A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices

Alexei Colin
Carnegie Mellon University
Pittsburgh, U.S.A.
acolin@andrew.cmu.edu

Emily Ruppel
Carnegie Mellon University
Pittsburgh, U.S.A.
eruppel@andrew.cmu.edu

Brandon Lucia
Carnegie Mellon University
Pittsburgh, U.S.A.
blucia@andrew.cmu.edu

Abstract

Battery-free, energy-harvesting devices operate using energy collected exclusively from their environment. Energy-harvesting devices allow maintenance-free deployment in extreme environments, but requires a power system to provide the right amount of energy when an application needs it. Existing systems must provision energy capacity statically based on an application's peak demand which compromises efficiency and responsiveness when not at peak demand. This work presents Capybara: a co-designed hardware/software power system with dynamically reconfigurable energy storage capacity that meets varied application energy demand.

The Capybara software interface allows programmers to specify the energy *mode* of an application task. Capybara's runtime system reconfigures Capybara's hardware energy capacity to match application demand. Capybara also allows a programmer to write *reactive* application tasks that pre-allocate a *burst* of energy that it can spend in response to an asynchronous (e.g., external) event. We instantiated Capybara's hardware design in two EH devices and implemented three reactive sensing applications using its software interface. Capybara improves event detection accuracy by 2x-4x over statically-provisioned energy capacity, maintains response latency within 1.5x of a continuously-powered baseline, and enables reactive applications that are intractable with existing power systems.

CCS Concepts • **Computer systems organization** → **Embedded hardware**; **Embedded software**;

Keywords Intermittent computing; energy-harvesting power system; energy burst

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173210

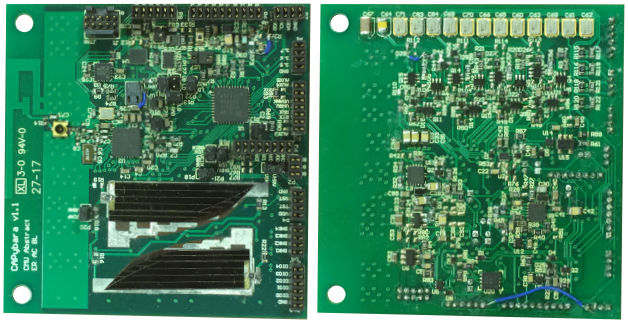

Figure 1. Capybara hardware prototype. The solar panels, microcontroller, radio, and, five sensors are on the front side (left), and the power system with five capacitor banks and four switches is on the back side (right).

ACM Reference Format:
Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173210

1 Introduction

The maturation of energy-harvesting power systems and extremely low-power computing and sensing technology has created a new domain of batteryless devices powered entirely by energy collected from a source in their environment, such as radio waves, light, or vibration. Freedom from a battery or tethered power supply enables developers to deploy applications that require little maintenance, even in harsh, remote environments, like glaciers and in Earth's orbit.

Energy-harvesting devices present a unique challenge by operating only intermittently, as energy is available. Such devices collect energy and store it in a buffer (e.g., a capacitor). When the capacitor reaches a threshold voltage, the device begins to run software, manipulating memory, accessing sensors, and communicating. When energy depletes, the device powers down. After powering down, the device accumulates energy until it can again begin operating.

Computing, sensing, and communication tasks in an application place a spectrum of constraints on the energy-harvesting power system of the device. *Intermittent* tasks

are typically computational tasks with few energy demands. Intermittent tasks, which were studied in prior work [4, 5, 10, 14, 16, 23, 25, 27, 38], can be executed opportunistically and interrupted almost arbitrarily. *Energy-capacity-constrained* tasks require a minimum energy storage capacity in the power system. Capacity-constrained tasks include operations that must execute *atomically* and cannot be interrupted by a power failure, such as sending a radio transmission or collecting correlated sensor data. *Temporally-constrained* tasks require energy to be available on-demand. A task that responds to an external event by transmitting a radio packet is *reactive* and requires energy to transmit immediately at the time of the event.

Capacity- and temporally-constrained tasks in the same application may have requirements of the power system that conflict. A hardware designer may build a power system with a large energy buffer to support the largest capacity-constrained task in the application. However, after depletion, a large energy buffer has a long recharge time. During the recharge period, the device is off and temporally-constrained tasks will not execute reactively. Alternatively, a hardware designer may build a power system with a small energy buffer designed to execute temporally-constrained tasks reactively, avoiding long, inactive recharge periods. However, the small buffer may store insufficient energy for a capacity-constrained task. The application is not fully functional with either power system.

Application developers must co-design application software with power system hardware. Software can control when and in what quantity to accumulate energy only to the extent of configuration supported by the hardware. Inflexible hardware may force all software tasks to use a large "worst-case" energy buffer, violating application requirements. Power systems of state-of-the-art energy-harvesting devices do not support programmatic reconfiguration of energy capacity at runtime. The limited control over charge and discharge timing available to programmers today must be expressed indirectly through control code that puts the device to sleep at key points in code. Such control code expresses task's high-level energy requirements indirectly and imperatively through *ad hoc* device-specific code.

We propose Capybara[1], a high-level abstraction for specifying capacity and temporal task constraints with a co-designed hardware/software system that executes applications according to those constraints. Capybara supports *declarative specification* of tasks' energy requirements and eliminates the need for imperative power system control code that entangles application logic with low level hardware configuration. Capybara allows an application to mix capacity and temporal requirements with a *reconfigurable hardware energy storage mechanism* that an application can reconfigure at run time to support different capacities. We developed two full hardware/software prototypes of Capybara built into custom energy-harvesting platforms, one of which is pictured in Figure 1. We use these prototypes to show that Capybara enables applications to match power system characteristics to task requirements, providing flexibility, efficiency, and reactivity. Given concise declarations of the tasks' energy needs, Capybara ensures that capacity-constrained tasks have sufficient energy to execute and temporally-constrained tasks execute reactively. The contributions of this work are:

- A hardware energy storage mechanism with capacity that is reconfigurable at runtime compatible with different capacitor types and energy harvