Our evaluation of Capybara in a solar-powered energy-harvesting device showed that reconfigurability improves application responsiveness and event detection accuracy.

Chapter 7

EDB: An Energy-interference-free Debugger

Intermittence can cause software that is correct on battery-powered platforms to misbehave on energy-harvesting platforms. Intermittence-induced jumps back to a prior point in an execution inhibit forward progress and may repeatedly execute code that should not be repeated. Intermittence can also leave memory in an inconsistent state that is impossible in a continuously-powered execution [101, 140]. These failure modes represent a new class of *intermittence bugs*. To avoid intermittence-related malfunction, code must *correctly* leverage non-volatile memory.

Implementing an intermittence-safe runtime system, such as Chain (cf. Chapter 4), or a custom bare application without any system, requires the programmer to understand, find, and fix intermittence bugs. In fact, an early implementation of Chain contained an intermittence bug in the code for self-channel management (cf. Section 4.2) that manifested rarely and *only* while running with a energy-harvesting power supply. A system abstracts some complexity of intermittent execution from the applications and minimizes the code vulnerable to intermittence bugs to the code in the implementation of the runtime of that system. However, some programmers may choose not to delegate anything to a system, in

order to aggressively optimize for performance with application-specific tricks. In this case, the code of the entire application becomes vulnerable to intermittence bugs.

To diagnose intermittence bugs in their code, programmers need to monitor system behavior, observe failures, and examine internal program state. Unfortunately, this simple debugging methodology is unusable for intermittence bugs because existing tools power the target device, masking intermittent behavior. Programmers are left with an unsatisfying dilemma: to use a debugger to monitor the system and never observe a failure; Or to run without a debugger and observe the failure but gain no insight into the system necessary for understanding the bug.

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

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

In the next section we study an example of a bug induced by intermittent execution

and the difficulties of diagnosing this bug with existing debugging tools. In Section 7.2 we propose EDB and present the debugging primitives it supports. We discuss our prototype in Section 7.3 and evaluate it on a set of debugging tasks in Section 7.4. Section 7.5 summarizes the proposed debugging support.

7.1 Intermittence and energy-interference

This section provides background on the challenges presented by intermittent energy-harvesting devices and illustrates that existing approaches to debugging fail to address these challenges. As a basis for our discussion, we assume an intermittent system that executes a C program that takes longer than a single charge-discharge execution cycle to complete. We assume our device has a mixture of volatile registers and memory, as well as some non-volatile memory. We further assume a checkpointing mechanism that periodically collects a checkpoint of volatile execution context (i.e., register file and stack) like prior work [141, 111, 76]. Note that this checkpointing assumption simplifies the exposition, but our ideas and prototype apply to task-based systems, such as Chain introduced in Chapter 4.

7.1.1 Bugs induced by intermittent execution

Power intermittence complicates understanding and debugging a system, because the behavior of an intermittent system is closely linked to its power supply. Figure 7.1 illustrates how intermittence induces bugs even with runtime support for checkpointing volatile state into non-volatile memory [141, 111, 76]. The code manipulates a linked-list in non-volatile memory using `append` and `remove` functions. A continuous execution completes the code sequentially, as expected. An intermittent execution, however, is not sequential. In the leftmost trace, a checkpoint happens to be collected at the top of the `while` loop and the processing continues until power fails at the indicated point. After the reboot, execution resumes from the checkpoint. This sequence of events later leads to undefined behavior. The

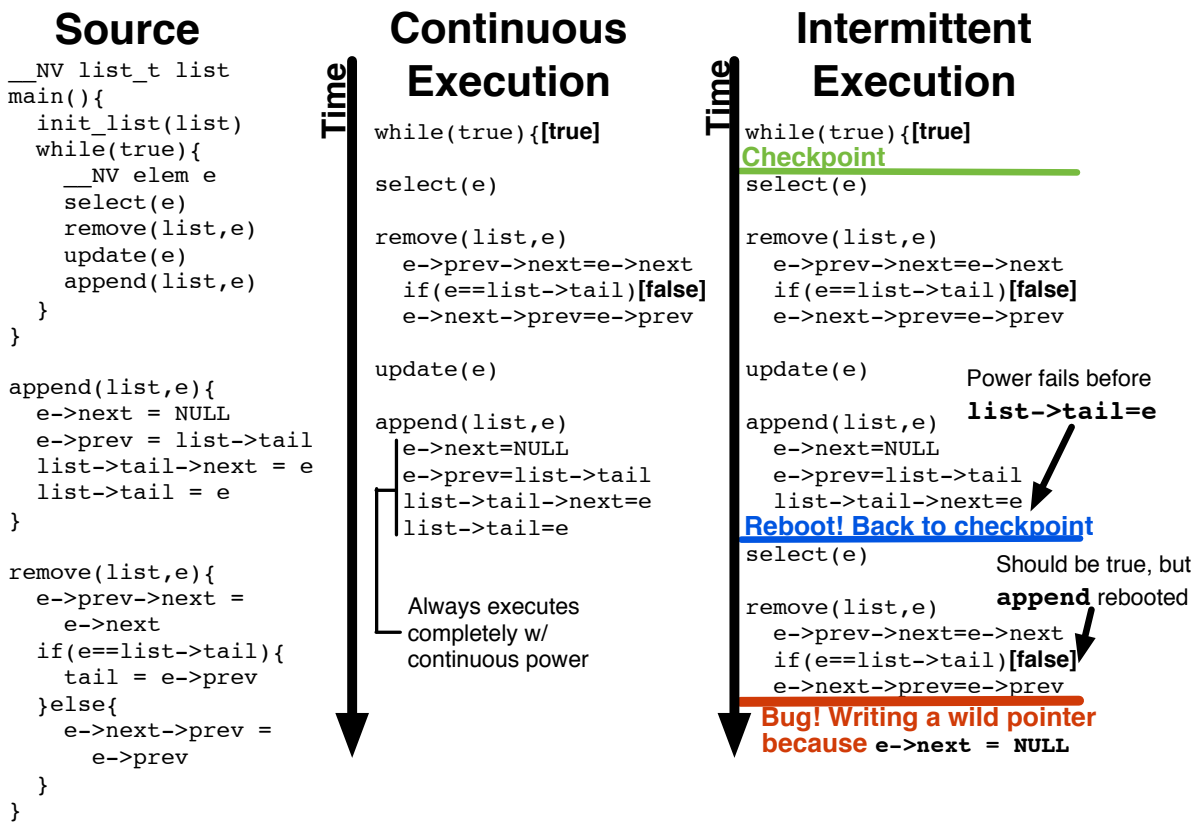


Figure 7.1: An intermittence bug. The linked-list stays correct with continuous power but is corrupted and leads to a wild pointer write with intermittent power.

execution violates the pre-condition assumed by `remove` that only the `tail`'s `next` should be `NULL`. The reboot interrupts `append` before it can make node `e` the list's new tail but after its `next` pointer is set to `NULL`. When execution resumes at the checkpoint, it attempts to remove node `e` again. The conditional in `remove` confirms that `e` is not the tail, then dereferences its `next` pointer (which is `NULL`). The `NULL` `next` pointer makes `e->next->prev` a wild pointer that, when written, leads to undefined behavior. This `NULL` pointer dereference cannot happen in a continuous execution and is an example of an *intermittence bug*.

7.1.2 Energy-interference during debugging

Debugging intermittence bugs, like the one in Figure 7.1, is difficult using existing tools. Conventional debuggers supply power to the device-under-test (DUT), which precludes ob-

servation of a realistically intermittent execution. Dedicated debugging equipment, like a JTAG [1] debugger, offers visibility into the device's state but is not useful because it provides continuous power and masks intermittence. JTAG power isolator devices [152] exist to decouple debug host power rails from DUT power rails, but these do not help with intermittence debugging, because the JTAG protocol fails if the DUT powers off. The inapplicability of JTAG precludes interactive debugging (e.g., like GDB or LLDB) for intermittent executions. Using a JTAG debugger for the code in Figure 7.1 would only ever result in the non-failing, continuous execution shown in the middle; the programmer would never see unexpected behavior.

One mostly energy-interference-free tool that can be used for debugging intermittent systems to a limited extent is a mixed-signal oscilloscope. An oscilloscope can collect an energy trace by probing DUT's power system and I/O lines. Unfortunately, an oscilloscope provides no insight into the internal state of the software running on the DUT. Oscilloscope-based debugging is not the interactive process familiar to most programmers. Moreover, oscilloscopes cost thousands of dollars, making them inaccessible to most developers. Using an oscilloscope to debug the code in Figure 7.1 would permit reproducing the problematic intermittent execution. However, the oscilloscope would not help relate changes in the device's energy state (which it can observe) to the software events that change device state and memory (which it cannot observe). That absent connective information is the key to understanding the failure in this code.

An alternative approach to diagnosing an intermittence bug is to directly write debugging instrumentation code into an application to trace certain program events. In embedded systems, a popular *ad hoc* approach is to toggle an LED at a point of interest. LED-based tracing does not work in energy-harvesting devices, because LEDs are power-hungry and their energy use changes the execution's behavior. As a case in point, it is prohibitively expensive to use an LED to indicate when a WISP energy-harvesting device [149] is executing code, rather than just charging. Powering an LED increases the WISP's current draw by five

times, from around 1 mA to over 5 mA.

Another tracing strategy is to manually instrument code to log program events to non-volatile memory. The resulting trace lacks information about the energy level, unless the developer also spends time, energy, and an ADC channel to log the DUT's energy state. Non-volatile tracing also consumes precious non-volatile storage space. To spare consuming non-volatile storage space, a programmer may write code to stream the event log to a separate, always-on system (e.g., via UART). Powering and clocking an I/O peripheral to transfer the log is expensive in time and energy and adds considerable complexity to code.

All of these instrumentation-based approaches change the point in the program at which energy is exhausted. As a result, the act of debugging alters the intermittent behavior of the application. Furthermore, the value of tracing depends on the events which the programmer decides to trace. To understand the intermittence bug in Figure 7.1, the programmer needs to log particular events in the `append` and `remove` routines. The bug manifests as a wild pointer write and may appear to crash inexplicably, in code far from the either of those routines, giving little to suggest that `append` and `remove` contain the culpable code.

Energy-interference and lack of visibility into intermittent executions makes debugging tools designed for battery-powered embedded devices inadequate for intermittence debugging.

7.2 Energy-interference-free debugging

EDB is an energy-interference-free platform for intermittence debugging that addresses the shortcomings of existing approaches described in Section 7.1. This section describes the high-level capabilities and functionality of EDB, while Section 7.3 describes the co-designed hardware and software implementation that make EDB energy-interference-free.

Figure 7.2 illustrates EDB's functionally. At the top are EDB's *capabilities* that together support the *debugging primitives* at the bottom. The functionality is organized into two

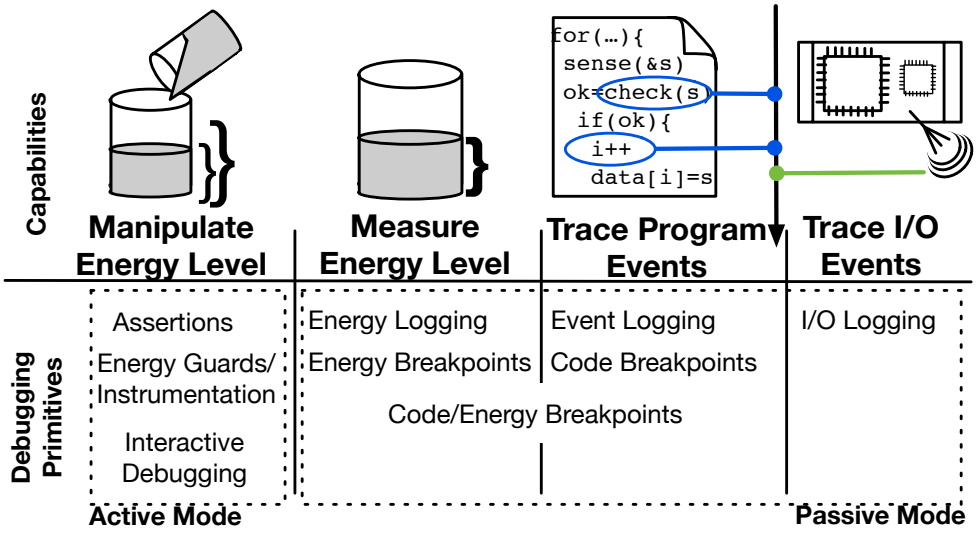


Figure 7.2: EDB's features and supported debugging tasks.

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

7.2.1 Passive mode operation

EDB's passive mode operation is built around the three right-most components at the top of Figure 7.2. The developer gets the ability to acquire a set of streams and relay them to the host workstation continuously in real-time without active involvement from the target whether it is on or off. Streams instrumental for debugging are the energy level, I/O events on wired buses, messages exchanged over RFID protocol, and program events marked by *watchpoints* in application code. A key advantage of EDB is the ability to gather this data concurrently, letting the developer correlate changes in system behavior with changes

in energy state. That correlation is important during development, but, as Section 7.1 describes, difficult or impossible using existing techniques.

7.2.2 Active mode operation

The capability to manipulate the amount of energy stored on the target device underlies EDB's active mode of operation. Active mode frees debugging tasks from the constraint of the small energy store of the target device. EDB can *compensate* for the energy consumed by a debugging task that involves a costly operation on the target, such as interacting with the programmer, executing arbitrary debug code, or conveying state to the debugger. Before performing an active task the energy on the target device is measured and recorded. While the active task executes, the target is continuously powered. After performing the active task, energy on the target device is restored to the level measured before the active task. Continuously powering active tasks enables them to consume arbitrary amounts of energy. Energy compensation provides the illusion of an unaltered, intermittent execution to the application. Without this support, debugging tasks that require considerable involvement from the target are out of reach.

7.2.3 Energy-interference-free debugging primitives

Using the monitoring and manipulation capabilities described so far, EDB creates a toolbox of energy-interference-free debugging primitives. EDB brings to intermittent platforms familiar debugging techniques that are currently confined to continuously-powered platforms. New intermittence-aware primitives are introduced to handle debugging tasks that arise only on intermittently-powered platforms.

Code and energy breakpoints

EDB implements three types of *breakpoints*. A *code breakpoint* is a conventional breakpoint that triggers at a certain code point. An *energy breakpoint* triggers when the target's energy

level is at or below a specified threshold. A *combined* breakpoint triggers when a certain code point executes and the target device's energy level is at or below a specified threshold. Breakpoints conditioned on energy level can help catch energy leaks due to unexpected code paths. They initiate an interactive debugging session precisely in problematic iterations when more energy was consumed than expected or when the device is about to brown-out.

Keep-alive assertions

EDB provides support for using familiar *assertions* on intermittent platforms. When an assertion fails, EDB immediately tethers the target to a continuous power supply to prevent it from losing state by browning out. This *keep-alive* feature turns what would have to be a postmortem reconstruction of events into an investigation on a live device. A postmortem analysis is limited to scarce clues in a tiny ad hoc "core dump" that a custom fault handler can manage to save into non-volatile state before the target runs out of energy and resets. The clues available in the interactive session that is automatically opened by EDB for a failing `assert` include the entire live target address space and I/O buses to peripherals.

Energy guards

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

(e.g. `printf`, LED) can be costly enough to repeatedly deplete the target energy supply and prevent forward progress.

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

Interactive debugging

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

7.3 Hardware-software implementation

EDB's capabilities and debugging primitives are implemented in custom co-designed hardware and software. Figure 7.3 shows a block diagram of EDB (depicted in green) connected to an RF energy harvesting target (in purple). The labeled wires are physical connections between EDB and the target that carry both analog and digital signals and are exposed through header pins. Our prototype hardware board can connect to any energy-harvesting

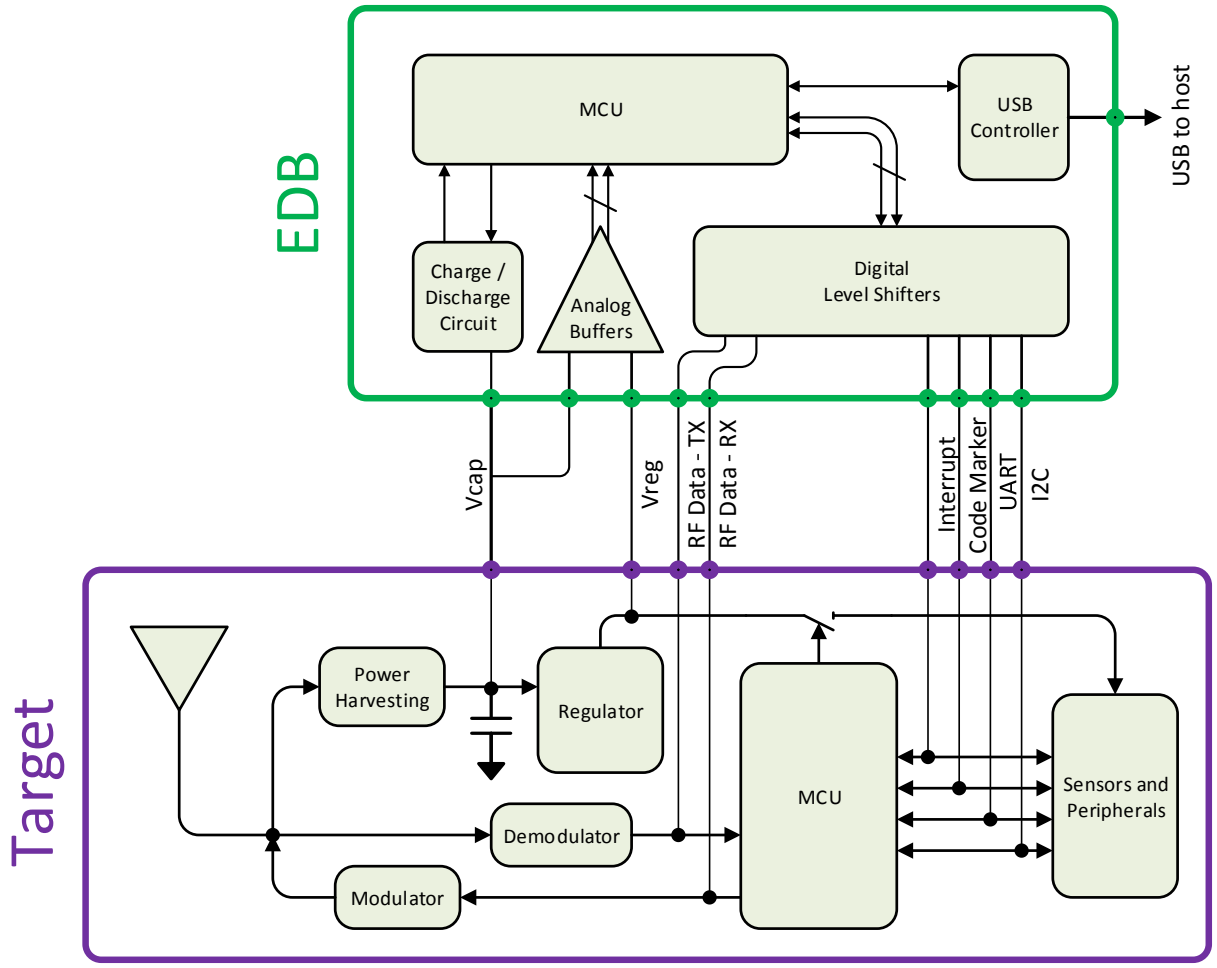


Figure 7.3: Block diagram of EDB connected to an RF energy-harvesting target. All signal lines are buffered to minimize energy interference. A charge/discharge circuit controls the voltage on the target's energy storage capacitor.

device with a microcontroller and a capacitor. To support a new device, the applicable physical connections from Figure 7.3 must be wired and target-side EDB software library of 1200 lines of C code must be ported to the new architecture.

7.3.1 Energy level monitoring

Energy-interference-free measurement of the target's energy level is essential to EDB's passive mode operation (Section 7.2.1). To measure the energy level in the energy buffer on the device, EDB uses two physical connections, V_{cap} and V_{reg}, to the target device's energy

storage capacitor and its regulated power line, respectively. These signals pass through a dual high impedance, unity gain instrumentation amplifier to minimize leakage current from the target to EDB. These analog voltages are digitized by an analog to digital converter (ADC) and logged or used internally for debugging tasks. While it is *possible* for energy harvesting devices to measure their stored energy levels, doing so uses energy, perturbing the energy state being measured and altering the intermittent behavior of the software.

7.3.2 Energy manipulation and compensation

Energy manipulation and compensation are the basis for EDB's active mode of operation (Section 7.2.2). EDB has a custom circuit consisting of a low pass filter, keeper diode, and GPIO pins that can charge and discharge the target's energy storage capacitor. This circuit is designed to prevent it from loading down or trickle charging the target while inactive.

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

7.3.3 I/O monitoring

EDB is designed to enable passive monitoring of arbitrary I/O and attached peripherals, such as sensors, communication buses, and radios. These digital signals (labeled RF Data Tx/Rx, UART, and I2C in Figure 7.3) connect to a digital buffer and level shifter. We use an extremely low-leakage buffer to prevent leakage current from the target to EDB, and we use the level shifter to match the buffer's voltage level to the target device's voltage level and avoid current flow between them.

Note that while the target device has an on-board regulator, the V_{reg} line may drop below its specified, regulated value during a power failure on the target device. We address the V_{reg} drop with a simple tracking circuit consisting of an analog buffer to keep the level shifter at the target's voltage. This circuit is important because too large a mismatch (i.e., over +/-0.3V [172]) activates the voltage protection diodes in the target's MCU, which perturbs the target's power state.

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

7.3.4 Program event monitoring

EDB can track program execution using the *Code Marker* connections in Figure 7.3. To monitor a code point, the programmer inserts a watchpoint with a unique identifier at that location. EDB's target-side software encodes this identifier onto the Code Marker lines when the program counter passes over the code point. On the debugger-side, transitions on the Code Marker lines are captured and decoded into watchpoint identifiers. EDB can simultaneously monitor $2^n - 1$ distinct watchpoints, where n is the number of GPIO lines allocated to the Code Marker function.

Monitoring program events using EDB is *practically* energy-interference-free. The main energy cost is the target device holding a GPIO pin high for one cycle to encode each traced code point as it executes. We measured the cost of this GPIO-based signaling to be negligible using the methodology described in Section 7.4. Without EDB, monitoring has a prohibitive cost in code, memory space, and energy. With EDB, events are not only logged without these costs, but also correlated with energy state into a multifaceted profile.

`libEDB` API	**Debug** Console Commands
`assert`(expr)	`charge`|discharge *energy_level*
`break`|watch_point(id)	`break`|watch `en`|dis *id* [*energy_level*]
`energy`_guard(begin|end)	`trace` {*energy,iobus,rfid,watchpoints*}
`printf`(fmt, ...)	`read`|write *address* [*value*]

Table 7.1: Developer's interfaces into EDB.

7.3.5 Developer's interface into EDB

EDB's debugging primitives are accessible to the end-user through two complimentary interfaces: the `libEDB` API and the host console commands. Both are listed in Table 7.1. The `libEDB` library statically links into the application and exports C macros for inserting assertions, breakpoints, watchpoints, energy guards, and energy-interference-free `printf` calls into the application code. Internally, the library implements the target-side half of the protocol for communicating with the debugger over a dedicated GPIO line and a UART link, which includes routines for reading from and writing to target address space.

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

7.3.6 Hardware and software components

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

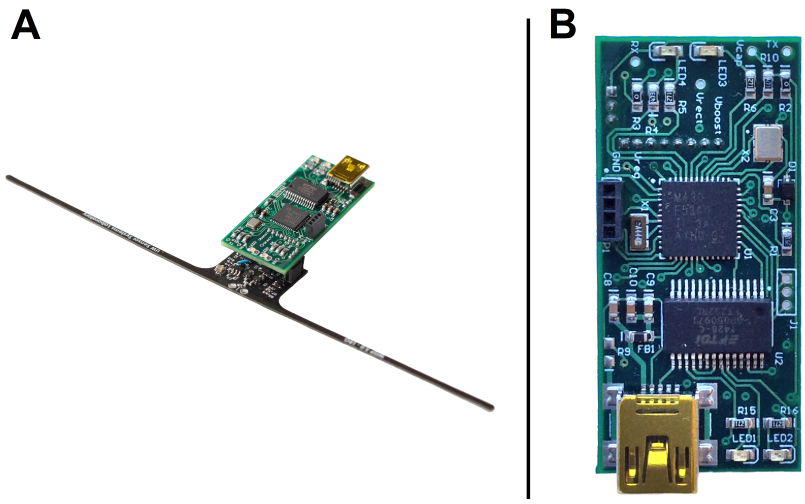


Figure 7.4: EDB, energy-interference-free system for monitoring and debugging energy-harvesting devices, attached to a WISP [149] (purple PCB) in Panel A and shown in detail in Panel B.

7.4 Evaluation

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

7.4.1 Experimental setup

Our experimental setup consists of the EDB prototype board, a target energy-harvesting device and wireless energy source, and measurement instruments. The EDB board is always connected to a development workstation via USB and to the target device through a dedicated header. EDB is controlled from the workstation programmatically or manually through the console described in Section 7.3.5.

Our target device is a Wireless Identification and Sensing Platform (WISP) version 5 [149]. The WISP has a 47 μF energy storage capacitor, a turn-on threshold of 2.4 V, a

brown-out threshold of 1.8 V, and an active current of approximately 0.5 mA at 4 MHz. The WISP is intermittently powered by RF radiation from an Impinj Speedway Revolution RFID reader. The reader is configured to continuously inventory tags at a transmit power of up to 30 dBm using the SLLURP toolset [19], and its antenna is placed at a distance of 1 m from the WISP. The amount of harvestable energy is inversely proportional to this distance. In our evaluation, we ran a collection of different software applications on the target device. We used a custom test program that manipulates non-volatile linked-list, and another that generates a persistent list of Fibonacci numbers. We also used two real applications, including the official WISP 5 RFID firmware, and a machine-learning-based activity recognition application from prior work [101].

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

7.4.2 Energy-interference

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

Current flow over electrical connections

Any current that flows between the target and the debugger through the connections in Figure 7.3 may inadvertently charge or discharge the capacitor on the target. As discussed

in Section 7.3, EDB's circuits are designed to minimize the amount of current that can flow across any of these connections to or from the target's power supply. Imperfections in components, such as reverse leakage current in the diodes, inevitably cause some current to flow. We measured the maximum possible current flow over each connection and verified that in the absolute *worst-case*, when all lines are active, the effect it can have on the target power supply is negligible.

We used a source meter to apply a voltage to the driving endpoint of each connection and measure the resulting current. We measured each connection with digital logic endpoints in both *LOW* and *HIGH* states by applying either 0 V or 2.4 V, which is the maximum voltage that can arise on any of the connections. We measured analog endpoints under the worst-case condition of 2.4 V. The sum of the worst-case current flow in either direction across all connections is 0.85 μA or 0.2% of the typical active mode current consumption of the MCU in our target device. Table 7.2 characterizes the energy interference, showing a breakdown of worst-case current by connection, driving endpoint, and logic state.

Accuracy of manipulating target energy level

To support debugging tasks presented in Section 7.2, EDB needs to save, change, and restore the amount of charge in the target's storage capacitor. A large discrepancy between the saved and restored level can undermine the illusion of an unaltered intermittent power supply that EDB presents to the target software. We quantified this energy level discrepancy, ΔE, by measuring the voltage on the capacitor before and after a save-restore operation and applying the expression for the energy stored in a capacitor: $\Delta E = \frac{1}{2}C(V^2_{\text{restored}} - V^2_{\text{saved}})$. This quantity was then expressed as a percentage of the maximum energy storable on the target: $\Delta\hat{E} = \Delta E/(\frac{1}{2}CV^2_{\text{max}})$, where $V_{\text{max}} = 2.4$ V.

We used the charge/discharge commands introduced in Section 7.3.5 to run 50 trials of a save-restore operation. For each trial, we set an energy-breakpoint at 2.3 V, charged the target capacitor to 2.4 V, waited for the target execution to be interrupted by the breakpoint,

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

Table 7.2: Measured worst-case current that can flow over electrical connections between the target device and EDB.

and then resumed the target. Table 7.3 summarizes two independent sets of measurements of $\Delta V = V_{\text{restored}} - V_{\text{saved}}$, ΔE, and $\Delta\hat{E}$: one from our oscilloscope and one from EDB's ADC. The accuracy of EDB's save-restore mechanism, $\Delta\hat{E}$, in our prototype implementation of EDB is, on average, 4.34% of the target's $47\mu F$ energy storage capacitor. Our prototype's energy level discrepancy is small enough that it is unlikely to be problematic. We expect that further software optimization will leave a discrepancy closer to the accuracy limit imposed by EDB's ADC. A 12-bit ADC with effective resolution of approximately 1 mV imposes a theoretical lower bound on $\Delta\hat{E}$ of 0.08%.

7.4.3 Debugging capabilities

We now illustrate the new capabilities that EDB brings to the development of intermittent software by applying it to debugging tasks that are particularly difficult to resolve using state-of-the-art tools. Energy-harvesting applications in the following case studies execute

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

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

Table 7.3: Accuracy with which EDB saves and restores energy level quantified as the difference in capacitor charge before saving and after restoring and measured using either an external oscilloscope or the internal ADC in EDB.

intermittently and keep state in non-volatile memory to make progress without relying on a runtime checkpointing system. A reboot causes execution to return to the program entry point (i.e., `main`).

Detecting memory corruption early

Memory corruption due to incorrect pointer arithmetic or a buffer overflow is a frequent yet difficult problem to debug. The root cause is obscured behind symptoms that are far from the offending memory write in time and in space. Memory corruption induced by intermittence is harder to diagnose still, because it is not reproducible in a conventional debugger as discussed in Section 7.1.2. This section studies an application that fails due to an intermittence-induced memory corruption and demonstrates how EDB's support for assertions exposes the root cause.

Application. The code listed in Figure 7.5 maintains a doubly-linked list data structure in non-volatile memory. On each iteration of the main loop, a node is appended to the linked list if the list is empty or removed from the list otherwise. The node is initialized with a pointer to a buffer in volatile memory. This pointer is retrieved when the node is removed from the list and data is written to the buffer it points to.[1] For illustrative purposes, at the beginning and end of the loop iteration, the code toggles a GPIO pin to indicate that the main loop is running.

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

Symptoms. After having run on harvested energy for some amount of time, the GPIO pin indicating main loop progress stops toggling. The real oscilloscope trace in the top of Figure 7.6 shows an early charge-discharge cycle when the main loop is still executing and a later one when it no longer does. After the main loop stops executing, the application never returns to normal, including after reboots on subsequent charge-discharge cycles. The only way to recover is to re-flash the device. Note that the failure problem never occurs when the device runs on continuous power.

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

EDB's intermittence-aware `assert` mechanism is designed to tackle this class of bugs. We assert the invariant that the tail pointer points to the last element in the list as shown in Figure 7.5 and run the program on harvested energy with EDB attached. EDB's console reports the assertion failure, halts the program, starts continuously powering the target, and opens an interactive debug session. This sequence is captured in the bottom oscilloscope trace in Figure 7.5. The discharge cycle on the right is the one during which the assert fails at instant 1 and the capacitor voltage is seen rising to the level of the tethered power supply.

In the interactive debug session summarized on the right in Figure 7.5, we check the device's internal state using EDB's commands for inspecting target memory. The tail pointer points to the penultimate element instead of the last one, which is a consequence of an `append` interrupted by intermittence. Because of this inconsistency, the `else`-clause in the `remove` function would dereference a `NULL` pointer, read the buffer pointer from an invalid

Application Code

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

Debug Console

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

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

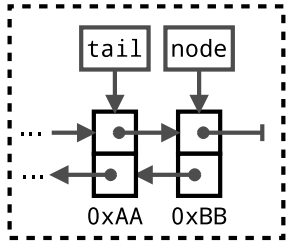


Figure 7.5: An intermittence bug that corrupts memory, diagnosed using EDB's intermittence-aware `assert` (left) and interactive console (right).

location, and cause `memset` to write to a wild pointer and corrupt non-volatile state beyond recovery. The `assert` and the interactive session uncovered the precise inconsistency in the data structure before any of these confounding consequences could take place.

Instrumenting code with consistency checks

To aid in debugging, applications often have separate *debug* and *release* build configurations. A debug build includes *instrumentation* code such as checks for consistency of data structures or array bounds. On continuously-powered platforms the convenience of the debug build comes at the cost of slower execution speed, higher memory usage, and higher energy consumption. However, on intermittently-powered platforms, the effect is more dire: the energy overhead of instrumentation can render an application non-functional by preventing it from making *any* forward progress, for the reason explained in Chapter 5. Yet, instrumented energy-harvesting applications must be run on harvested energy to diagnose intermittence-induced bugs, since these bugs are invisible while the device is continuously powered. In this case study we demonstrate how an application can be instrumented with debug code of arbitrary energy cost using EDB's energy guards.

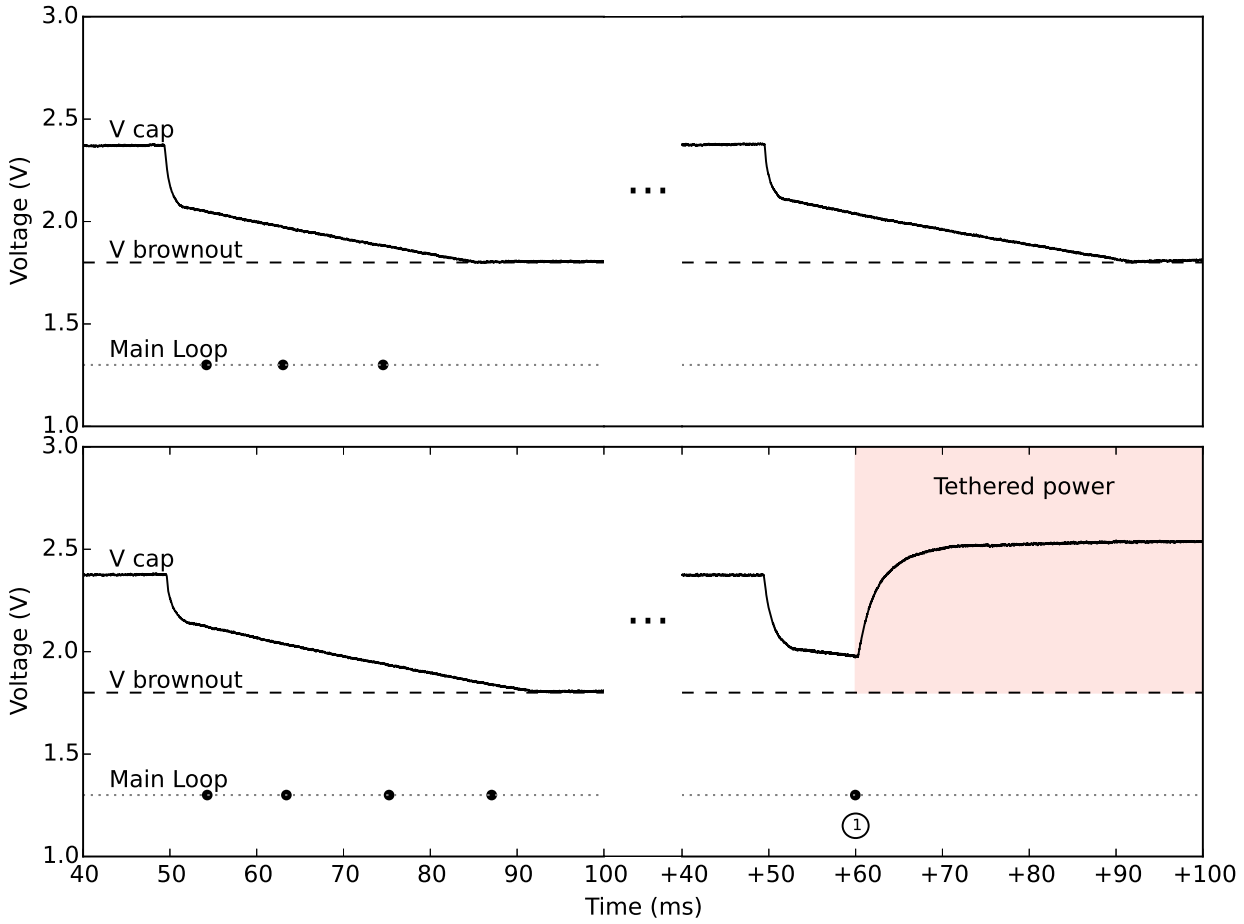


Figure 7.6: Oscilloscope trace of a memory-corrupting intermittence bug and EDB's intermittence-aware `assert` in action. Without the `assert` (top) the main loop runs at first (left) but mysteriously stops running in later discharge-cycles (right). With the `assert` (bottom), when it fails at instant 1, EDB halts the device and tethers it to a continuous power supply.

Application. The code in Figure 7.7 generates the Fibonacci sequence and appends each number to a non-volatile, doubly-linked list. For illustrative purposes, each iteration of the main loop toggles a GPIO pin to track progress. In the debug build, `main` begins with an energy-hungry consistency check that traverses the list and asserts that the previous and next pointers and the Fibonacci value in each node are consistent. This invariant helps detect problems early before they precipitate into mysterious failures akin to the one in Section 7.4.3. With intermittent power, the invariant was violated in several experimental trials.

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

Figure 7.7: Application code instrumented with a consistency check of arbitrary energy cost using EDB's energy guards.

Symptoms. The application's release build produces an inconsistent list without any indication that there is a problem. The debug build stops executing the main loop after having added approximately 555 items to the list. The top trace in Figure 7.8 shows an early charge cycle when the main loop executes and a later one when it no longer does.

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

EDB lets the developer keep the consistency check without breaking application functionality by wrapping the check with energy guards as shown in Figure 7.7. The effect this has on the target energy state is captured in the bottom oscilloscope trace in Figure 7.8. At instance 1, the target enters the energy guard, and EDB tethers it to a power supply. The capacitor starts charging, while the target continues executing the code within the energy guard. At instance 2, the target exits the energy guard, and EDB cuts the power supply and starts to discharge the capacitor to the level it had at instance 1. After the discharge completes, the target is allowed to continue. This sequence of events later takes place again between instances 3 and 4. With the energy guard around the consistency check, the main

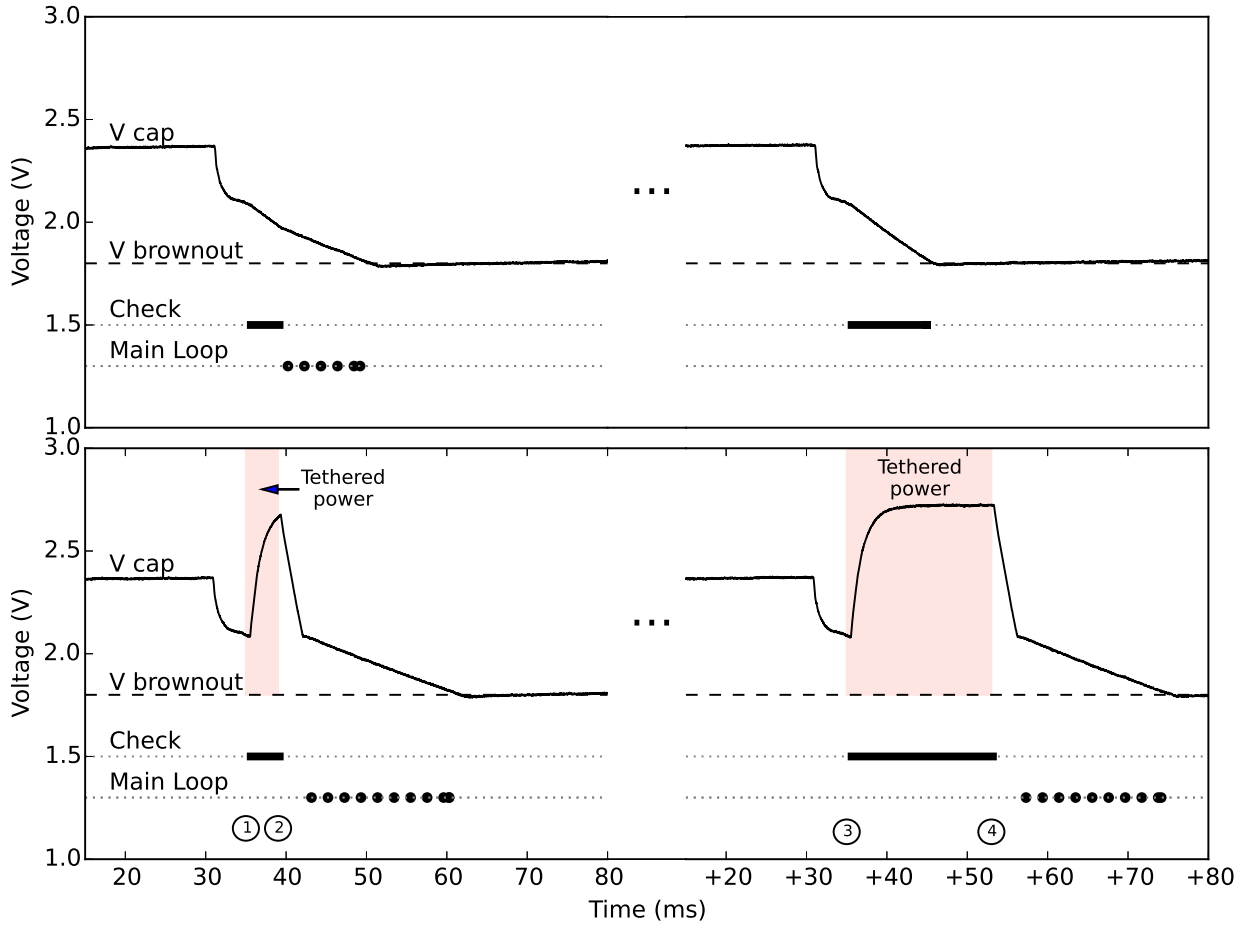


Figure 7.8: Oscilloscope trace of an application instrumented with a consistency check of high energy cost. Without an energy guard (top), the check and main loop both execute at first (left) but only the check executes in later discharge-cycles (right). With an energy guard (bottom), the check executes on tethered power from instant 1 to 2 and 3 to 4, and the main loop always executes.

loop gets the same amount of energy in both early charge-discharge cycles when the list is short (left) and later ones when it is longer (right).

Tracing events and profiling energy cost

Intermediate results of calculations, frequency of events, and energy cost of operations are valuable clues for quick diagnosis of erroneous code. Directly extracting such information from an energy-harvesting device using existing tools changes the application's behavior. For example, the sample rate of a sensing application may increase by a factor of 100-1000x when

powered continuously in the lab relative to when harvesting energy in a realistic deployment. This section demonstrates how EDB's energy-interference-free `printf` and watchpoints can peek under the hood of running code with minimal impact on application behavior.

Application. The activity recognition application outlined in Figure 7.9 reads an accelerometer sample, classifies the sample as "stationary" or "moving", and records statistics in non-volatile memory.

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

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

The measurements in Table 7.4 demonstrate the impact on application behavior of using UART. The energy cost of the print statement changes the iteration success rate, i.e. the fraction of iterations that successfully complete out of the total attempted. To trace application progress without disrupting its behavior, we instrumented the loop body with an EDB `printf` and three watchpoints as shown in Figure 7.9. The `printf` produces a stream of intermediate classification results for each iteration. The watchpoints produce a time and energy profile of a loop iteration as well as an independent calculation of the statistics that is useful for manual verification. The energy profile shown in Figure 7.10 was calculated from the difference between energy level snapshots taken by watchpoints 1 and 2, and watchpoints 1 and 3. Reference classification statistics can be calculated by counting occurrences of watchpoints 2 and 3.

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

Figure 7.9: Tracing and profiling an activity recognition application using EDB's energy-interference-free `printf` and watchpoints.

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

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

Table 7.4: Cost of debug output and its impact on the behavior of the activity recognition application.

Debugging and tuning RFID applications

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

Application. The WISP RFID firmware [187] decodes RFID query commands from a reader in software and replies with a unique identifier.

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

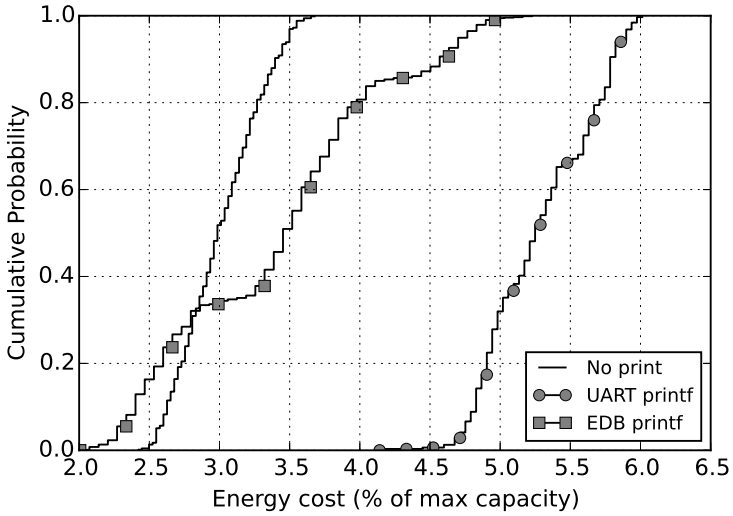


Figure 7.10: Energy profile of one loop iteration in the activity recognition application when instrumented with different output mechanisms.

responses per queries received. Correctness cannot be verified without evidence that the application software successfully decodes and acts on each valid incoming query message.

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

We use EDB to stream RFID message identifiers and target energy readings to the host. From data plotted in Figure 7.11 we find that in our lab setup the application responded 86% of the time for an average of 13 replies per second. The view focused on one discharge cycle confirms that the application successfully received consecutive incoming query messages and replied. To produce such a mixed trace of I/O and energy using existing equipment, the target would have to be burdened with logging duties that exceed the computational resources left after message decoding and response transmission.

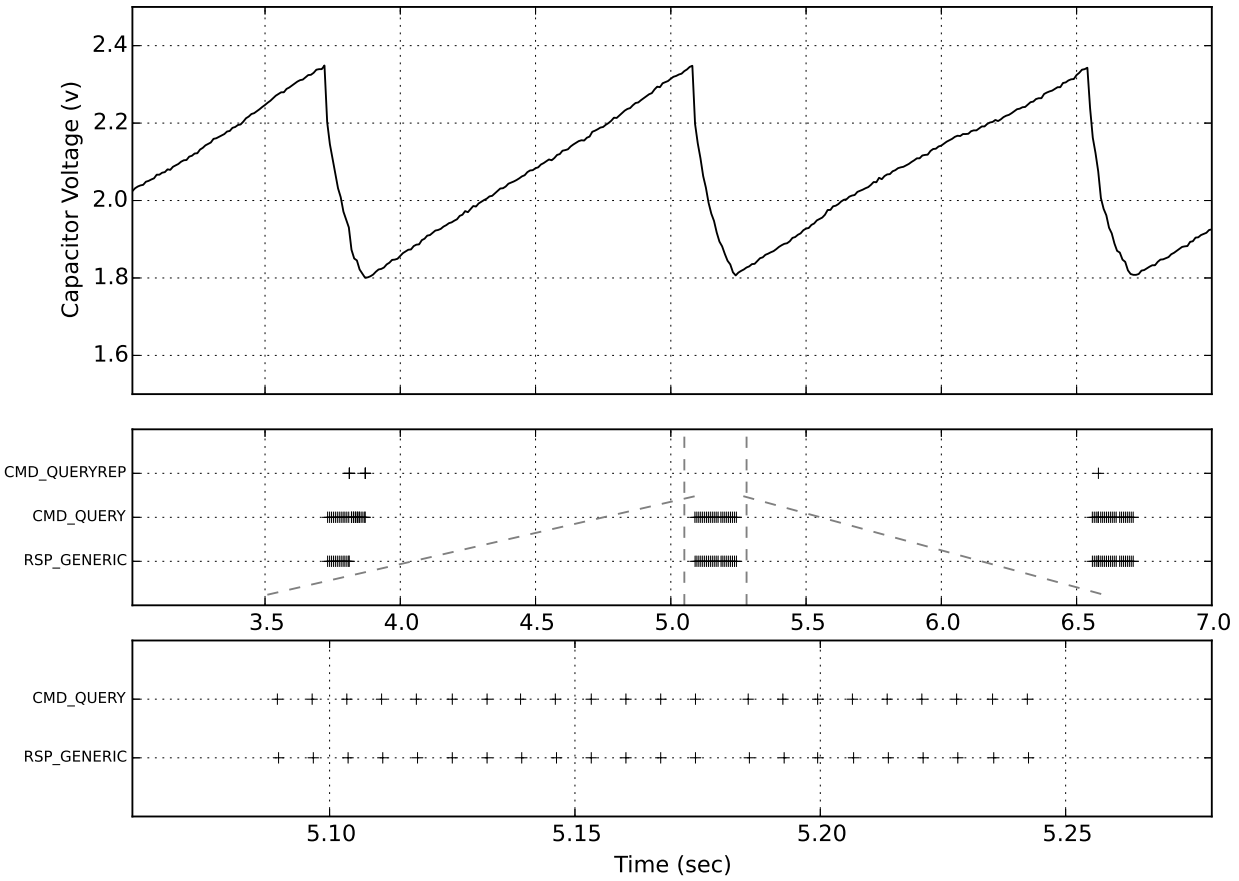


Figure 7.11: Incoming and outgoing RFID messages correlated with energy level recorded by EDB.

7.5 Summary

Intermittently executing, energy-harvesting devices present unique system reliability challenges, and our work presents the first debugging system that is designed to address those challenges. We identified *energy-interference-freedom* as a property that is essential to the utility of a debugging platform for power intermittent systems and built EDB to espouse that property from its circuits to its software.

EDB supports passive monitoring of a target device's energy, software events, and I/O. Using its ability to manipulate a target device's energy, EDB also supports active debugging tasks with energy-interference-freedom, including assertions, instrumentation, tracing, and interactive debugging. We evaluated our prototype of EDB, including custom hardware,

showing that it is energy-interference-free in both its passive and active tasks, and that it provides valuable debugging information that is out of reach using existing tools and techniques. We see energy-interference-freedom as a necessary property for future debugging and profiling tools for energy-harvesting devices.

Chapter 8

A Design Methodology for Intermittent Systems

In this chapter, we aggregate the system support developed in the preceding chapters into a re-usable design procedure for intermittently-powered systems. We propose a template for power system hardware to accelerate the design. We apply this design methodology to build an intermittently-powered energy-harvesting device for a space mission in low-earth orbit.

8.1 Design methodology

The design of an energy-harvesting system for a chosen application involves software tasks and hardware tasks that take place concurrently. We propose a top-down design approach, where the application requirements and constraints drive the design decisions. The top-down approach is appropriate, because energy-harvesting devices are generally *special-purpose*, i.e. intended for one specific application as opposed to general-purpose computing systems intended for multiple applications. As a consequence, the system is customized to the specific application.

In particular, the type and magnitude of a harvestable energy source is intrinsic to each application environment, and the energy source determines the achievable capabilities of the

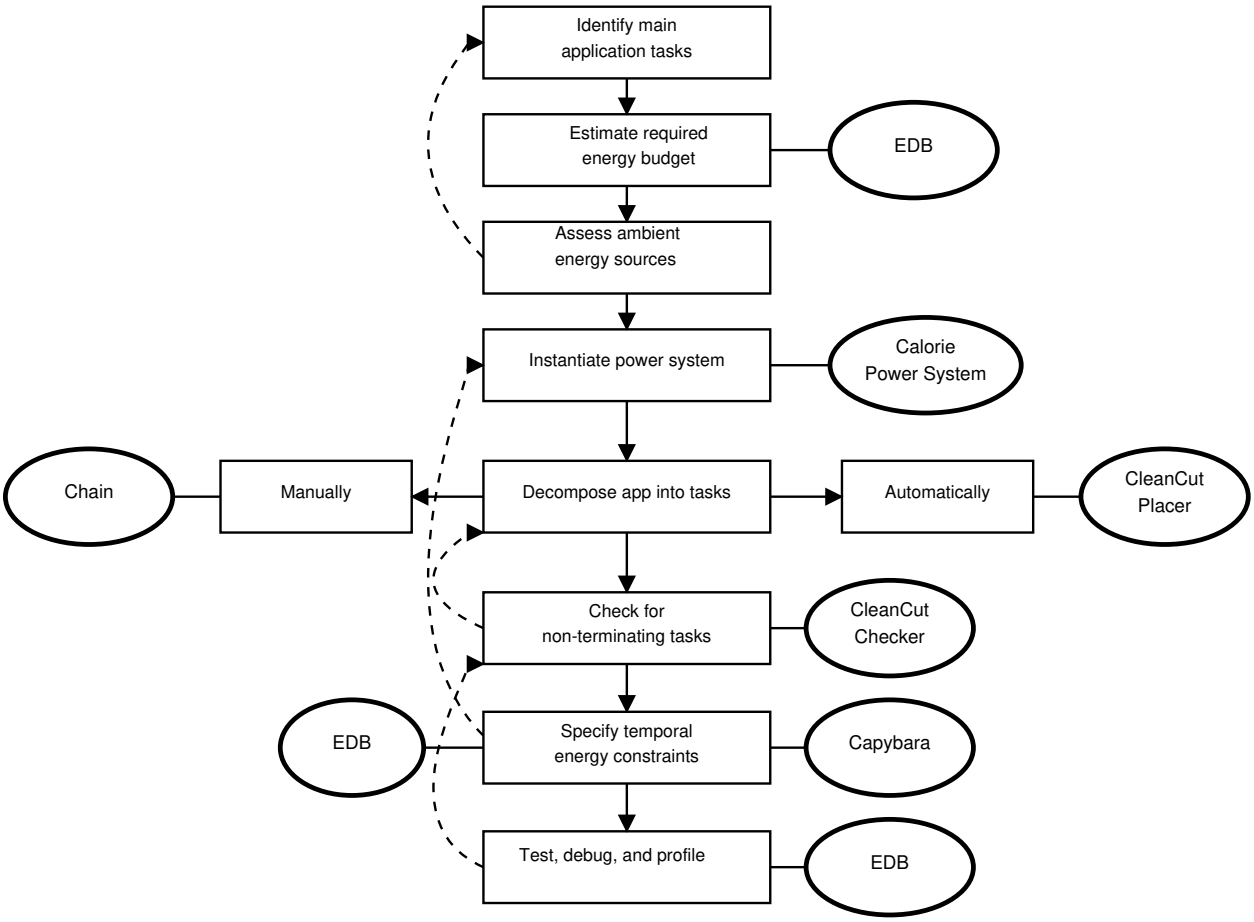


Figure 8.1: Design flow for building applications on energy-harvesting platforms, leveraging system support for intermittent computing proposed in this thesis (shown in ellipses).

device. For example, a soil moisture sensor in a field may have a power budget of tens to hundreds of milliwatts from a solar panel, which enable long-distance radio transmissions, while a wearable step counter may have only microwatts of power from a piezoelectric harvester, capable of only a weak radio link to a nearby smart phone. The constraints on the size, shape, and material composition of the device are also determined by the application. For example, a device destined to be carried by an insect must weigh no more than a gram [44], which disqualifies large supercapacitors. The tight coupling between the application and the platform does not permit an exact design methodology, however, we decompose the procedure into tasks that we anticipate to be common across applications.

We outline the top-down design flow in Figure 8.1 and explain how the system support

developed in this work helps with each of the software and hardware tasks. Since energy plays a central role in the design, first, the application tasks must be defined sufficiently to estimate their rough energy requirements. This estimate would then be converted into an energy budget requirement on the energy source. Once a budget is established, the energy sources available in the target deployment environment can be assessed against that budget to determine the feasibility of the core functionality in the application. A battery is one of the potential sources to be considered, alongside the harvestable energy sources, and for some subset of applications a battery may be adequate, for the remaining subset the only viable choice would be an energy harvester. For each promising energy source, the size of the harvester relative to its power output must be assessed in the context of the size constraints of the application.

At this point, the designer may choose to loop back to the application definition step to revise the application mission, potentially simplifying it or enlarging it. This stage of the design, may make use of documentation for part specifications and of energy measurement functionality of the EDB tool introduced in Chapter 7. The documented specifications list maximum current consumption of key hardware components – processor, radios, sensors. EDB can measure the energy consumption of prototyped tasks core to the application, such as sensor access, radio transmission, and bulk computations. In the next section we propose a template for the hardware power system and continue with the software design flow afterwards.

8.2 Calorie power system template

Once the harvester and its approximate power output is determined, a power system design can be selected. A power system accepts energy from the harvester, charges energy buffering capacitors, and generates a usable output voltage to power the load. There is a wide design space within a fixed power budget from the harvester, and each point in that design space

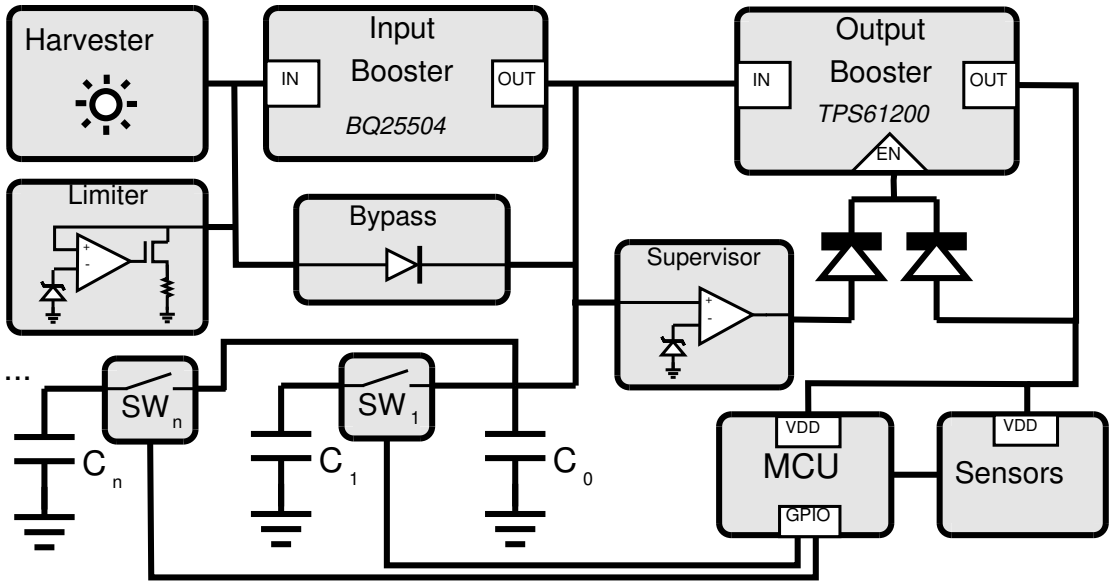


Figure 8.2: Calorie, a power system template for energy-harvesting devices.

differs significantly in effective usable energy, capacitor volume, and area occupied by the power system. Optimization across this design space consists in choosing and parameterizing the power system components listed in Chapter 2. For example, if minimizing capacitor volume is essential, then input and output power conditioning circuits may be included to provide the same energy storage capacity in a smaller volume. This section introduces a *template* for a power system from which concrete designs can be instantiated. This template can be customized depending on application constraints, and it must be instantiated by choosing the input and output power conditioning, the energy buffer type and size.

Calorie, our proposed power system template shown in Figure 8.2 (with detailed schematics in Appendix A) is versatile. It is compatible with harvesters of a wide range of input voltage, from 0.3 V to 5 V. Energy buffers may use high-ESR capacitors, such as small dense super-capacitors listed in Table 2.1. The hardware supports loads with voltage requirements that may exceed the harvester voltage output or the capacitor voltage rating, such as sensors and radios. These benefits stem from the input voltage limiter, and input and output boosters.

The *voltage limiter* circuit allows the harvester voltage input to rise above the ratings

of the components in the system, allowing a wide dynamic range of input power conditions. For example, the limiter allows solar panels to be connected in series to handle dim lighting conditions, while avoiding damagingly high voltages in bright light. The limiter decreases the voltage by burning energy that cannot be used. The I-V curve of the harvester (cf. Figure 2.2) implies that draining current will decrease the voltage. Our limiter drains current through a resistor. The value of this resistor R must be chosen such that for the maximum current, I, that the harvester can provide above 3 V (the limit of our input booster), the voltage drop across the resistor is within 3 V, i.e. $IR < 3$ V.

The *input booster* is located between the harvester and the energy buffering capacitors and allows the device to use weak input power from the harvester by boosting its voltage. Charging capacitors from a boosted voltage, instead of the voltage from the harvester, allows using harvesters that produce a voltage too low to operate the system. The TI BQ25504 input booster has a "cold-start" phase that substantially slows charging of large capacitors at low input power. To reduce charge time, when the harvester is producing sufficient energy to charge quickly, we added an *input booster bypass* optimization. The bypass circuit keeps the capacitors disconnected from the booster output and charges them directly from the harvester (through a keeper diode), until the booster starts and the capacitor voltage is above the cold start threshold.

We observed that the bypass optimization can reduce the charge time by up to 8x, as an example in Figure 8.3 shows. Figure 8.3 shows the charging of the capacitor bank without (top) and with the bypass optimization. Without the bypass, the input booster (BQ25504) must simultaneously charge both its output capacitor (BQOUT) and the main capacitor bank (VBANK) in the inefficient cold start mode. With the bypass, the main bank remains disconnected from the output of the booster and charges directly from the harvester (VHARVEST). Charge from the harvester begins to flow into the main capacitor (VBANK) when the booster stops loading it, which happens when the booster's output capacitor (BQOUT) is charged to the configured output value for the input booster (2.8 V).

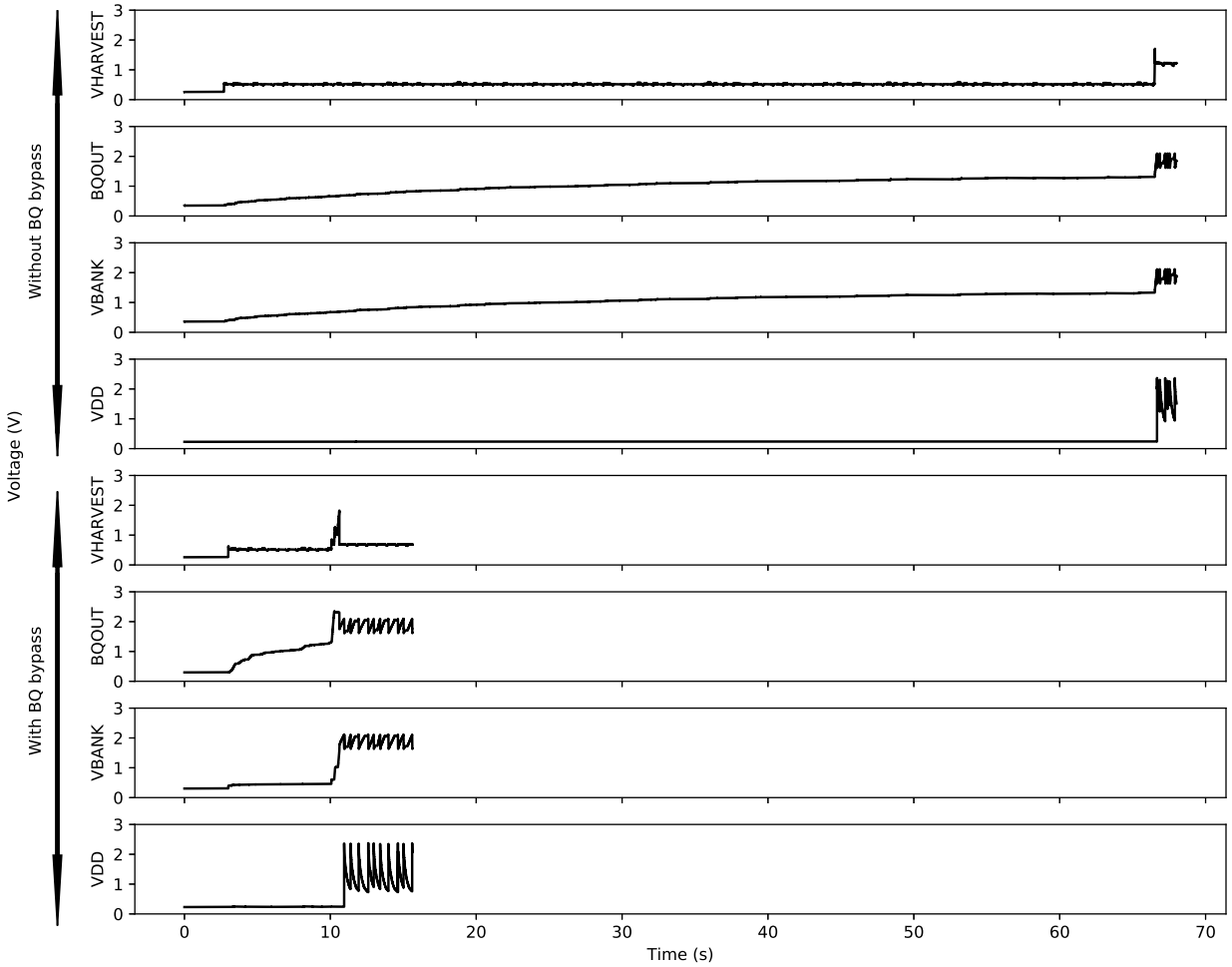


Figure 8.3: Cold start with and without bypass optimization. The top trace shows the time the baseline power system takes to startup, and the bottom trace shows the reduced startup time with the bypass optimization. Both traces show the voltage from the harvester (VHARVEST), on the output from the input booster (BQOUT), on the capacitor bank (VBANK), and supplied to the load (VDD).

Direct charging of the main bank continues until the main capacitor charges to 1.6 V, at which point the bank is connected to the booster output. Once connected, the main bank continues to charge to booster's output voltage (2.8 V) *in the efficient mode*, since its voltage is now above the cold start voltage threshold. The bypass reduces the amount of capacitance that must be charged in the inefficient cold start mode to only the output capacitor of the booster, and not the main bank.

The *output booster* allows the system to extract more stored energy from the energy

buffering capacitors than a direct connection to the load. The boost on the output also helps support sensors that require voltage higher than the maximum voltage supported by the capacitor. The booster produces stable output voltage, despite decreasing capacitor voltage until the capacitor is discharged nearly completely. For example, the TPS61200 booster with a minimum input voltage rating of 0.3 V extracts 98% of energy storable in a CPX3225A supercapacitor [153] rated for 2.6 V. Output boosting is required especially for high-density high-ESR supercapacitors to compensate for the voltage droop induced by the ESR under load, illustrated in Figure 2.4. For high-ESR capacitors, this immediate voltage droop is large enough to bring the voltage close to or below the minimum voltage required by the load. This behavior makes no or very little energy extractable from the capacitor, canceling out the gain from the its high energy density. Furthermore, the regulated voltage of the output booster is instrumental for powering sensors, actuators, and radios with a minimum operating voltage that is close to or above the capacitor rating (e.g. 2.5v gesture sensor or 2.0v for BLE radio). Without the booster, the load's minimum voltage requirement would be satisfied for no or very short time, while the capacitor is nearly fully charged.

The combination of an input and output booster, makes Calorie flexible to support capacitors and supercapacitors with a wide range of voltage ratings. Flexibility in capacitor choice is important for the system designer, because the same energy storage capacity is achievable with combinations of different capacitor types and size, each with different density, leakage, and endurance trade-offs (cf. Table 2.1).

Implementing the above design raised challenges with the choice of the output booster model and efficiency of the input booster. In an energy-harvesting device, the output booster is used out-of-spec, because it has to operate on a weak voltage source, e.g. a capacitor with a high ESR (cf. Section 2.2). We evaluated several booster models, a subset of which worked in our energy-harvesting context – an inductor-based (TPS61200) and a capacitor-based model (LTC1682). We chose the inductor-based booster for its higher efficiency and lower minimum input voltage, compromising on the footprint area.

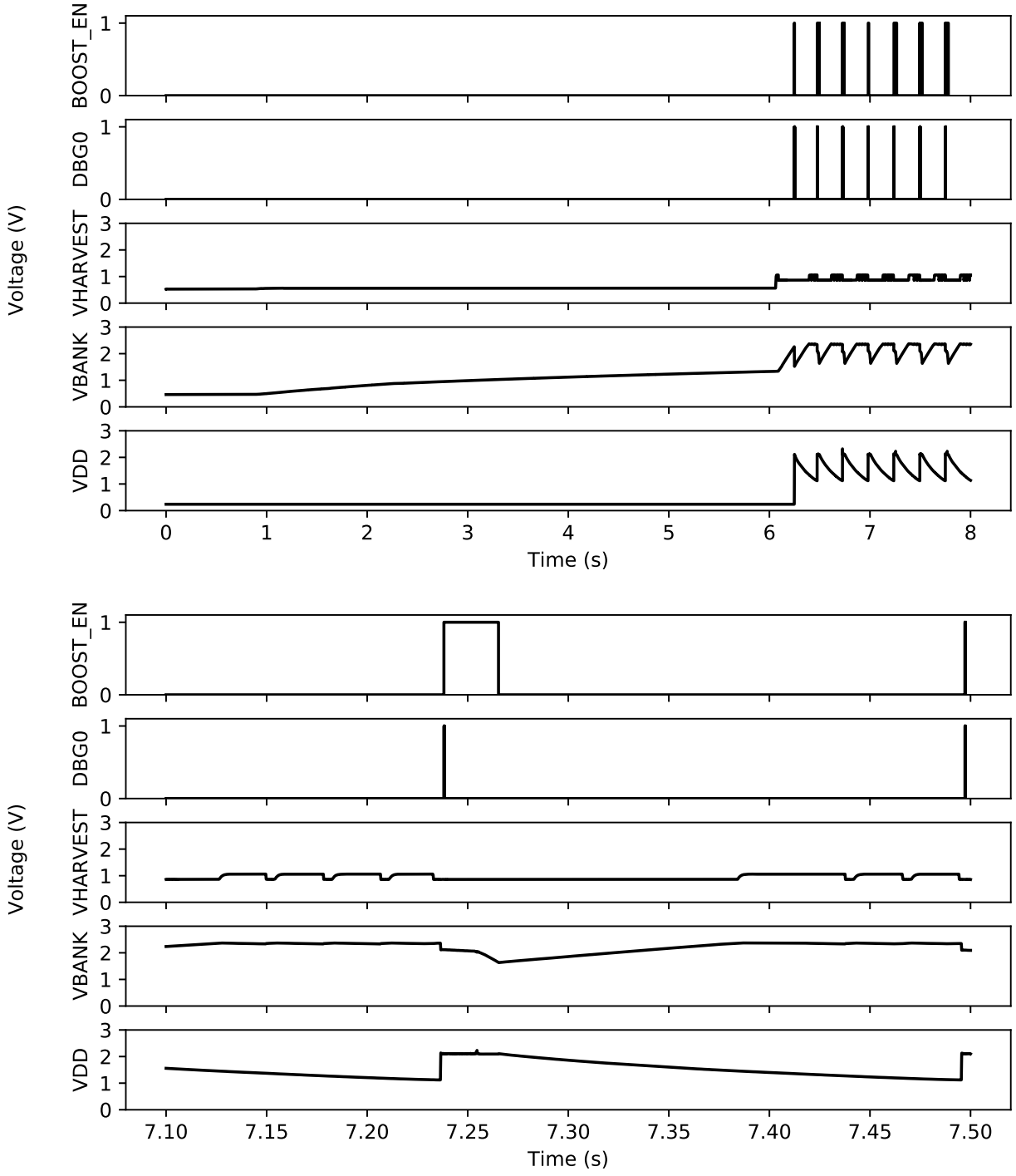


Figure 8.4: Trace of power system operation. The top figure shows the startup and operation, and bottom figure zooms in on one charge cycle. The traces show voltage from the harvester (VHARVEST), on the capacitor bank (VBANK), and at the load (VDD), as well as the signals that mark the open trigger (BOOST_EN), and the initialization code executing on the processor (DBG0).

A trace that demonstrates how the power system operates with a solar harvester is shown in Figure 8.4. The energy source used for this experiment is an incandescent bulb placed at 0.5 m away from the IXYS solar panels on the Capybara board prototype (cf. Section 6). The voltage trace shows that the charging of the capacitor happens in two phases: the cold-start phase of the BQ25504 voltage converter from 0 V to 1.5 V, which takes 5 s, followed by normal booster operation from 1.5 V to 2.6 V, which takes 150 ms. The cold-start phase only happens one time, and is avoided as long as there is some incoming energy, by implementing a close trigger (cf. Section 2.4), which disconnects the load as soon as the capacitor voltage drops close to the cold-start exit threshold (1.8 V). During charging in cold-start mode, the BQ25504 input booster loads the harvester, limiting its voltage output (VHARVEST) to 0.3 V, according to the harvester's I-V curve (cf. Figure 2.2). In normal charge mode, the harvester voltage fluctuates as the booster adjusts its load to track the maximum power point (cf. Figure 2.2) and input power varies.

Once the capacitor is charged to open trigger threshold (2.6 V), the power system opens the current path to the load, indicated by the BOOST_EN signal raised high and the corresponding ramp up of the voltage output (VDD) to 2.5 V. At this point, the processor begins executing code, and consumes the accumulated energy, as the trace of VBANK shows. After operating for 20 ms, the close trigger described earlier disconnects the load, and the discharging of the capacitor bank stops. The following charge intervals do not require cold-start, as explained above, and take only 230 ms each.

8.3 Hardware-software co-design for application tasks

To choose a capacitor configuration, energy storage capacity required by the software must be estimated. The energy storage capacity is a function of the energy consumption of individual tasks in the software. The software tasks fall into three categories, that should be handled in order: atomic tasks, reactive tasks, computation tasks.

The energy storage sizing process should start with the tasks with an on time constraint (cf. Chapter 6), since these tasks determine the maximum energy capacity the power system must be able to provide. Full-system energy consumption of such tasks can be measured using EDB. After atomic tasks, power system configurations for synchronous and asynchronous sensing tasks must be setup, by identifying their temporal constraints on off time defined in Chapter 6. Once their temporal constraints have been determined, tasks must be annotated according to the Capybara API (cf. Chapter 6). By annotating the application tasks with Capybara primitives, the number of separate capacitor banks will become determined and ready to be passed to the hardware design.

The remaining computational part of the software differs from tasks that interface with the physical world. Computational tasks lack temporal constraints and tolerate being interrupted and resumed at any point. Consequently, the computation can be partitioned into tasks using automatic decomposition algorithm in CleanCut introduced in Chapter 5. The output of the partitioning algorithm is a set of locations in the program where task boundaries should be placed. These locations can be manually mapped into tasks in the Chain abstraction introduced in Chapter 4, or into checkpoints for a system like DINO [101]. Compared to writing the application without any support in the programming language, Chain brings the advantage of guaranteed memory consistency without explicit management of non-volatile state. Delegating the consistency problem to Chain frees up developer time for diagnosing and fixing bugs in the application software. After the task decomposition and the energy storage have been fixed, all tasks are checked for non-termination using the CleanCut checker.

Subsequent steps in the development process involve characterization of the device in order to assess application performance. Since this software is running on an intermittently-powered device, the energy-aware primitives of the energy-interference-free debugger (cf. Chapter 7) are necessary to debug the code interactively while the device is powered by harvested energy. For representative testing and evaluation, it is essential to run the device

using energy from the harvester and profile it *in situ*. The passive monitoring features of EDB facilitate *in situ* profiling. If the device size constraint permits it, the designer may choose to integrate EDB onto the device, or at least on its engineering variants. The evaluation stage is likely to uncover problems and direct the designer along a backedge in the design flow to the software modifications, which will change the energy consumption of the tasks. The change in task energy will necessitate a re-computation of task decomposition and re-verification of non-termination.

8.4 Design of a solar-powered nano-satellite

We apply the proposed design methodology to build an energy-harvesting (solar-powered) board-scale nano-satellite, named EDBsat and shown in Figure 8.5. We begin by defining the application mission and then present the design of the hardware and software system. The overall mission objective is to assess the viability of an energy-harvesting board-scale device as a low-cost platform for scientific missions in low-earth orbit. The specific mission of our device is to collect samples from a temperature sensor, magnetometer, and accelerometer,[1] average them over several time scales, and transmit the time series over radio to a receiver on Earth. A secondary objective is to record a profile of the energy harvested by solar panels as a histogram of the capacitor voltage at different points in the program.

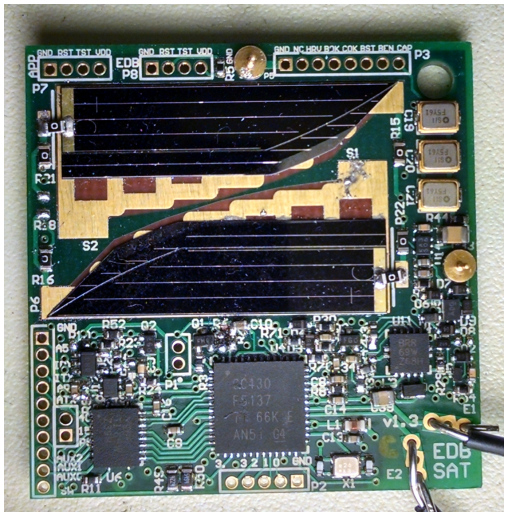


Figure 8.5: EDBsat: a solar-powered nano-satellite (front).

[1] Acceleration in orbit is expected to be zero, and accelerometer samples are included as a control data point.

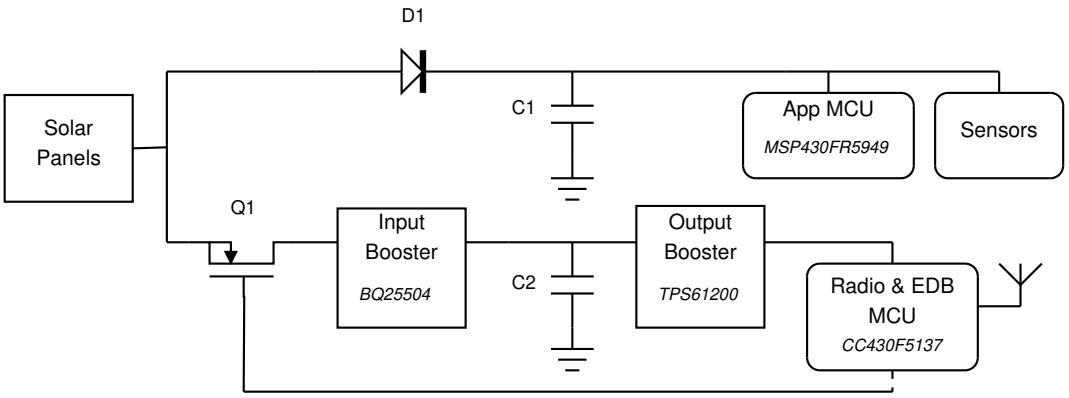


Figure 8.6: EDBsat power system hardware design

Our design is a variant of a *sprite* in the KickSat project [194]. The KickSat sprites are stand-alone satellites that fit on one PCB board. A set of 100-200 sprites are launched into orbit by loading them into a carrier cube-sat that then releases them using a spring-loaded mechanism. Our variant complies with the form factor imposed by the carrier: 35x35x4 mm. This size constraint is the main limiting factor that affects key hardware and software design choices.

8.4.1 EDBsat hardware

Following the first step of the design flow, we estimate the energy requirement of the key part of the application: the transmission of the radio packet to Earth. We employ the original KickSat radio stack. We use the current consumption specified in the radio datasheet (33 mA at 2 V) and measure the time to transmit the smallest packet of one byte (250 ms) to estimate the energy requirement of $33\ \text{mA} \cdot 2\ \text{V} \cdot 250\ \text{ms} = 17\ \text{mJ}$. The harvester must output this amount of energy in an interval of time that is short enough relative to the transmission window during which the satellite travels above the receiver on the ground, which is about 6 minutes. A solar-panel of size 8x26 mm produces 34 mW at its maximum power, under ideal incident angle. Assuming ideal light angle and ideal energy storage, this harvester outputs sufficient energy for 4 packets each second, confirming the feasibility of the application. In practice, to account for non-ideal conditions, we double the harvester area and, in contrast

to the KickSat reference sprite, we install panels on both sides of the satellite to continue harvesting energy in as many orientations towards the sun as possible. After allocating space for solar panels, only 50% of board area remains for the power system, the processor, sensors, and radio.

To satisfy the on-time constraint of the radio transmission, we specialized the Calorie power system to the one shown in Figure 8.6 and described in detail schematics in Appendix B. Our main design objective was to minimize the board space occupied by the energy storage buffer and the power system circuits. The system distributes power from one harvester (an array of four solar panels) to two isolated load domains: the application MCU with sensors and the radio MCU. Each domain has a dedicated capacitor bank, C1 and C2, and the load in each domain can access energy stored only in its own capacitor bank. This partitioned architecture is less flexible than the Capybara power system which allows sharing of the energy buffer across all loads, but it occupies less area, since it eliminates the switch between banks. In an alternative design with only one MCU for both workloads, a Capybara switch for one of the banks would be beneficial. The application domain uses a simple rectifier-based power supply circuit explained in Section 2.1. The Q1 p-channel FET switch enables interference-free profiling of the application by granting it exclusive access to the harvester. When the switch is opened by the EDB software, the EDB domain stops loading the harvester and operates using energy from its storage bank (C2).

The thickness constraint of 4 mm severely restricts the choice of supercapacitors, and requires that their volume be minimized. The radio and EDB domain uses a power system with both input and output power conditioning voltage converters, in order to minimize capacitor volume and guarantee an atomic burst of energy sufficient for a radio transmission (under the assumption of *any* positive incoming energy), as explained in Section 2.3.

8.4.2 EDBsat software

The satellite application was written in the Chain programming model, introduced in Chapter 4, from the ground up. The application consists of a task graph with 8 tasks and 12 channels. The main data structure that holds the hierarchical window averages is passed between tasks via channels, i.e. the data structure is allocated as a set of copies, where each copy is associated to a pair of tasks. The implementation relies on Chain's self-channels to encapsulate data within a task and make it accessible only to instances of that same task. In return for the effort spent on expressing the program as a graph of tasks instead of as a monolithic program, we gained the guarantee of consistent program state in volatile and non-volatile memory. During the development of the satellite we observed a non-terminating task which prompted a change in the task decomposition.

To fulfill the secondary mission of profiling energy available in the low earth orbit to a satellite of board-scale form factor, we built into the satellite a custom implementation of the Energy-Interference Debugger (EDB) [38]. We provisioned the satellite with two MCUs – one for the application, and one for the radio stack and EDB. In contrast to its reference design as a stand-alone debugging tool powered from the USB port of a computer, the EDB module on the satellite is powered from the main energy harvester. Despite a shared harvester, the EDB design prevents energy-interference from the debugger on the application, by activating switches that disconnect it from the harvester, as described in the previous section. Once disconnected, EDB uses its dedicated energy capacitor during profiling. EDB watchpoints were inserted at the start of key tasks in the application for tracing the program and recording both its progress and the energy at the start of each task. A compressed histogram of the energy at each watchpoint was packed in a dedicated type of packet and sent over the radio alongside the packets with the sensing data payload.

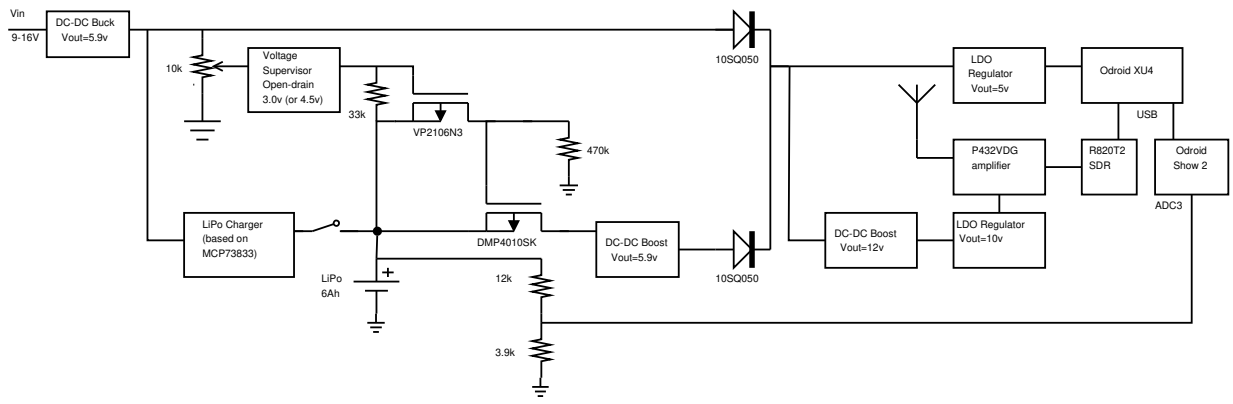


Figure 8.7: Hardware for a portable stand-alone receiver for transmissions from EDBsat satellite.

8.4.3 Ground station

We developed a customized receiver based on the reference design from KickSat project [194]. The receiver collects the radio signal using an antenna connected to an amplifier and digitizes it using a software-defined radio USB device. A software implementation processes the signal to extract valid packet transmissions. Our improvements to the original design allow it to run on an ARM-based quad core embedded single-board computer instead of a high-performance laptop. This was accomplished by parallelizing the decoder implementation using OpenMP [43] and enabling support for NEON extensions in the FFT library. We also make the receiver portable by including a battery with power conditioning and charging circuits, and a display for showing the decoded packets. An overview of the receiver hardware is shown in Figure 8.7.

8.4.4 Experimental results

We have designed and manufactured the EDBsat circuit board and deployed onto it the firmware with the application code and the EDB profiling code. To verify the operation of the power system we inspected our device with a logic analyzer while powering it with light from an incandescent light bulb. Figure 8.8 shows a trace of the voltage on the capacitor,

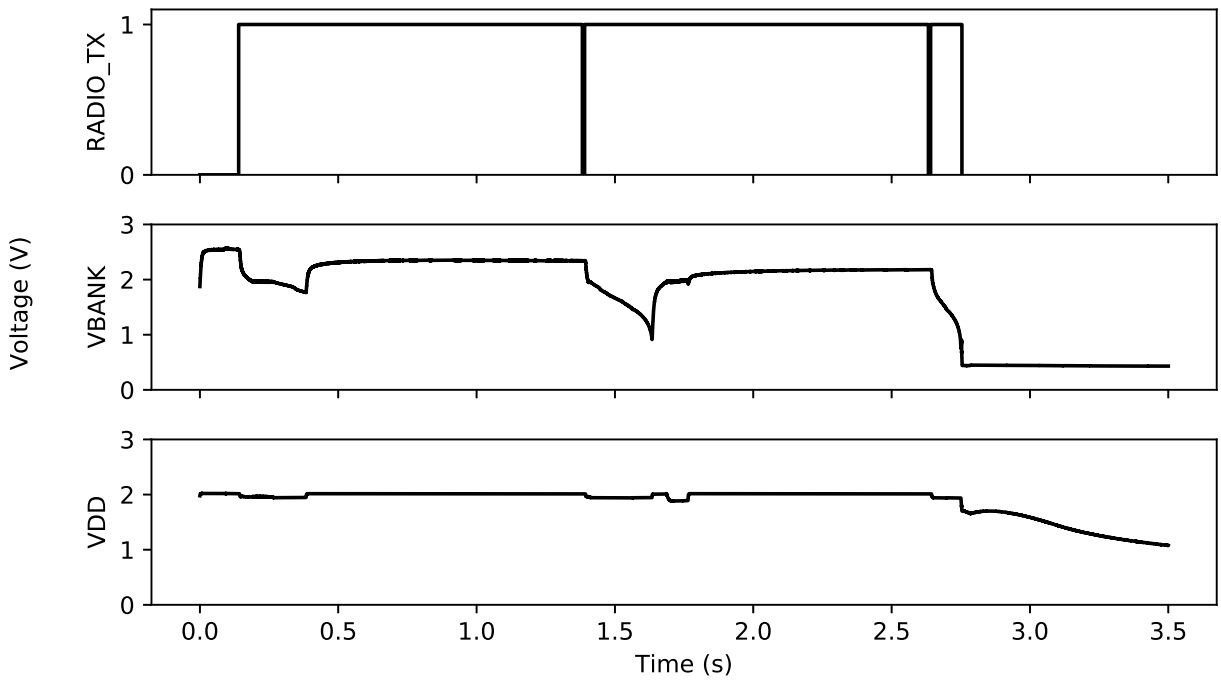


Figure 8.8: Trace of voltage on the energy capacitor (VBANK) and in the radio MCU domain (VDD) during transmission of a 2-byte radio packet (each byte indicated by high level of RADIO_TX).

and in the load domain of the radio, as the device transmits a 2-byte radio packet. The trace begins at the instant of the open trigger (cf. Section 2.4), when the supply to the load (VDD) is activated. As the device sends each byte, the radio is active for 250 ms followed by a MAC-protocol delay of 1 s during which the radio is in sleep mode. These two phases of each byte transmission are visible as the dips in the voltage on the capacitor. The rebound in the capacitor voltage is not due to incoming energy but due to the reduction in the voltage droop due to the ESR of the capacitor (cf. Section 2.2) when the load current diminishes. A transmission of a third byte is attempted, but is not successful because energy in the capacitor is exhausted before the transmission finishes.

The power draw of the radio for each of the two bytes brings the capacitor voltage below the threshold of operation of the radio, but the output voltage remains stable thanks to the output boost converter. The input booster guarantees that the capacitors will be charged to their maximum voltage even if the incoming voltage from the harvester is lower, such as

Time	Temp	Mag	Accel
Short	30	(7, 0, -5)	(1, -1, -3)
Long	30	(7, 0, -6)	(1, -1, -3)

Task	#HighE	#LowE
Boot	1	0
Sample	1	29
Avg-1	1	29
Avg-2	0	31

Table 8.1: Radio packets with sensor data and energy profile histograms from the satellite.

when the panel is oriented at a sharp angle to the sunlight.

To verify the sensing and energy profiling functionality we received and decoded the radio packets from the device over one hour while it was powered by light from an incandescent bulb. Table 8.1 shows a received packet with averages of sensor values down-sampled to 4 bits each over two time scales, and the energy profile for 4 instrumented points in the program. The profile shows that the program booted once and had sufficient incoming energy to execute all its tasks without rebooting, but starting each task with a partially depleted capacitor most of the time. We conducted a similar test outside by placing the EDBsat in the sunlight, configuring its transmission power to maximum (+12 dBm), and traveling with the receiver to 600 meters away from the satellite. The receiver decoded valid packets as long as the obstructions on the line of sight to the satellite were sparse (trees).

Chapter 9

Conclusion and Future Research Directions

As processors are miniaturized, it becomes possible to embed higher computational capacity into things, habitats, equipment, and the body. Embedded computation is a prerequisite for making these objects and environments "smart" and driving a new generation of applications in the industrial, civil, medical, and scientific domains. However, these emerging applications place tighter constraints on the device size and external operating conditions, which in turn constrains their energy source. Batteries cannot always meet the constraints due to being large, heavy, costly to replace, and limited in their tolerance to external temperature. Energy-harvesting circuits emerged as an alternative energy source capable of replacing the battery in some applications. An energy-harvester converts ambient energy in light, radio waves, temperature gradients, or motion into electric current.

In this work, we investigated energy-harvesting devices as a platform for embedded applications. Replacing a battery with an energy-harvester eliminates some of its limitations but creates a challenge for the software application by making the power supply *intermittent*. Intermittent power to the processor imposes an intermittent execution model on the software, which we defined in Chapter 2. Our work presented in the preceding chapters is motivated

by the thesis that software destined to be deployed within the intermittent execution model, requires system support across the stack, from programming language to hardware mechanisms, in order to execute reliably and efficiently. In our work we developed system support that addresses the following key challenges imposed by the intermittent execution model.

Programs written in the C language without regard to power failures may not execute correctly and efficiently in this intermittent model for three key reasons. First, the program will not execute to completion, because it will be interrupted by power failures. Second, the program may produce incorrect results due to power failures leaving its state in memory inconsistent. Third, the program may be slow to execute due to the overhead spent on saving and restoring intermediate state to and from non-volatile memory. To address these problems, we developed a specialized programming model and runtime for writing intermittent programs without explicit management of non-volatile state. In our programming model introduced in Chapter 4 programs are expressed as a graph of tasks that exchange values via persistent channels. By restricting inputs to and outputs from a task to disjoint channels, Chain ensures that tasks are idempotent and can be restarted if interrupted at any point. Chain latches intermediate results implicitly through saving task inputs and outputs into channels, eliminating the overhead of checkpoints.

To verify that the execution can progress along each possible path, we developed a complementary program analysis that formed the basis of two tools, introduced in Chapter 5. The CleanCut checker is a tool for checking a task-based intermittent program for non-terminating paths. The checker estimates the energy of each path in each task and compares it to the energy storage capacity of the device. Tasks that contain paths that would consume more energy than the device can store are reported as non-terminating. We employ the same program analysis to build a second tool, the CleanCut placer, that automates the decomposition of a program into tasks. The tasks are generated by repeatedly splitting the maximum-energy path in the program until the energy of all paths is below the energy storage capacity of the device.

We recognize that like traditional embedded systems, energy-harvesting systems must perform not only computation tasks but also sensing and actuation tasks. As opposed to interruptible computation tasks, tasks that interact with the physical world have temporal constraints. An actuation task, such as a radio transmission, must be executed to completion without interruption to be successful. A sensing task might benefit from being executed as frequently as possible to adequately sample a changing quantity and to react to external events on time. In Chapter 6 we developed the Capybara system that can satisfy the temporal constraints on energy that such tasks place on the power system. Capybara reconfigures the energy storage capacity of the device at runtime in order to control when the device accumulates energy and when it supplies the stored energy to tasks.

Our experience with writing software for energy-harvesting platforms has exposed the inadequacy of the debugging tools designed for continuously-powered devices. Because interactive debuggers interfere with the power supply of the device, they mask the bugs induced by intermittence and alter the behavior of applications that react to the energy state of the device. In Chapter 7 we developed an energy-interference-free debugging and profiling tool, EDB, that is specialized to work while the device under test is powered intermittently. EDB brings energy-aware variants of familiar debugging primitives to energy-harvesting devices, such as breakpoints, watchpoints, and assert statements, and introduces specialized primitives, such as energy guarded regions with energy compensation.

We unify the proposed solutions into a methodology for co-designing hardware and software for an energy-harvesting system, described in Chapter 8. The design flow begins with a feasibility assessment of energy-harvesting based on available energy sources, provides a template for power system designs, and provides a guide for analyzing and transforming the program into a task-based form for intermittent execution. We follow the proposed design methodology to build an end-to-end board-scale nano-satellite ready to be deployed into low-earth-orbit. This case study demonstrates the impact potential of the work developed in this thesis: facilitating development of novel applications enabled by energy-harvesting

technology. The hardware design files and source code that comprise the system support developed in this thesis are open and available at `http://intermittent.systems`.

9.1 Future work

To enable more and better applications with less effort from the developer, the system support must continue to expand through further research. Follow-up work in the near future can build directly on top of the solutions developed in this thesis. In particular, the Chain programming model presented in Chapter 4 can be improved in efficiency and usability with a compiler front-end that promotes the task and channel primitives to the level of language syntax. The CleanCut program analysis from Chapter 5 depends on an accurate energy model. Consequently, CleanCut as well as future energy-aware program analyses, will benefit from research into energy estimation techniques, particularly whole-system techniques that can track the hardware state of peripherals. Energy modeling is also a prerequisite for an extension to Capybara (cf. Chapter 6) for automated capacitor bank provisioning based on task annotations. Another improvement to the Capybara reconfigurable storage mechanism is a capability to maintain charge on the capacitors that hold energy for asynchronous tasks, to compensate for energy lost to self-discharge. New energy measurement tools, particularly power tracing functionality, may also be integrated into EDB (cf. Chapter 7) to support program analysis and to enable new debugging primitives such as energy consumption breakdown across software components.

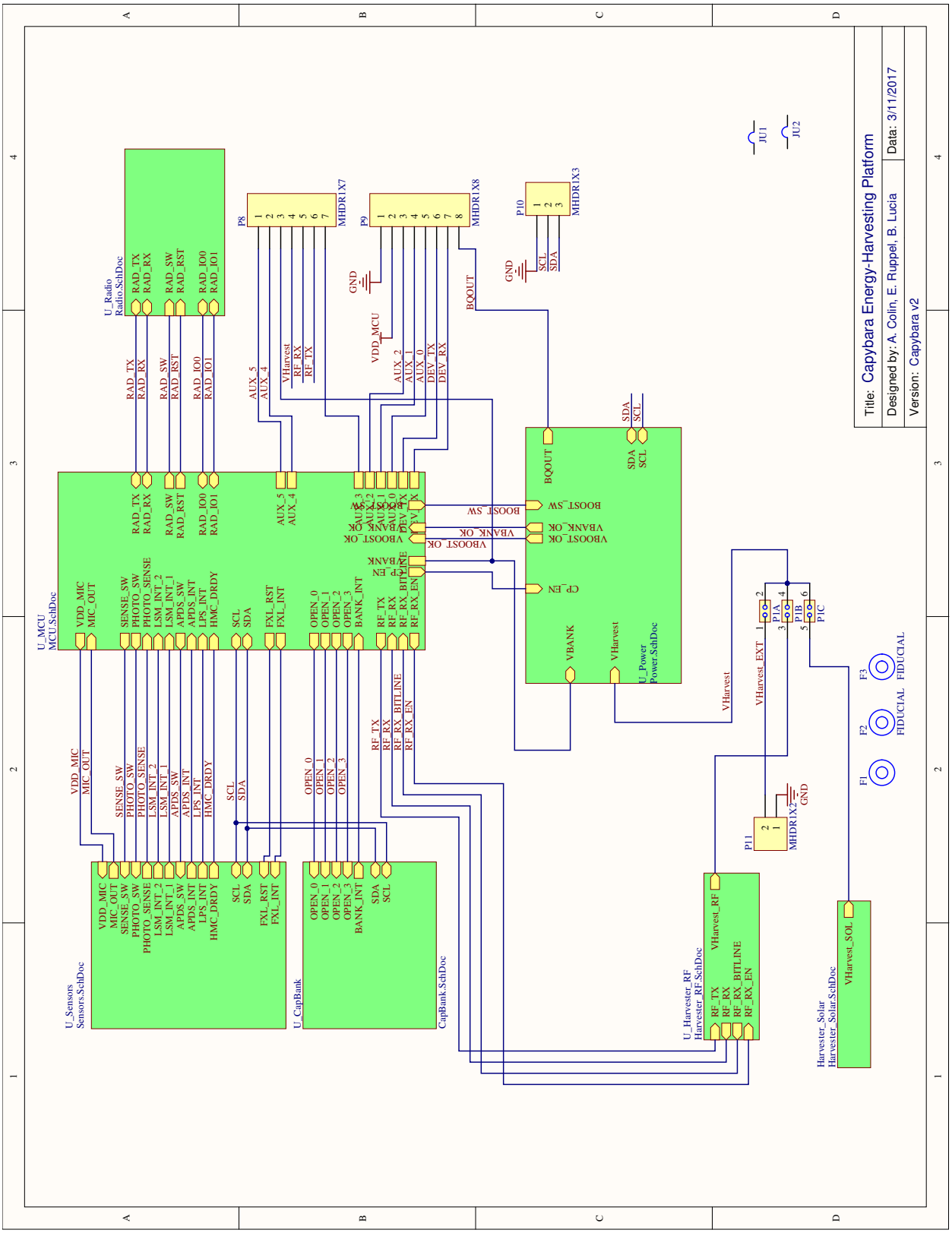
Long-term research effort is needed to *verify correctness* of programs in the intermittent execution model prior to deployment. Detecting bugs early in the design cycle is particularly important for energy-harvesting devices, because these devices are likely to be inaccessible after deployment (e.g. implanted or molded into construction materials). Despite availability of systems like Chain (Chapter 4) that guard against memory inconsistency, some developers may choose to manually engineer the code to tolerate power failures in an application-

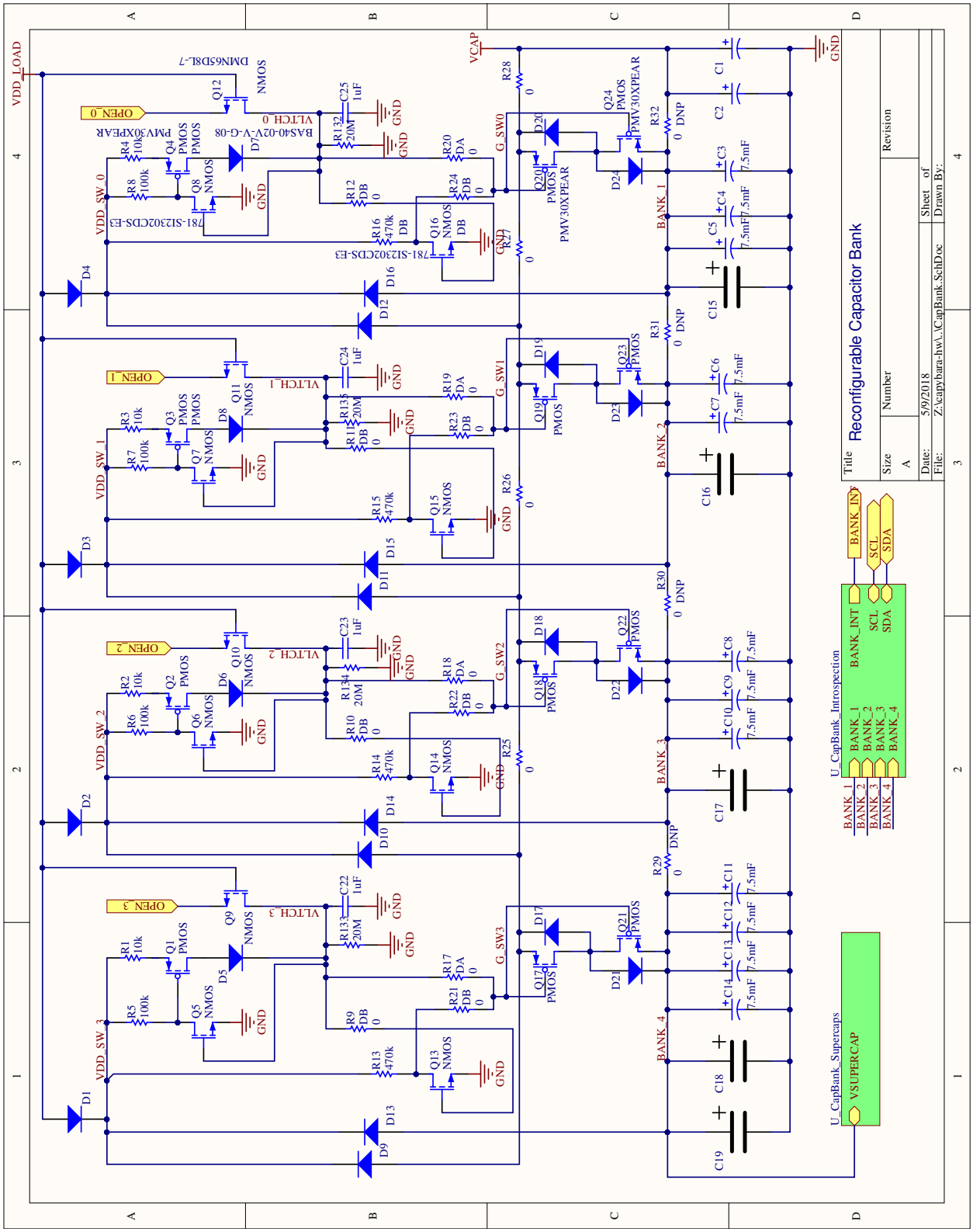
specific manner, in order to extract maximum performance. A verification tool is essential to make this error-prone approach practical. Bug finding techniques applicable to intermittent systems may originate in static analysis, model checking, or theorem-proving. Techniques for modeling failures used for verification of file systems may be applicable to intermittent systems. However, specialization will be necessary to remove the assumption of a file-system-like interface between the non-volatile and volatile part of the system. Furthermore, radical innovation will be necessary to make the verification tool capable of reasoning about energy. A successful verification tool should be able to prove that a program has the same behavior in the intermittent execution model as in a continuously-powered execution.

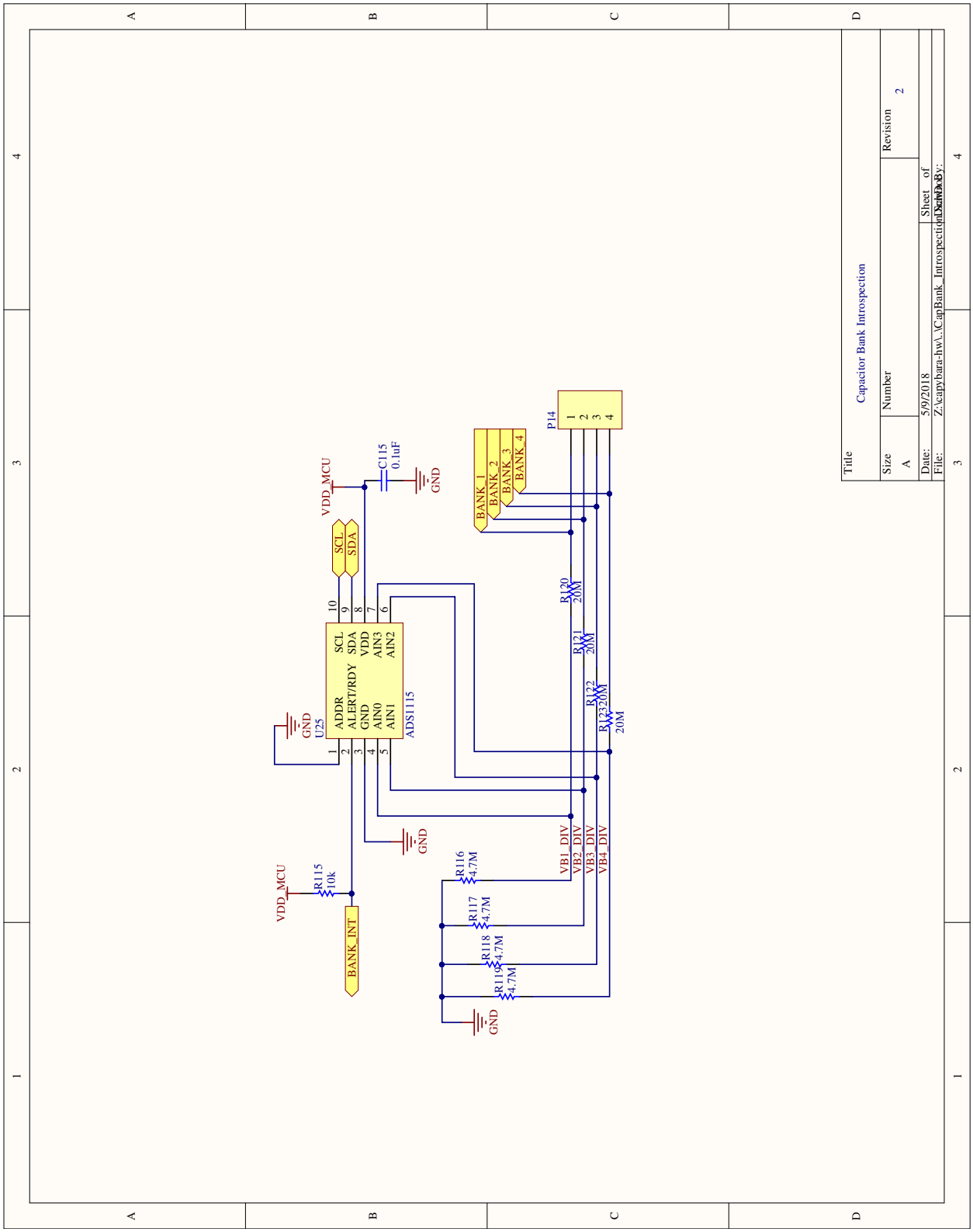
While this thesis has focused on support for intermittent computing on a single device, support for *intermittent communication* is an open challenge. Communication between intermittently-powered nodes is difficult, because devices cannot be assumed to be on at the same time, nor have energy to receive a packet at any moment. A solution to the communication challenge requires innovation in radio hardware, wireless protocols, time keeping and synchronization. Research on these topics in the domain of battery-powered wireless sensor networks can serve as a basis for solutions optimized for intermittent devices. Once a communication link between nodes is possible, research can proceed to macro-programming networks of intermittent nodes. Abstractions for programming a network of intermittent computers become essential for applications such as structural monitoring, which require correlated sensor data from multiple sensors. System support for reliable programming of a single intermittent computer, initiated in this thesis, augmented with support for programming networks of intermittent computers is a prerequisite for novel battery-free applications.

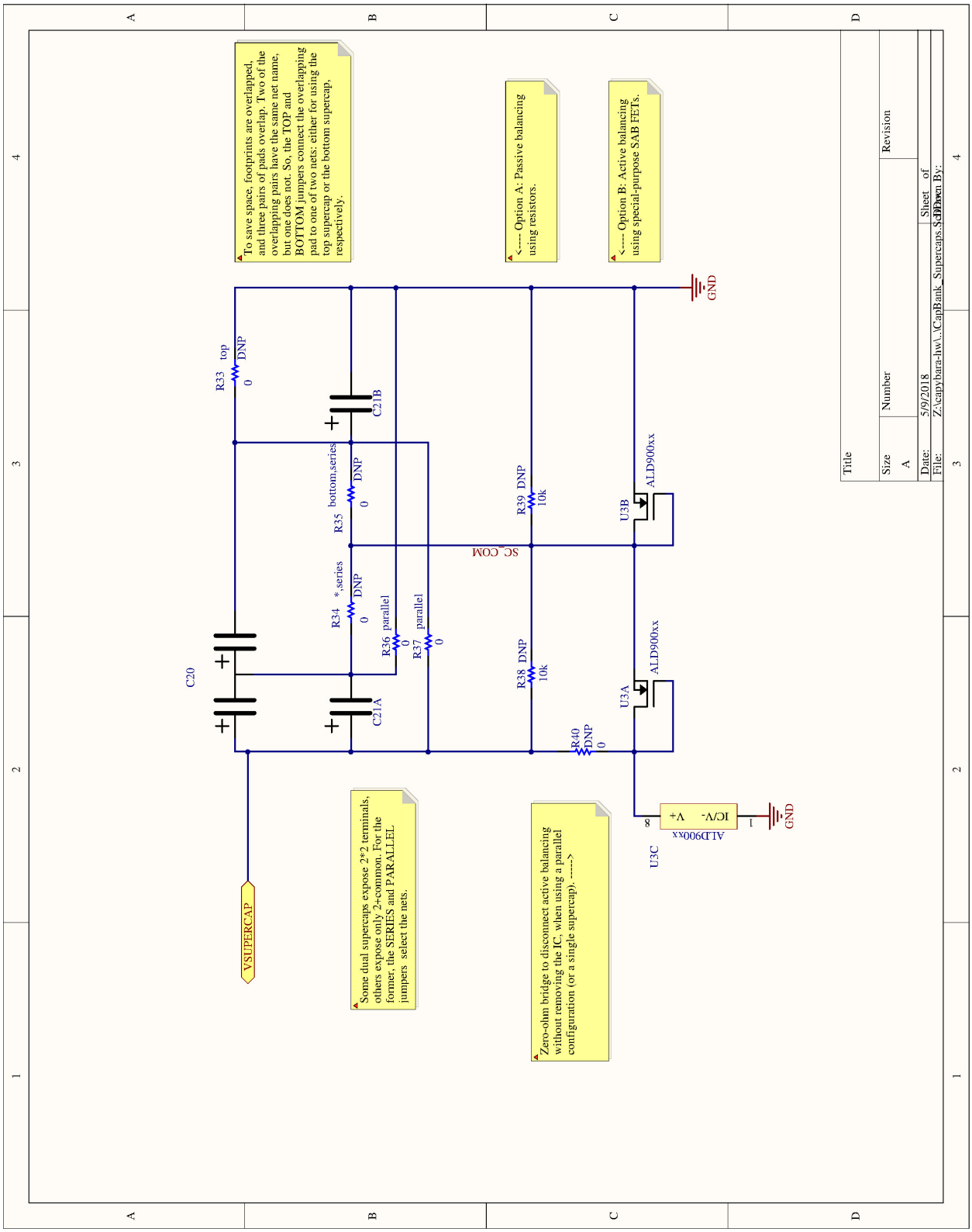
Appendix A

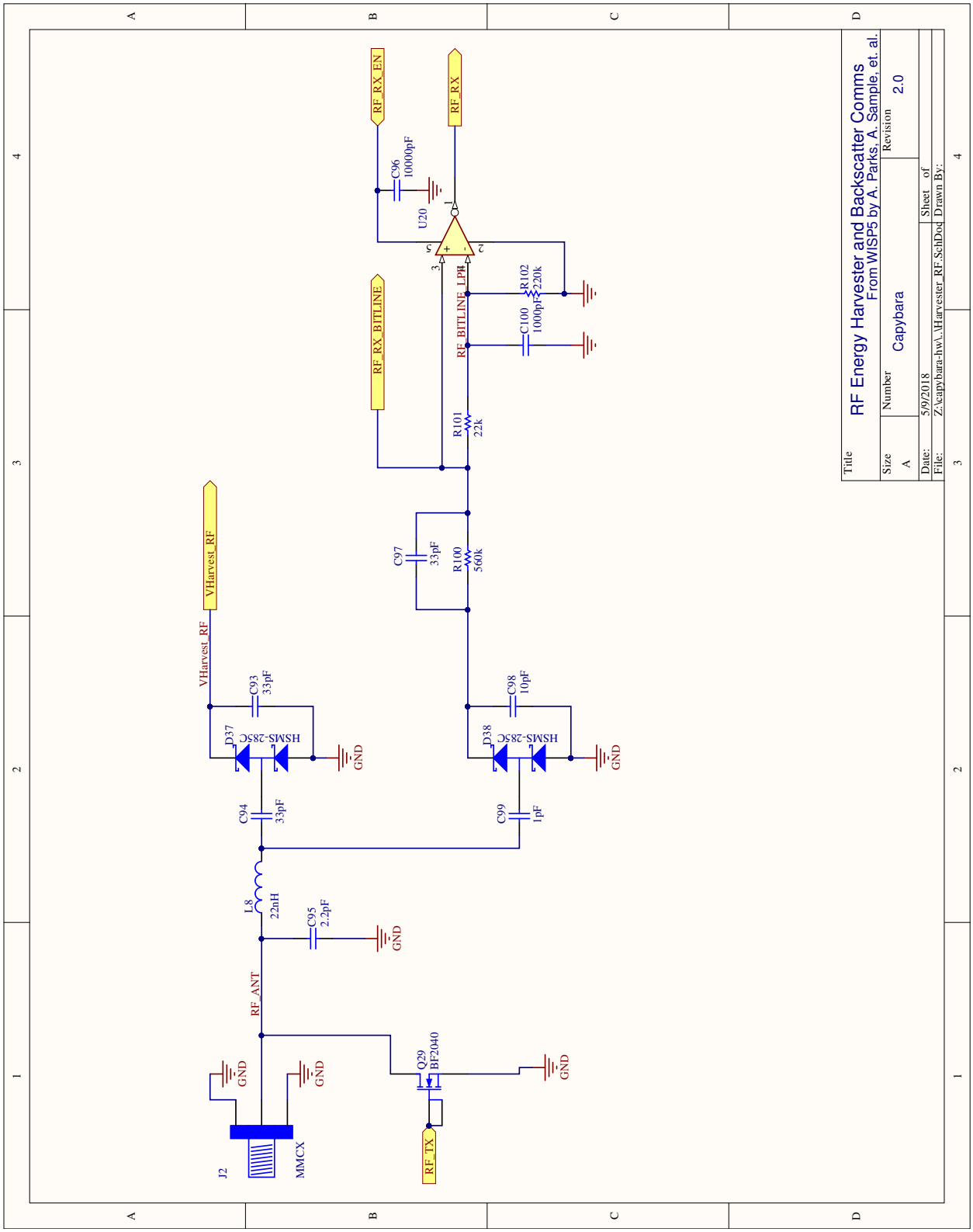
Capybara Hardware Schematics

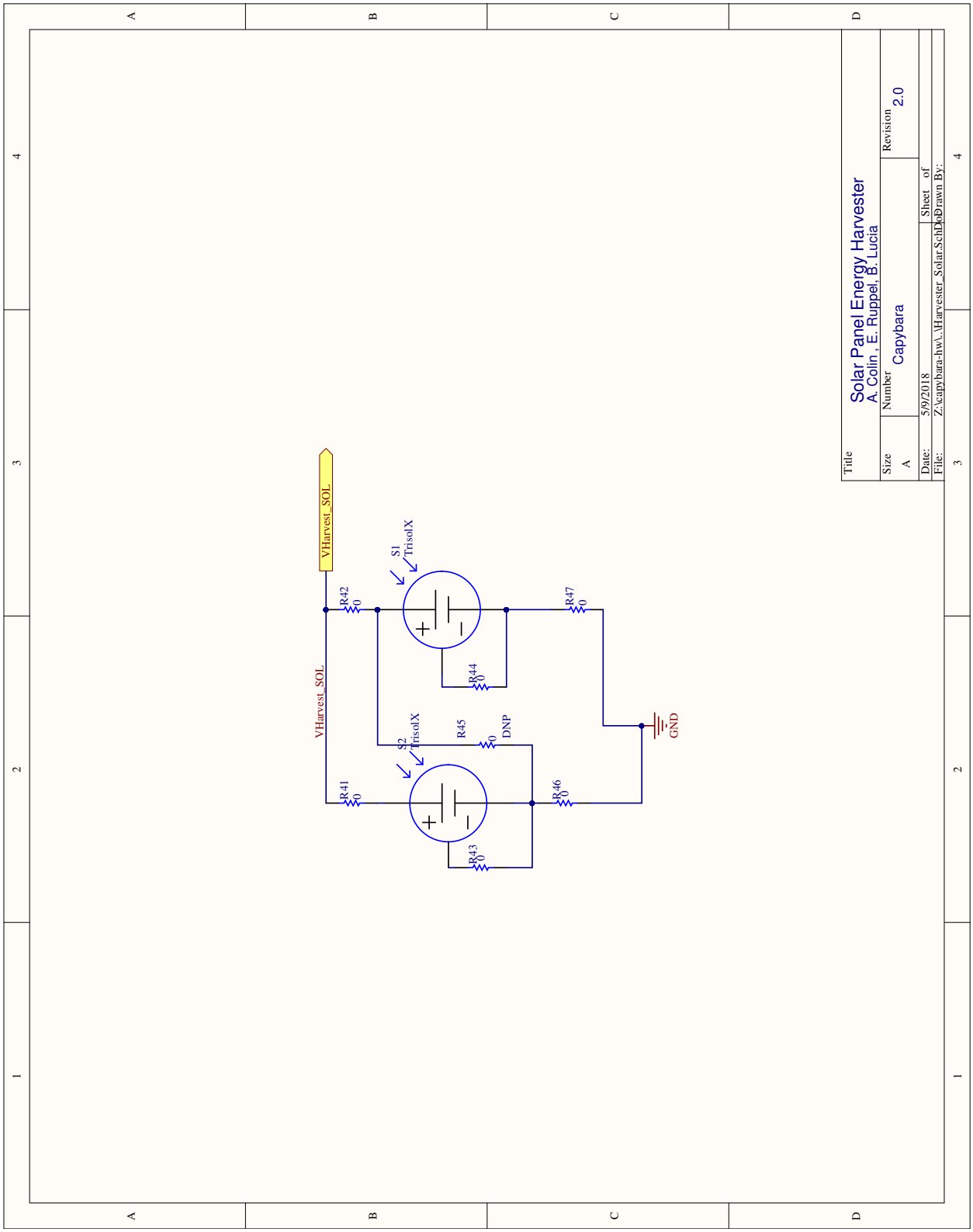

Title: Capybara Energy-Harvesting Platform
Designed by: A. Colin, E. Ruppel, B. Lucia
Data: 3/11/2017
Version: Capybara v2
U_Radio
Radio.SchDoc
U_MCU
MCU.SchDoc
U_Power
Power.SchDoc
U_Sensors
Sensors.SchDoc
U_CapBank
CapBank.SchDoc
U_Harvester_RF
Harvester_RF.SchDoc
Harvester_Solar
Harvester_Solar.SchDoc
P8
P9
P10
P11
P1A
P1B
P1C
MHDR1X7
MHDR1X8
MHDR1X3
MHDR1X2
JU1
JU2
F1
F2
F3
FIDUCIAL
FIDUCIAL
GND
VDD_MCU
BQOUT
SDA
SCL
VHarvest
VHarvest_EXT
VHarvest_SOL
VHarvest_RF
VBANK
CP_EN
BOOST_SW
VBOOST_OK
VBANK_OK
RAD_TX
RAD_RX
RAD_SW
RAD_RST
RAD_IO0
RAD_IO1
AUX_5
AUX_4
AUX_3
AUX_2
AUX_1
AUX_0
DEV_TX
DEV_RX
VHarvest
RF_TX
RF_RX
RF_RX_BITLINE
RF_RX_EN
VDD_MIC
MIC_OUT
SENSE_SW
PHOTO_SW
PHOTO_SENSE
LSM_INT_2
LSM_INT_1
APDS_SW
APDS_INT
LPS_INT
HMC_DRDY
SCL
SDA
FXL_RST
FXL_INT
OPEN_0
OPEN_1
OPEN_2
OPEN_3
BANK_INT

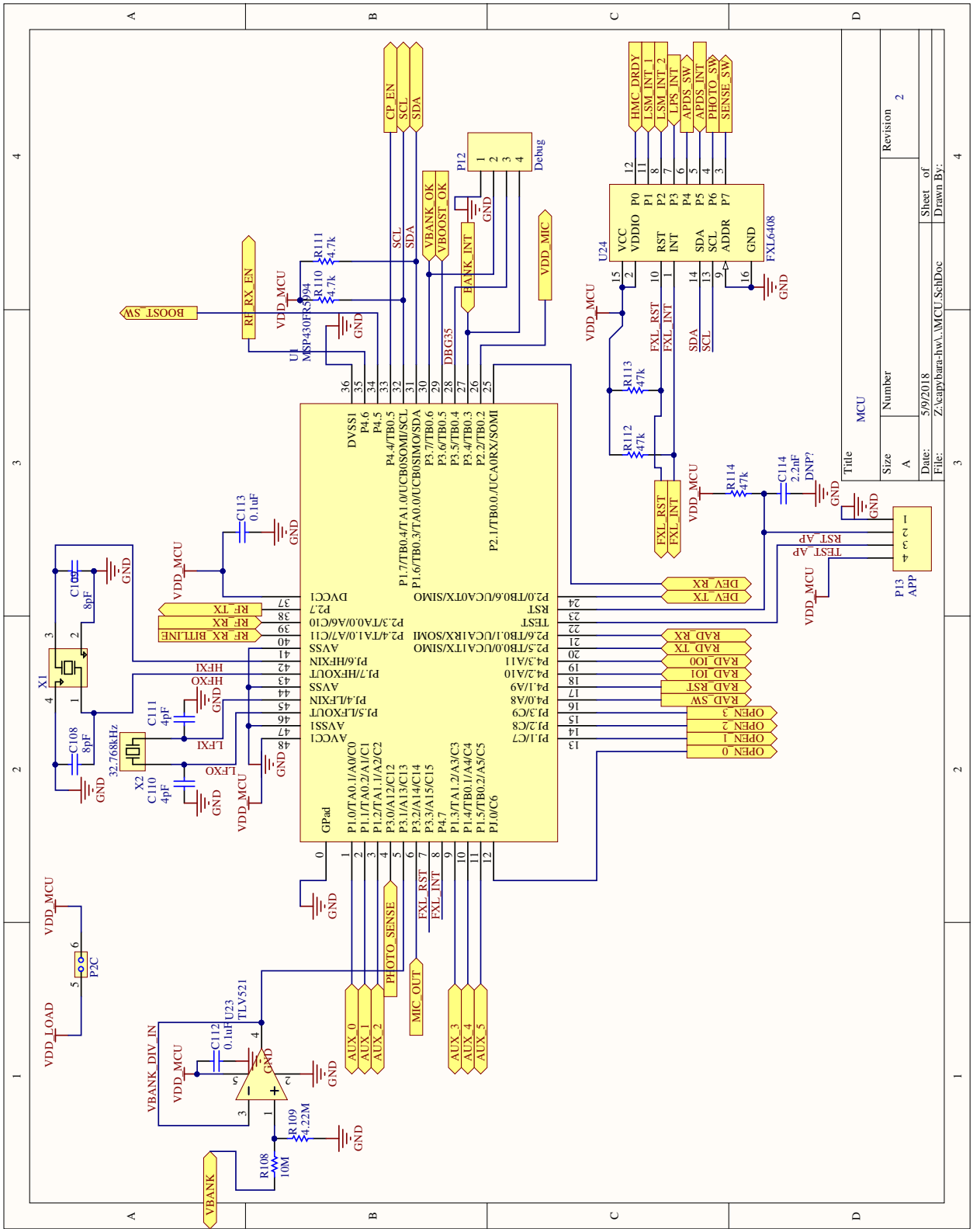

Reconfigurable Capacitor Bank
Title
Size
A
Number
Revision
Date: 5/9/2018
File: Z:\capybara-hw\..\CapBank.SchDoc
Sheet of
Drawn By:
U_CapBank_Introspection
BANK_INT
SCL
SDA
BANK_1
BANK_2
BANK_3
BANK_4
U_CapBank_Supercaps
VSUPERCAP
VDD_LOAD
VCAP
GND
VDD_SW_0
VDD_SW_1
VDD_SW_2
VDD_SW_3
G_SW0
G_SW1
G_SW2
G_SW3
OPEN_0
OPEN_1
OPEN_2
OPEN_3
VLTCH_0
VLTCH_1
VLTCH_2
VLTCH_3
781-SI2302CDS-E3
BAS40-02V-V-G-08
PMV30XPEAR
DMN6SD8L-7
PMV30XPEAR
DNP

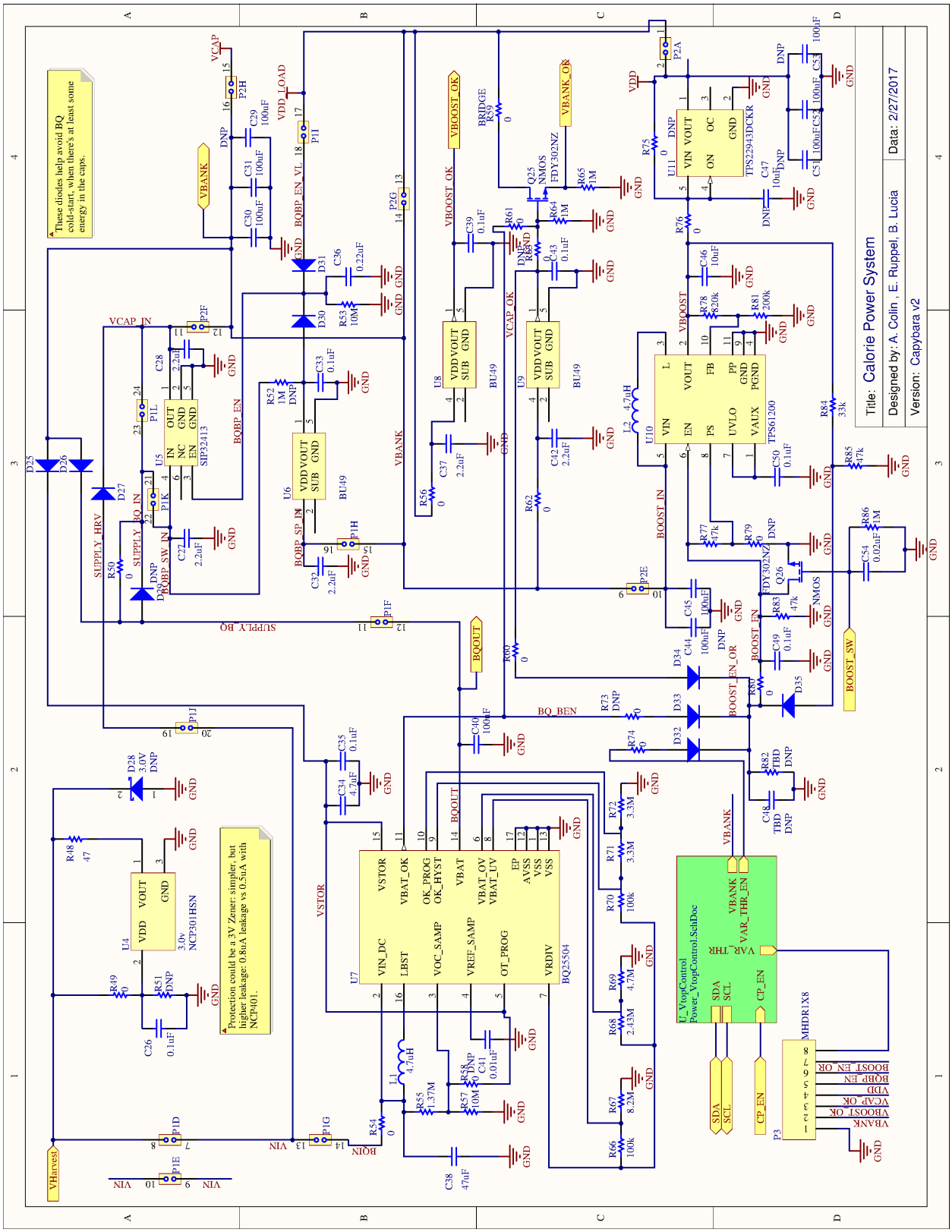

Title
Capacitor Bank Introspection
Size
A
Number
Revision
2
Date: 5/9/2018
Sheet of
File: Z:\capybara-hw\..\CapBank_Introspection.SchDoc
Drawn By:
VDD_MCU
C115
0.1uF
GND
SCL
SDA
BANK_1
BANK_2
BANK_3
BANK_4
P14
R120
20M
R121
20M
R122
20M
R123
20M
U25
ADDR
ALERT/RDY
GND
AIN0
AIN1
SCL
SDA
VDD
AIN3
AIN2
ADS1115
R115
10k
BANK_INT
R116
4.7M
R117
4.7M
R118
4.7M
R119
4.7M
VB1_DIV
VB2_DIV
VB3_DIV
VB4_DIV

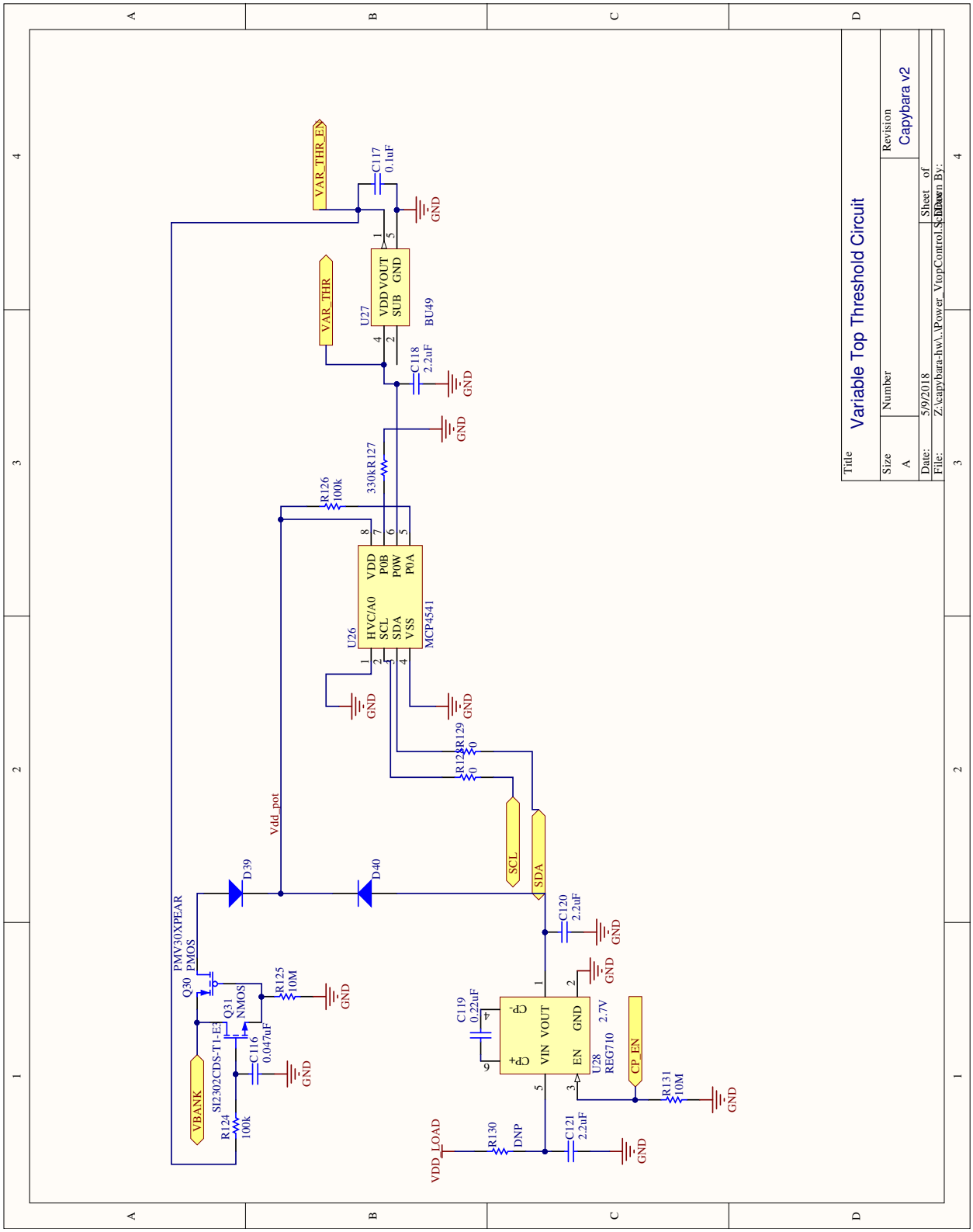

To save space, footprints are overlapped, and three pairs of pads overlap. Two of the overlapping pairs have the same net name, but one does not. So, the TOP and BOTTOM jumpers connect the overlapping pad to one of two nets: either for using the top supercap or the bottom supercap, respectively.
<---- Option A: Passive balancing using resistors.
<---- Option B: Active balancing using special-purpose SAB FETs.
R33 top DNP 0
R35 bottom,series DNP 0
R34 *,series DNP 0
R36 parallel 0
R37 parallel 0
R38 DNP 10k
R39 DNP 10k
R40 DNP 0
C20
C21A
C21B
U3A ALD900xx
U3B ALD900xx
U3C ALD900xx
IC/V- V+
SC_COM
VSUPERCAP
GND
Some dual supercaps expose 2*2 terminals, others expose only 2-common. For the former, the SERIES and PARALLEL jumpers select the nets.
Zero-ohm bridge to disconnect active balancing without removing the IC, when using a parallel configuration (or a single supercap). ----->
Title
Size A
Number
Revision
Date: 5/9/2018
Sheet of
File: Z:\capybara-hw\..\CapBank_Supercaps.SchDoc
Drawn By:

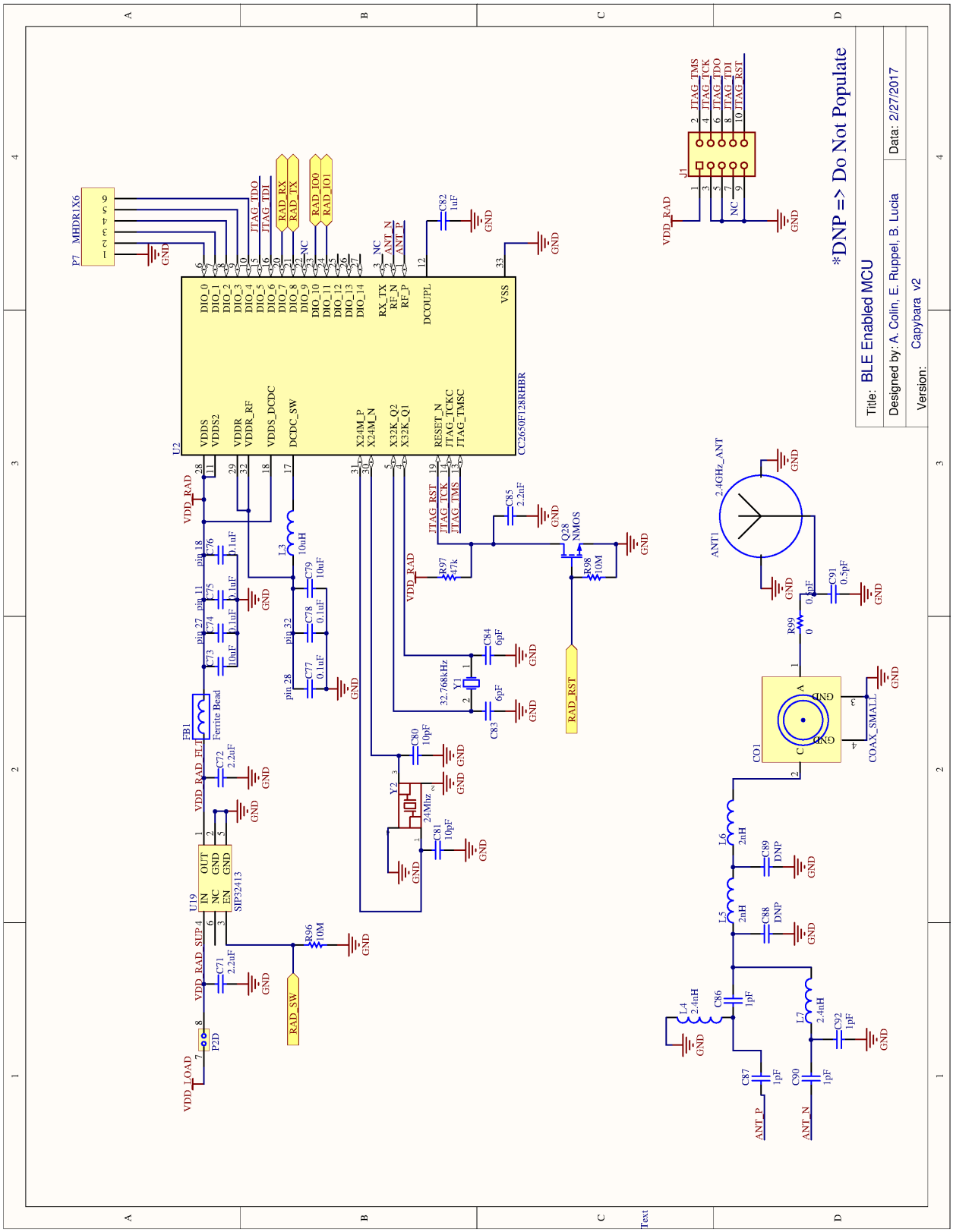

RF Energy Harvester and Backscatter Comms
From WISP5 by A. Parks, A. Sample, et. al.
Title
Size
A
Number
Capybara
Revision
2.0
Date:
5/9/2018
Sheet of
File:
Z:\capybara-hw\..\Harvester_RF.SchDoc
Drawn By:
RF_RX_EN
RF_RX
C96
10000pF
U20
RF_RX_BITLINE
RF_BITLINE_LPF
R102
220k
C100
1000pF
R101
22k
C97
33pF
R100
560k
VHarvest_RF
C93
33pF
D37
HSMS-285C
C98
10pF
D38
HSMS-285C
GND
C94
33pF
C99
1pF
L8
22nH
C95
2.2pF
RF_ANT
Q29
BF2040
J2
MMCX
RF_TX

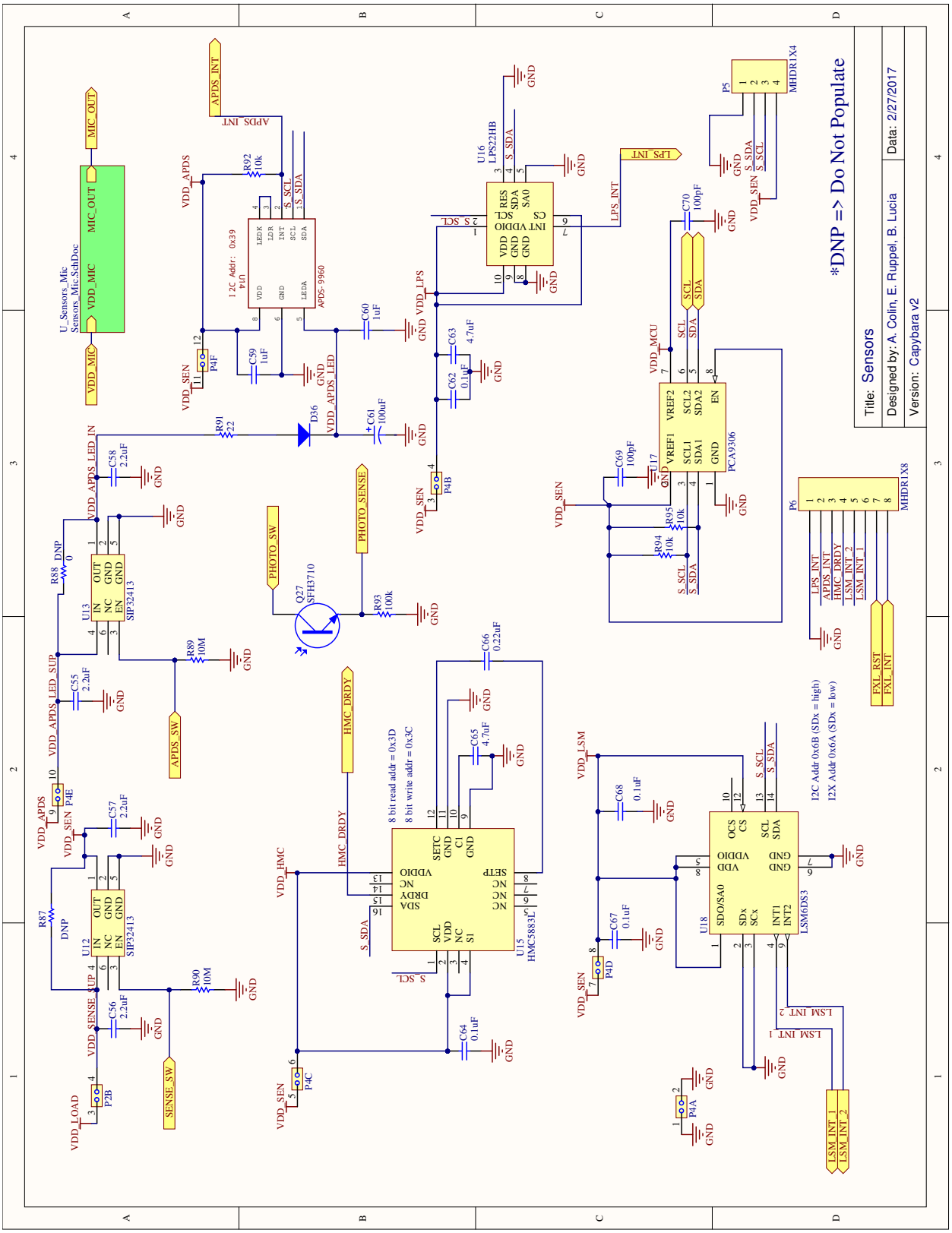

Title
Solar Panel Energy Harvester
A. Colin , E. Ruppel, B. Lucia
Size
A
Number
Capybara
Revision
2.0
Date:
5/9/2018
Sheet of
File:
Z:\capybara-hw\..\Harvester_Solar.SchDoc
Drawn By:
VHarvest_SOL
VHarvest_SOL
S1
TrisolX
S2
TrisolX
R42
0
R47
0
R44
0
R45
0
DNP
R41
0
R46
0
R43
0
GND

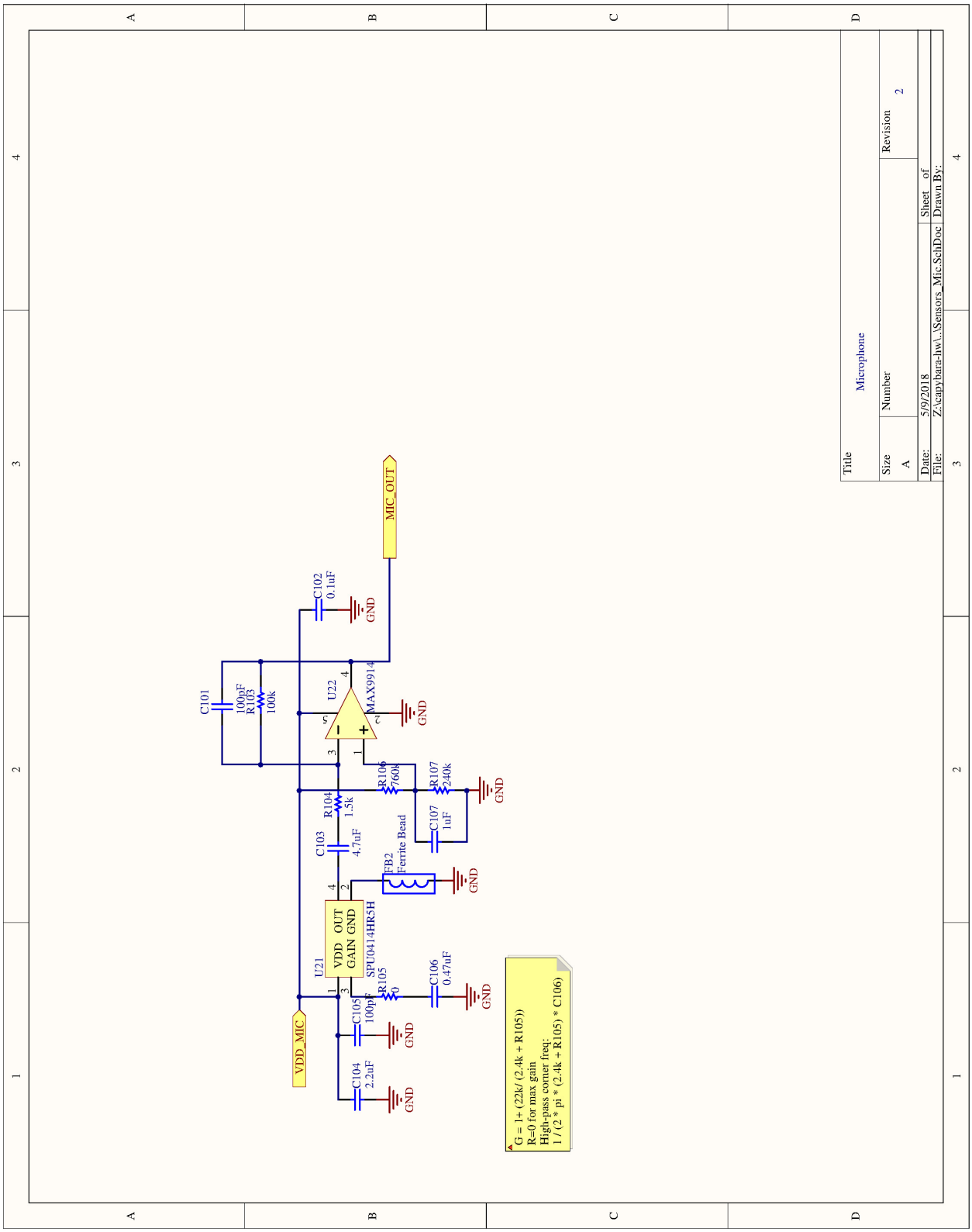

U1
MSP430FR5994
P12
Debug
U24
FXL6408
P13
APP
U23
TLV521
X1
X2
32.768kHz
P2C
R108
10M
R109
4.22M
R110
4.7k
R111
4.7k
R112
47k
R113
47k
R114
47k
C108
8pF
C110
4pF
C111
4pF
C112
0.1uF
C113
0.1uF
C114
2.2nF
DNP?
VDD_MCU
VDD_LOAD
VBANK
VBANK_DIV_IN
BOOST_SW
RF_RX_EN
CP_EN
SCL
SDA
VBANK_OK
VBOOST_OK
HANK_INT
VDD_MIC
DBG35
HMC_DRDY
LSM_INT_1
LSM_INT_2
LPS_INT
APDS_SW
APDS_INT
PHOTO_SW
SENSE_SW
FXL_RST
FXL_INT
MIC_OUT
PHOTO_SENSE
AUX_0
AUX_1
AUX_2
AUX_3
AUX_4
AUX_5
DEV_TX
DEV_RX
RAD_TX
RAD_RX
RAD_IO0
RAD_IO1
RAD_RST
RAD_SW
OPEN_0
OPEN_1
OPEN_2
OPEN_3
RF_TX
RF_RX
RF_RX_BITLINE
HFXI
HFXO
LFXI
LFXO
TEST_AP
RST_AP
GND
Title
MCU
Size
A
Number
Revision
2
Date:
5/9/2018
File:
Z:\capybara-hw\..\MCU.SchDoc
Sheet of
Drawn By:


These diodes help avoid BQ cold-start, when there's at least some energy in the caps.
Protection could be a 3V Zener: simpler, but higher leakage: 0.8uA leakage vs 0.5uA with NCP401.
Title: Calorie Power System
Designed by: A. Colin , E. Ruppel, B. Lucia
Data: 2/27/2017
Version: Capybara v2
VHarvest
VIN
VSTOR
BQOUT
BQ_BEN
BOOST_SW
BOOST_EN_OR
BOOST_EN
VBANK
VCAP
VCAP_IN
VDD_LOAD
VBOOST
VBOOST_OK
VCAP_OK
VBANK_OK
SUPPLY_HRV
SUPPLY_BQ_IN
BQBP_SW_IN
BQBP_SP_IN
BQBP_EN
BQBP_EN_VL
BOOST_IN
SDA
SCL
CP_EN
BQ25504
TPS61200
TPS22943DCKR
SIP32413
NCP301HSN
FDY302NZ
MHDR1X8
GND


Variable Top Threshold Circuit
Capybara v2
Size
A
Number
Revision
Date: 5/9/2018
Sheet of
File: Z:\capybara-hw\..\Power_VtopControl.SchDoc
Drawn By:
VAR_THR_EN
VAR_THR
C117
0.1uF
U27
VDD VOUT
SUB GND
BU49
C118
2.2uF
R127
330k
R126
100k
U26
HVC/A0 VDD
SCL P0B
SDA P0W
VSS P0A
MCP4541
R128
0
R129
0
SCL
SDA
C120
2.2uF
Vdd_pot
D39
D40
PMV30XPEAR
PMOS
Q30
Q31
NMOS
SI2302CDS-T1-E3
R125
10M
C116
0.047uF
R124
100k
VBANK
C119
0.22uF
CP+
CP-
VIN VOUT
EN GND
U28
REG710
2.7V
CP_EN
R131
10M
R130
DNP
C121
2.2uF
VDD_LOAD
GND

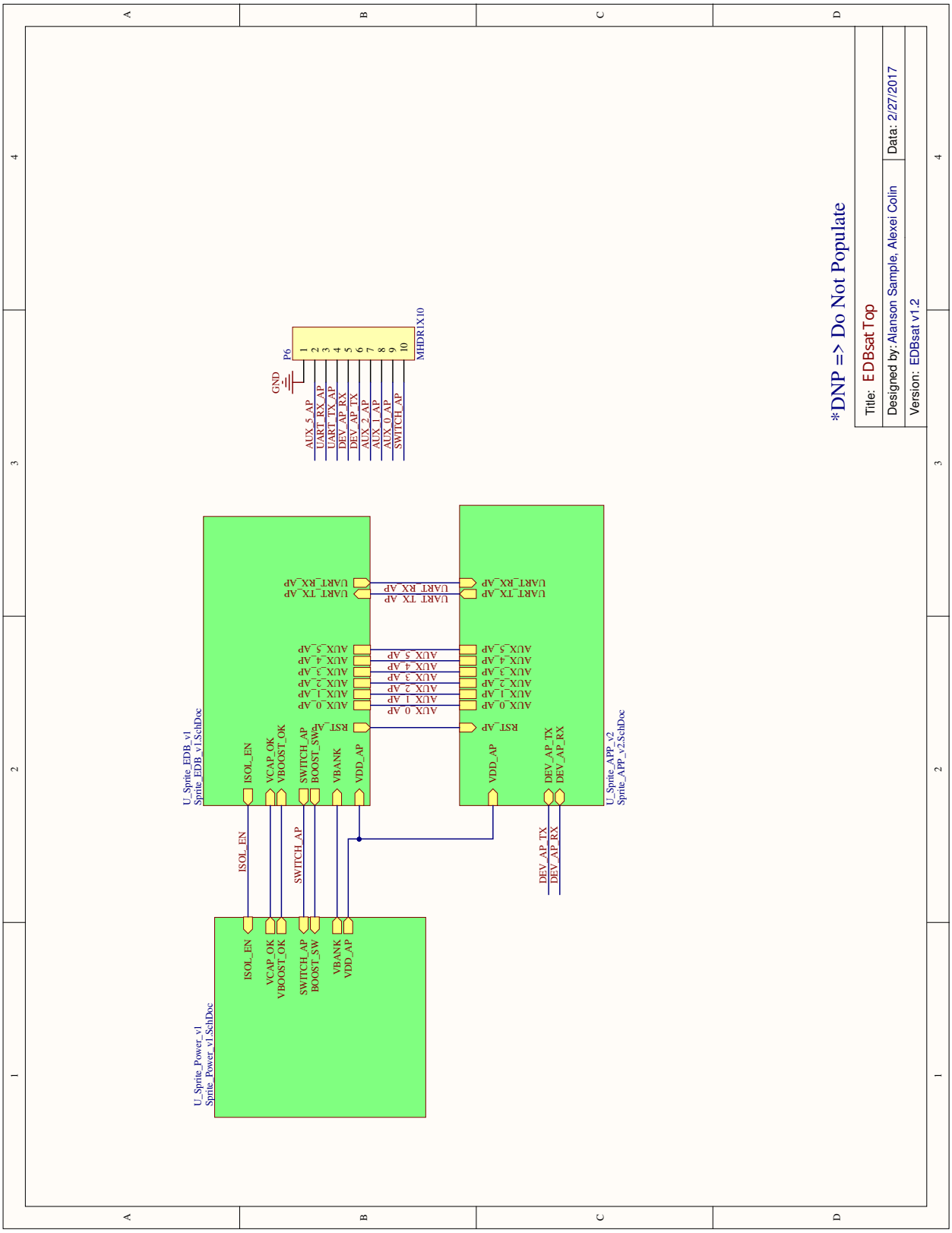

*DNP => Do Not Populate
Title: BLE Enabled MCU
Designed by: A. Colin, E. Ruppel, B. Lucia
Data: 2/27/2017
Version: Capybara v2
U2
CC2650F128RHBR
U19
SIP32413
J1
ANT1
2.4GHz_ANT
CO1
COAX_SMALL
Y1
32.768kHz
Y2
24Mhz
Q28
NMOS
FB1
Ferrite Bead
P7
MHDR1X6
P2D
VDD_RAD
VDD_LOAD
VDD_RAD_SUP
VDD_RAD_FLT
RAD_SW
RAD_RST
JTAG_TDO
JTAG_TDI
RAD_RX
RAD_TX
RAD_IO0
RAD_IO1
JTAG_TMS
JTAG_TCK
JTAG_TDO
JTAG_TDI
JTAG_RST
ANT_P
ANT_N
GND

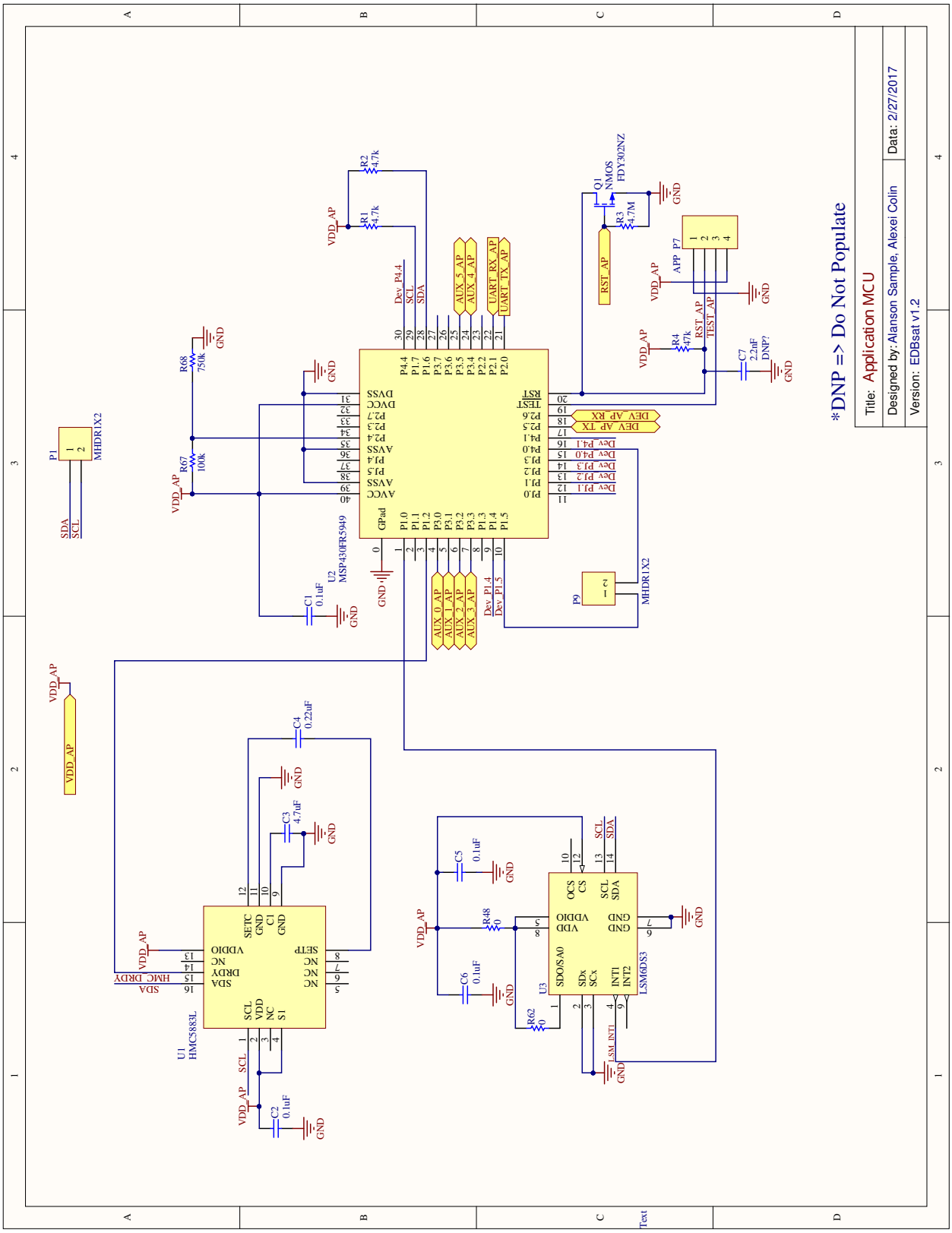

*DNP => Do Not Populate
Title: Sensors
Designed by: A. Colin, E. Ruppel, B. Lucia
Data: 2/27/2017
Version: Capybara v2
U_Sensors_Mic
Sensors_Mic.SchDoc
VDD_MIC
MIC_OUT
APDS_INT
VDD_APDS
R92 10k
S_SCL
S_SDA
U14
I2C Addr: 0x39
APDS-9960
C59 1uF
C60 1uF
VDD_SEN
P4F
VDD_APDS_LED
D36
R91 22
C61 100uF
VDD_LPS
U16
LPS22HB
C63 4.7uF
C62 0.1uF
P4B
LPS_INT
C70 100pF
P5
MHDR1X4
VDD_MCU
U17
PCA9306
C69 100pF
R95 10k
R94 10k
P6
MHDR1X8
LPS_INT
APDS_INT
HMC_DRDY
LSM_INT_2
LSM_INT_1
FXL_RST
FXL_INT
VDD_APDS_LED_IN
C58 2.2uF
R88 DNP 0
U13
SIP32413
VDD_APDS_LED_SUP
C55 2.2uF
R89 10M
APDS_SW
PHOTO_SW
Q27
SFH3710
PHOTO_SENSE
R93 100k
C66 0.22uF
C65 4.7uF
HMC_DRDY
8 bit read addr = 0x3D
8 bit write addr = 0x3C
U15
HMC5883L
C64 0.1uF
VDD_HMC
VDD_APDS
P4E
C57 2.2uF
R87 DNP
U12
SIP32413
VDD_SENSE_SUP
C56 2.2uF
R90 10M
SENSE_SW
VDD_LOAD
P2B
VDD_SEN
P4C
VDD_LSM
C68 0.1uF
C67 0.1uF
P4D
U18
LSM6DS3
I2C Addr 0x6B (SDx = high)
I2X Addr 0x6A (SDx = low)
P4A
LSM_INT_1
LSM_INT_2

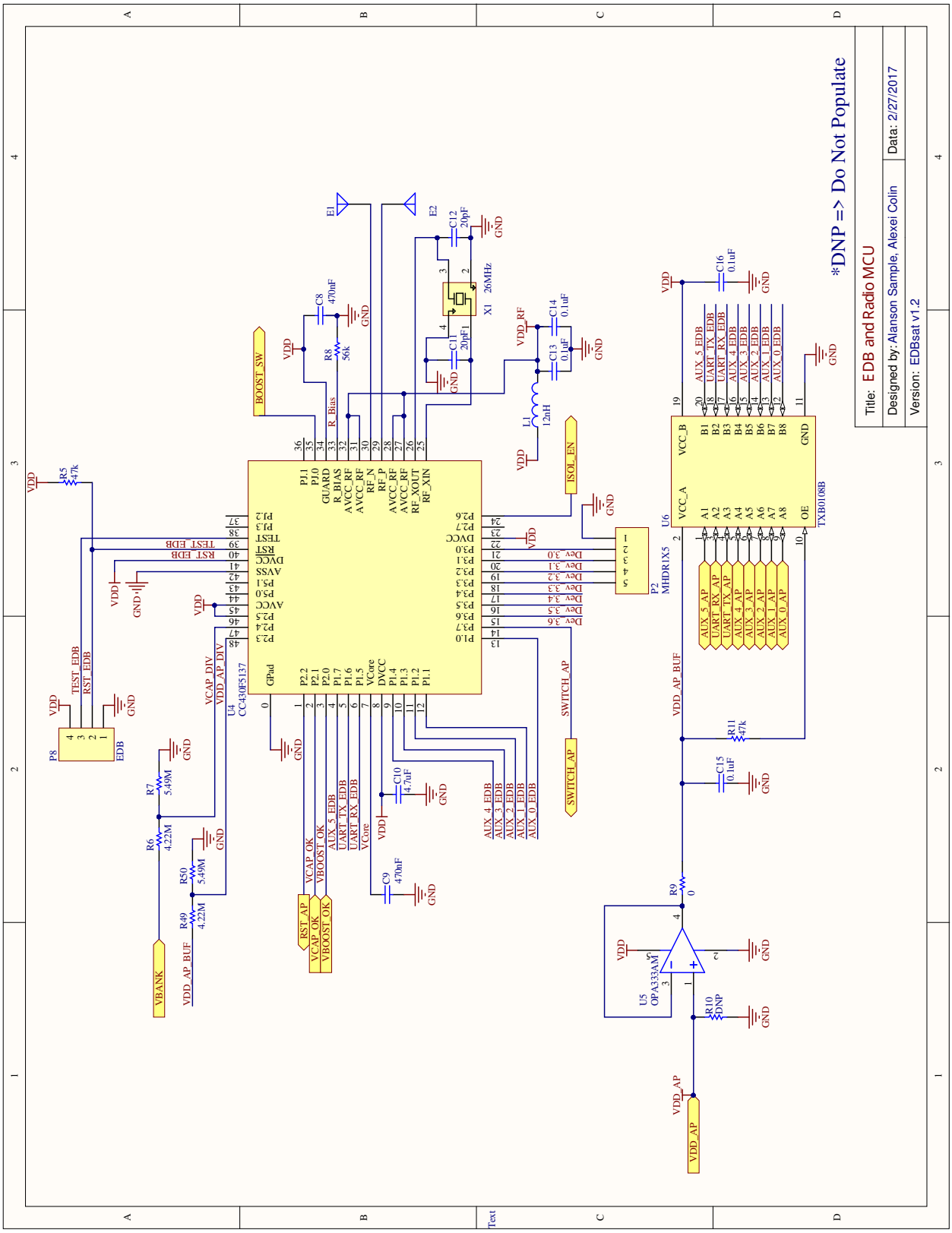

Microphone
Size A
Number
Revision 2
Date: 5/9/2018
Sheet of
File: Z:\capybara-hw\..\Sensors_Mic.SchDoc
Drawn By:
MIC_OUT
VDD_MIC
C101 100nF
R103 100k
C102 0.1uF
U22 MAX9914
R104 1.5k
R106 760k
R107 240k
C103 4.7uF
C107 1uF
FB2 Ferrite Bead
U21 SPU0414HR5H
VDD OUT GAIN GND
R105 0
C105 100pF
C106 0.47uF
C104 2.2uF
GND
G = 1+ (22k/ (2.4k + R105))
R=0 for max gain
High-pass corner freq:
1 / (2 * pi * (2.4k + R105) * C106)

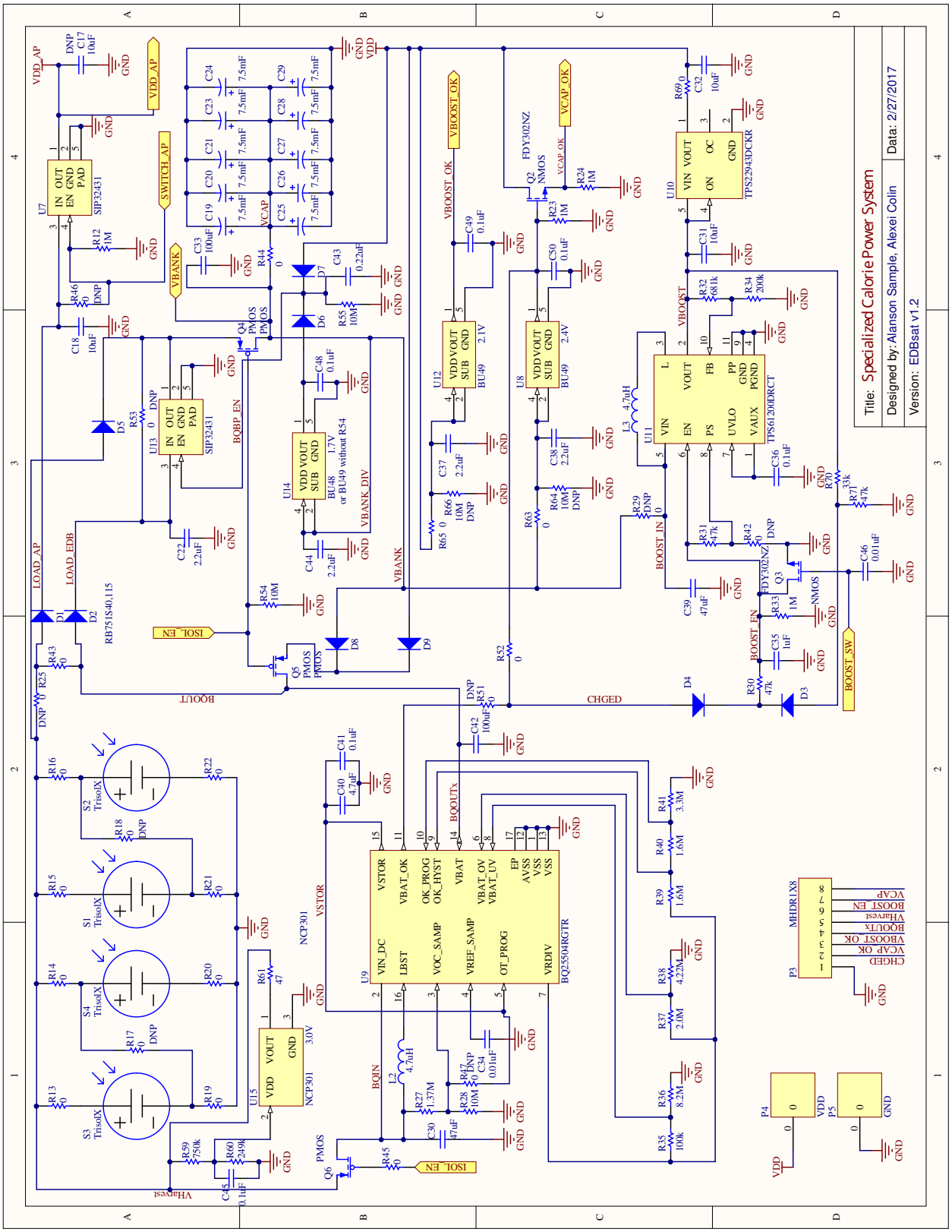
Appendix B

EDBsat Hardware Schematics


*DNP => Do Not Populate
Title: EDBsatTop
Designed by: Alanson Sample, Alexei Colin
Data: 2/27/2017
Version: EDBsat v1.2
P6
MHDR1X10
GND
AUX_5_AP
UART_RX_AP
UART_TX_AP
DEV_AP_RX
DEV_AP_TX
AUX_2_AP
AUX_1_AP
AUX_0_AP
SWITCH_AP
U_Sprite_EDB_v1
Sprite_EDB_v1.SchDoc
U_Sprite_APP_v2
Sprite_APP_v2.SchDoc
U_Sprite_Power_v1
Sprite_Power_v1.SchDoc
ISOL_EN
VCAP_OK
VBOOST_OK
SWITCH_AP
BOOST_SW
VBANK
VDD_AP
RST_AP
DEV_AP_TX
DEV_AP_RX


*DNP => Do Not Populate
Title: Application MCU
Designed by: Alanson Sample, Alexei Colin
Data: 2/27/2017
Version: EDBsat v1.2
U2 MSP430FR5949
U1 HMC5883L
U3 LSM6DS3
P1 MHDR1X2
P9 MHDR1X2
P7
Q1 NMOS FDY302NZ
R1 4.7k
R2 4.7k
R3 47M
R4 47k
R67 100k
R68 750k
R62 0
R48 0
C1 0.1uF
C2 0.1uF
C3 4.7uF
C4 0.22uF
C5 0.1uF
C6 0.1uF
C7 2.2nF DNP?
VDD_AP
GND
SDA
SCL
RST_AP
TEST_AP
APP
AUX_0_AP
AUX_1_AP
AUX_2_AP
AUX_3_AP
AUX_4_AP
AUX_5_AP
UART_RX_AP
UART_TX_AP
DEV_AP_TX
DEV_AP_RX
Dev_P4.4
Dev_P1.4
Dev_P1.5
LSM_INT1
HMC_DRDY


*DNP => Do Not Populate
Title: EDB and Radio MCU
Designed by: Alanson Sample, Alexei Colin
Data: 2/27/2017
Version: EDBsat v1.2
U4 CC430F5137
U5 OPA333AM
U6 TXB0108B
P2 MHDR1X5
P8 EDB
X1 26MHz
L1 12nH
E1
E2
VBANK
VDD_AP_BUF
VDD_AP
VDD_AP_DIV
VCAP_DIV
BOOST_SW
ISOL_EN
SWITCH_AP
RST_AP
VCAP_OK
VBOOST_OK
TEST_EDB
RST_EDB
AUX_5_EDB
UART_TX_EDB
UART_RX_EDB
AUX_4_EDB
AUX_3_EDB
AUX_2_EDB
AUX_1_EDB
AUX_0_EDB
VDD_RF
VCore
R_Bias
R5 47k
R6 4.22M
R7 5.49M
R49 4.22M
R50 5.49M
R8 56k
R9 0
R10 DNP
R11 47k
C8 470nF
C9 470nF
C10 4.7uF
C11 20pF
C12 20pF
C13 0.1uF
C14 0.1uF
C15 0.1uF
C16 0.1uF
VDD
GND
GND1
Text


Title: Specialized Calorie Power System
Designed by: Alanson Sample, Alexei Colin
Data: 2/27/2017
Version: EDBsat v1.2

Bibliography

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

[2] H. Aantjes, A. Y. Majid, P. Paweczak, J. Tan, A. Parks, and J. R. Smith. Fast downstream to many (computational) RFIDs. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[3] G. A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass, 1986.

[4] Alexei Colin and Ragunathan (Raj) Rajkumar. Technical report, Carnegie Mellon University, 2015.

[5] M. P. Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*, pages 1–12. IEEE, 2016.

[6] Apple, Inc. iBeacon.

[7] Avago Technologies. APDS-9960. `https://web.archive.org/web/20180423011530/https://cdn.sparkfun.com/assets/learn_tutorials/3/2/1/Avago-APDS-9960-datasheet.pdf`, 2018.

[8] AVX. BestCap Ultra-low ESR High Power Pulse Supercapacitors. `https://web.archive.org/web/20170817155017/http://catalogs.avx.com/BestCap.pdf`, 2018.

[9] AVX. CoreCap NbO Ceramic Multianode Chip Capacitors. `https://web.archive.org/web/20180419183400/http://www.farnell.com/datasheets/86284.pdf`, 2018.

[10] AVX. F93 Series Standard Tantalum Capacitors. `https://web.archive.org/web/20171007170642/http://datasheets.avx.com/F93.pdf`, 2018.

[11] AVX. OxiCap NOJ Series Niobium Standar and Low Profile Niobium Oxide Capacitors. `https://web.archive.org/web/20170115223958/http://datasheets.avx.com:80/NOJ.pdf`, 2018.

[12] A. Badam, E. Skiani, R. Chandra, J. Dutra, A. Ferrese, S. Hodges, P. Hu, J. Meinershagen, T. Moscibroda, and B. Priyantha. Software defined batteries. pages 215–229. ACM Press, 2015.

[13] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[14] S. S. Baghsorkhi and C. Margiolas. Automating Efficient Variable-grained Resiliency for Low-power IoT Systems. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 38–49, New York, NY, USA, 2018. ACM.

[15] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[16] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. 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.

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

[18] T. W. Barr and S. Rixner. Medusa: Managing concurrency and communication in embedded systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 439–450, Berkeley, CA, USA, 2014. USENIX Association.

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

[20] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):299–316, 2000.

[21] N. A. Bhatti and L. Mottola. HarvOS: efficient code instrumentation for transiently-powered embedded sensing. pages 209–219. ACM Press, 2017.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[23] R. L. Bocchino, E. Gamble, K. P. Gostelow, and R. R. Some. Spot: A programming language for verified flight software. In *Proceedings of the 2014 ACM SIGAda Annual*

Conference on High Integrity Language Technology, HILT '14, pages 97–102, New York, NY, USA, 2014. ACM.

[24] R. Bondade, Y. Zhang, B. Wei, T. Gu, H. Chen, and D. B. Ma. Integrated Auto-Reconfigurable Power-Supply Network With Multidirectional Energy Transfer for Self-Reliant Energy-Harvesting Applications. *IEEE Transactions on Industrial Electronics*, 63(5):2850–2861, May 2016.

[25] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 83–98, New York, NY, USA, 2016. ACM.

[26] D. E. Boyle, M. E. Kiziroglou, P. D. Mitcheson, and E. M. Yeatman. Energy provision and storage for pervasive computing. *IEEE Pervasive Computing*, 15(4):28–35, 2016.

[27] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[28] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: an energy-aware runtime for computational rfid. In *Proc. USENIX NSDI*, pages 197–210, 2011.

[29] B. Campbell and P. Dutta. An energy-harvesting sensor architecture and toolkit for building monitoring and event detection. pages 100–109. ACM Press, 2014.

[30] Canan Dagdeviren, Zhou Li, and Zhong Lin Wang. Energy Harvesting from the Animal/Human Body for Self-Powered Electronics. *Annual Review of Biomedical Engineering*, 2017.

[31] A. Canino and Y. D. Liu. Proactive and adaptive energy-aware programming with mixed typechecking. pages 217–232. ACM Press, 2017.

[32] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 1–13, New York, NY, USA, 2006. ACM.

[33] G. Chen, H. Ghaed, R.-u. Haque, M. Wieckowski, Y. Kim, G. Kim, D. Fick, D. Kim, M. Seok, K. Wise, and others. A cubic-millimeter energy-autonomous wireless intraocular pressure monitor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 310–312. IEEE, 2011.

[34] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori. Determining Application-specific Peak Power and Energy Requirements for Ultra-low Power Processors. pages 3–16. ACM Press, 2017.

[35] Christopher England. Ceramic Capacitor Aging Made Simple. https://web.archive.org/web/20170919113441/https://www.johansondielectrics.com/downloads/Cap_Aging_rev_B.pdf, 2012.

[36] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, Mar. 2011.

[37] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

[38] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *SIGOPS Oper. Syst. Rev.*, 50(2):577–589, Mar. 2016.

[39] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 514–530. ACM, 2016.

[40] A. Colin and B. Lucia. Termination checking and task decomposition for task-based intermittent programs. In *International Conference on Compiler Construction*, Feb. 2018.

[41] A. Colin, E. Ruppel, and B. 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.

[42] Cornell Dubilier. Type AFK SMT Aluminum Electrolytic Capacitors. https://web.archive.org/web/20170612010102/http://www.cde.com/resources/catalogs/AFK.pdf, 2018.

[43] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[44] D. C. Daly, P. P. Mercier, M. Bhardwaj, A. L. Stone, Z. N. Aldworth, T. L. Daniel, J. Voldman, J. G. Hildebrand, and A. P. Chandrakasan. A Pulsed UWB Receiver SoC for Insect Motion Control. *IEEE Journal of Solid-State Circuits*, 45(1):153–166, Jan. 2010.

[45] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 140–151, New York, NY, USA, 2011. ACM.

[46] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. *ACM SIGPLAN Notices*, 47(6):475–486, 2012.

[47] S. DeBruin, B. Campbell, and P. Dutta. Monjolo: An energy-harvesting energy meter architecture. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 18. ACM, 2013.

[48] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 373–390, New York, NY, USA, 2013. ACM.

[49] A. Dementyev, J. Gummeson, D. Thrasher, A. Parks, D. Ganesan, J. R. Smith, and A. P. Sample. Wirelessly powered bistable display tags. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 383–386, New York, NY, USA, 2013.

[50] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 312–315, 2000.

[51] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical python. *Computers in Physics*, 10(3), May/June 1996.

[52] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

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

[54] Editor. Understanding TPMS Batteries. *Motor Age*, June 2014.

[55] EnOcean. Self-powered IoT. `https://www.enocean.com/`. Visited April, 2018.

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

[57] A. Gomez, L. Sigrist, T. Schalch, L. Benini, and L. Thiele. Efficient, long-term logging of rich data sensors using transient sensor nodes. *ACM Trans. Embed. Comput. Syst.*, 17(1):4:1–4:23, Sept. 2017.

[58] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[59] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 12–21. ACM, 2015.

[60] L. Gu and J. A. Stankovic. T-kernel: Providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 1–14, New York, NY, USA, 2006.

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

[62] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, volume 32, page 102. IEEE Computer Society, 2004.

[63] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[64] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, 1993.

[65] J. Hester, S. Lord, R. Halter, D. Kotz, J. Sorber, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, and K. Freeman. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. pages 216–229. ACM Press, 2016.

[66] J. Hester, L. Sitanayah, and J. 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.

[67] J. Hester and J. Sorber. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. pages 1–13. ACM Press, 2017.

[68] J. Hester, K. Storer, and J. Sorber. Timely Execution on Intermittently Powered Batteryless Sensors. Association for Computing Machinery, 2017.

[69] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, and J. Sorber. Persistent Clocks for Batteryless Sensing Devices. *ACM Transactions on Embedded Computing Systems*, 15(4):1–28, Aug. 2016.

[70] M. Hicks. Clank: Architectural Support for Intermittent Computation. pages 228–240. ACM Press, 2017.

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

[72] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[73] A. S. Holmes, G. Hong, K. R. Pullen, and K. R. Buffard. Axial-flow microturbine with electromagnetic generator: design, CFD simulation, and prototype demonstration. In *17th IEEE International Conference on Micro Electro Mechanical Systems. Maastricht MEMS 2004 Technical Digest*, pages 568–571, 2004.

[74] Illinois Capacitor. DCN Super Capacitors. `https://web.archive.org/web/20170624105617/http://www.illinoiscapacitor.com/pdf/seriesDocuments/DCN%20series.pdf`, 2018.

[75] Intel. RAPL Power Meter. `https://01.org/rapl-power-meter`. Visited April, 2018.

[76] H. Jayakumar, A. Raha, and V. Raghunathan. Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pages 330–335. IEEE, 2014.

[77] R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 81–90. IEEE, 2006.

[78] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 65. IEEE Press, 2005.

[79] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[80] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-08-14].

[81] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. 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.

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

[83] H. Kalantarian and M. Sarrafzadeh. Pedometers Without Batteries: An Energy Harvesting Shoe. *IEEE Sensors Journal*, 16(23):8314–8321, Dec. 2016.

[84] M. Kambadur and M. A. Kim. NRG-loops: adjusting power from within applications. pages 206–215. ACM Press, 2016.

[85] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. pages 661–676. ACM Press, 2013.

[86] M. E. Karagozler, I. Poupyrev, G. K. Fedder, and Y. Suzuki. Paper generators: harvesting energy from touching, rubbing and sliding. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 23–30. ACM, 2013.

[87] I. Kazi, P. Meinerzhagen, P.-E. Gaillardon, D. Sacchetto, A. Burg, and G. De Micheli. A ReRAM-based non-volatile flip-flop with sub-V T read and CMOS voltage-compatible write. In *New Circuits and Systems Conference (NEWCAS), 2013 IEEE 11th International*, pages 1–4. IEEE, 2013.

[88] KEMET. T491 Industrial Grade MnO2 Capacitors. `https://web.archive.org/web/20180419174511/https://content.kemet.com/datasheets/KEM_T2005_T491.pdf`, 2018.

[89] KEMET. T52X/T530 Polymer Electrolytic Capacitors. `https://web.archive.org/web/20180419174646/https://content.kemet.com/datasheets/KEM_T2076_T52X-530.pdf`, 2018.

[90] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, Apr. 2015.

[91] S. Kim, R. Vyas, J. Bito, K. Niotaki, A. Collado, A. Georgiadis, and M. M. Tentzeris. Ambient RF Energy-Harvesting Technologies for Self-Sustainable Standalone Wireless Sensor Platforms. *Proceedings of the IEEE*, 102(11):1649–1666, Nov. 2014.

[92] Y.-J. Kim, H. S. Bhamra, J. Joseph, and P. P. Irazoqui. An Ultra-Low-Power RF Energy-Harvesting Transceiver for Multiple-Node Sensor Application. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(11):1028–1032, Nov. 2015.

[93] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 671–690, New York, NY, USA, 2010. ACM.

[94] Y. Kwon, K. Mechitov, and G. Agha. *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, chapter Design and Implementation of a Mobile Actor Platform for Wireless Sensor Networks, pages 276–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[95] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. 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.

[96] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and others. TinyOS: An operating system for sensor networks. *Ambient intelligence*, 35:115–148, 2005.

[97] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM.

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

[99] U. Liqat, Z. Bankovic, P. López-García, and M. V. Hermenegildo. Inferring energy bounds statically by evolutionary analysis of basic blocks. *CoRR*, abs/1601.02800, 2016.

[100] X. Lu and S.-H. Yang. Thermal energy harvesting for WSNs. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 3045–3052, Oct. 2010.

[101] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM SIGPLAN Notices*, volume 50, pages 575–585. ACM, 2015.

[102] G. Lukosevicius, A. R. Arreola, and A. S. Weddell. Using Sleep States to Maximize the Active Time of Transient Computing Systems. pages 31–36. ACM Press, 2017.

[103] N. A. Lynch and M. Fischer. On describing the behavior and implementation of distributed systems, 1981.

[104] K. Ma, X. Li, S. Li, Y. Liu, J. J. Sampson, Y. Xie, and V. Narayanan. Nonvolatile processor architecture exploration for energy-harvesting applications. *IEEE Micro*, 35(5):32–40, 2015.

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

[106] K. Maeng, A. Colin, and B. 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.

[107] M. Magno and D. Boyle. Wearable Energy Harvesting: From body to battery. In *2017 12th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pages 1–6, Apr. 2017.

[108] R. Margolies, P. Kinget, I. Kymissis, G. Zussman, M. Gorlatova, J. Sarik, G. Stanje, J. Zhu, P. Miller, M. Szczodrak, B. Vigraham, and L. Carloni. Energy-Harvesting Active Networked Tags (EnHANTs): Prototyping and Experimentation. *ACM Transactions on Sensor Networks*, 11(4):1–27, Nov. 2015.

[109] P. Martin, Z. Charbiwala, and M. Srivastava. DoubleDip: Leveraging thermoelectric harvesting for low power monitoring of sporadic water use. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 225–238. ACM, 2012.

[110] A. Mirhoseini, B. D. Rouhani, E. Songhori, and F. Koushanfar. Chime: Checkpointing Long Computations on Interm ittently Energized IoT Devices. *IEEE Transactions on Multi-Scale Computing Systems*, 2(4):277–290, Oct. 2016.

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

[112] P. Mitcheson, E. Yeatman, G. Rao, A. Holmes, and T. Green. Energy Harvesting From Human and Machine Motion for Wireless Electronic Devices. *Proceedings of the IEEE*, 96(9):1457–1486, Sept. 2008.

[113] J. Morse, S. Kerrison, and K. Eder. On the infeasibility of analysing worst-case dynamic energy. *CoRR*, abs/1603.02580, 2016.

[114] Murata. Supercapacitor (EDLC) DMHA14R5V353M4ATA0. `https://web.archive.org/web/20180419182024/https://gateway.ipfs.io/ipfs/QmUbhKB6a5PX7PBXAH7dxT3361dpKToymskpqwRNdWfG6U/DMHA14R5V353M4ATA0_Ref_Sheet.pdf`, 2018.

[115] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith. Towards Battery-Free HD Video Streaming.

[116] S. Naderiparizi, A. N. Parks, Z. Kapetanovic, B. Ransford, and J. R. Smith. Wispcam: A battery-free rfid camera. In *2015 IEEE International Conference on RFID (RFID)*, pages 166–173. IEEE, 2015.

[117] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *ASPLOS*, Mar. 2012.

[118] NASA. Estimating the Temperature of a Flat Plate in Low Earth Orbit. `https://web.archive.org/web/20170415081719/https://www.grc.nasa.gov/www/k-12/Numbers/Math/Mathematical_Thinking/estimating_the_temperature.htm`, 2017.

[119] S. K. Nayar, D. C. Sims, and M. Fridberg. Towards self-powered cameras. In *Computational Photography (ICCP), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.

[120] Nippon Chemi-con. Alchip MZJ Series Surface Mount Aluminum Electrolytic Capacitors. `https://web.archive.org/web/20180419175738/https://gateway.ipfs.io/ipfs/QmQeSXcbDwHjcmz1TR4CVopmn4ffbkyUdzfWRqcpqdFx2n/nippon-chemi-con-alchip-mzj.pdf`, 2018.

[121] H. Nishimoto, Y. Kawahara, and T. Asami. Prototype implementation of ambient RF energy harvesting wireless sensor networks. In *Sensors, 2010 IEEE*, pages 1282–1287. IEEE, 2010.

[122] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but No Force: Efficient Hardware Undo+ Redo Logging for Persistent Memory Systems. pages pp. 336–349, Vienna, Austria, Feb. 2018.

[123] L. C. Olsen, P. Cabauy, and B. J. Elkind. Betavoltaic power sources. *Physics Today*, 65(12):35–38, Nov. 2012.

[124] N. Onizawa, A. Mochizuki, A. Tamakoshi, and T. Hanyu. Sudden Power-Outage Resilient In-Processor Checkpointing for Energy-Harvesting Nonvolatile Processors. *IEEE Transactions on Emerging Topics in Computing*, 5(2):151–163, Apr. 2017.

[125] G. Orecchini, L. Yang, M. M. Tentzeris, and L. Roselli. Wearable battery-free active paper printed RFID tag with human-energy scavenger. In *2011 IEEE MTT-S International Microwave Symposium*, pages 1–4, June 2011.

[126] Panasonic. SP-Cap Conductive Polymer Aluminum Capacitors. `https://web.archive.org/web/20180419180443/https://gateway.ipfs.io/ipfs/QmWSVp3VKL6f7oAkcYiY5ZLbaQVAg5ZMvMuYni4NzB54ar/FD_CD_UD_UE_Rev_Mar2015.pdf`, 2018. CD Series.

[127] J. A. Paradiso and M. Feldmeier. A compact, wireless, self-powered pushbutton controller. In *International Conference on Ubiquitous Computing*, pages 299–304. Springer, 2001.

[128] C. Park and P. 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.

[129] C. Park, J. Liu, and P. H. Chou. Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 54. IEEE Press, 2005.

[130] A. N. Parks, A. P. Sample, Y. Zhao, and J. R. Smith. A wireless sensing platform utilizing ambient RF energy. In *Biomedical Wireless Technologies, Networks, and Sensing Systems (BioWireleSS), 2013 IEEE Topical Conference on*, pages 154–156. IEEE, 2013.

[131] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, June 2014.

[132] M. Perez, S. Boisseau, P. Gasnier, J. Willemin, M. Geisler, and J. L. Reboud. A cm scale electret-based electrostatic wind turbine for low-speed energy harvesting applications. *Smart Materials and Structures*, 25(4):045015, 2016.

[133] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 48. IEEE Press, 2005.

[134] Powercast Corporation. P2110B 915MHz RF Powerharvester Receiver. `http://www.powercastco.com/products/powerharvester-receivers/`, 2017.

[135] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. 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.

[136] A. Raha, A. Jaiswal, S. S. Sarwar, H. Jayakumar, V. Raghunathan, and K. Roy. Designing Energy-Efficient Intermittently Powered Systems Using Spin-Hall-Effect-Based Nonvolatile SRAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):294–307, Feb. 2018.

[137] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 36–36. USENIX Association, 2012.

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

[139] D. Rancourt, A. Tabesh, and L. G. Frchette. Evaluation of centimeter-scale micro windmills: aerodynamics and electromagnetic power generation. In *Proc. PowerMEMS*, volume 20079. 2007.

[140] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, page 5. ACM, 2014.

[141] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170, 2012.

[142] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[143] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A Comparison of High-Level Full-System Power Models. *HotPower*, 8:3–3, 2008.

[144] ROHM Semiconductor. Standard CMOS Voltage Detector IC. `https://web.archive.org/web/20160314151909/http://rohmfs.rohm.com/en/products/databook/datasheet/ic/power/voltage_detector/bu48xxg-e.pdf`, 2016.

[145] A. Rudoff. Persistent Memory Programming. *Login: The Usenix Magazine*, 42:34–40, 2015.

[146] N. Sakimura, Y. Tsuji, R. Nebashi, H. Honjo, A. Morioka, K. Ishihara, K. Kinoshita, S. Fukami, S. Miura, N. Kasai, T. Endoh, H. Ohno, T. Hanyu, and T. Sugibayashi. 10.5 A 90nm 20mhz fully nonvolatile microcontroller for standby-power-critical applications. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 184–185, Feb. 2014.

[147] Saleae, Inc. Saleae Logic Analyzer.

[148] A. P. Sample and J. R. Smith. The Wireless Identification and Sensing Platform. In J. R. Smith, editor, *Wirelessly Powered Sensor Networks and Computational RFID*, pages 33–56. Springer New York, New York, NY, 2013. DOI: 10.1007/978-1-4419-6166-2_3.

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

[150] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

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

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

[153] Seiko. Micro Battery Product Catalogue. `https://web.archive.org/web/20170921095837/http://www.sii.co.jp:80/en/me/files/2016/02/English_Micro-Battery-2016.pdf`, 2016. CPX,CPH,XH Series.

[154] F. K. Shaikh and S. Zeadally. Energy harvesting in wireless sensor networks: A comprehensive review. *Renewable and Sustainable Energy Reviews*, 55:1041–1054, Mar. 2016.

[155] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[156] F. Simjee and P. H. Chou. Everlast: long-life, supercapacitor-operated wireless sensor node. In *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, pages 197–202. IEEE, 2006.

[157] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th annual Design Automation Conference*, pages 524–529, 2001.

[158] A. Sinha and A. P. Chandrakasan. JouleTrack-a Web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 220–225, 2001.

[159] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 161–174. ACM, 2007.

[160] ST microelectronics. EnFilm - rechargeable solid state lithium thin film battery. `https://web.archive.org/web/20170808215544/http://www.st.com/content/ccc/resource/technical/document/datasheet/cd/ac/89/0b/b4/8e/43/0b/CD00270103.pdf/files/CD00270103.pdf/jcr:content/translations/en.CD00270103.pdf`, 2018.

[161] S. Sudevalayam and P. Kulkarni. Energy Harvesting Sensor Nodes: Survey and Implications. *IEEE Communications Surveys Tutorials*, 13(3):443–461, 2011.

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

[163] Taiyo Yuden. Multilayer Ceramic Capacitors. `https://web.archive.org/web/20180419173534/https://www.yuden.co.jp/productdata/catalog/mlcc01_e.pdf`, 2018. AMK series.

[164] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith. Battery-Free Cellphone. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(2):1–20, June 2017.

[165] V. Talla, B. Kellogg, B. Ransford, S. Naderiparizi, S. Gollakota, and J. R. Smith. Powering the Next Billion Devices with Wi-Fi. *ArXiv e-prints*, May 2015.

[166] J. Tan, P. Pawełczak, A. Parks, and J. 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.

[167] TDK. Electric Double Layer Capacitor. `https://web.archive.org/web/20180419181115/https://gateway.ipfs.io/ipfs/QmPCJy6nQSPsiyD4m7cimiWj9mdjjqfdeA4utehV39X99H/EDLC252520-351-2F-21.pdf`, 2018.

[168] Texas Instruments. FRAM - New Generation of Non-volatile Memory, 2014.

[169] Texas Instruments. FRAM FAQs, 2014.

[170] Texas Instruments. EnergyTrace. http://www.ti.com/tool/ENERGYTRACE, 2018.

[171] M. Thielen, L. Sigrist, M. Magno, C. Hierold, and L. Benini. Human body heat for powering wearable devices: From thermal energy to application. *Energy Conversion and Management*, 131:44–54, Jan. 2017.

[172] TI Inc. Overview for MSP430FRxx FRAM. http://ti.com/wolverine, 2014. Accessed: 2014-07-28.

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

[174] TI Inc. CC430F613x, CC430F612x, CC430F513x MSP430 SoC With RF Core datasheet (Rev. H). http://www.ti.com/lit/gpn/cc430f5137, 2018.

[175] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. In *Technologies for wireless computing*, pages 139–154. Springer, 1996.

[176] TrisolX. Solar Wings. https://web.archive.org/web/20170406213629/http://www.trisolx.com/wp-content/uploads/2016/07/TrisolX-Solar-Wing-Data-Sheet-160701.pdf, 2018.

[177] J. Van Der Woude and M. 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.

[178] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[179] N. Villar and S. Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, Jan. 2010.

[180] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, Mar. 2011.

[181] R. Vullers, R. van Schaijk, I. Doms, C. Van Hoof, and R. Mertens. Micropower energy harvesting. *Solid-State Electronics*, 53(7):684–693, July 2009.

[182] P. Wagemann, T. Distler, T. Honig, H. Janker, R. Kapitza, and W. Schroder-Preikschat. Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems. pages 105–114. IEEE, July 2015.

[183] J. Wang, Y. Liu, H. Yang, and H. Wang. A compare-and-write ferroelectric nonvolatile flip-flop for energy-harvesting applications. In *The 2010 International Conference on Green Circuits and Systems*, pages 646–650, June 2010.

[184] A. S. Weddell, D. Zhu, G. V. Merrett, S. P. Beeby, and B. M. Al-Hashimi. A practical self-powered sensor system with a tunable vibration energy harvester. 2012.

[185] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[186] A. Wickramasinghe, D. Ranasinghe, and A. Sample. Windware: Supporting ubiquitous computing with passive sensor enabled rfid. In *RFID (IEEE RFID), 2014 IEEE International Conference on*, pages 31–38, April 2014.

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

[188] P. Wgemann, T. Distler, T. Hnig, H. Janker, R. Kapitza, and W. Schrder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 105–114, Washington, DC, USA, 2015. IEEE Computer Society.

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

[190] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, and J. Hu. Avoiding Data Inconsistency in Energy Harvesting Powered Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, 23(3):1–25, Mar. 2018.

[191] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

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

[193] L. Yerva, B. Campbell, A. Bansal, T. Schmid, and P. Dutta. Grafting energy-harvesting leaves onto the sensornet tree. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 197–208. ACM, 2012.

[194] Zac Manchester. KickSat. `http://zacinaction.github.io/kicksat/`, 2015.

[195] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers. Live code update for IoT devices in energy harvesting environments. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, pages 1–6. IEEE, 2016.

[196] H. Zhang, J. Gummeson, B. Ransford, and K. Fu. Moo: A batteryless computational rfid and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011.

[197] H. Zhang, M. Salajegheh, K. Fu, and J. Sorber. Ekho: bridging the gap between simulation and reality in tiny energy-harvesting sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, page 9. ACM, 2011.

[198] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, Dec. 2013.

[199] T. Zhu, Y. Gu, T. He, and Z.-L. Zhang. eShare: a capacitor-driven energy storage and sharing network for long-term operation. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 239–252. ACM, 2010.

[200] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of Energy-Cognizant Scheduling Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1447–1464, July 2013.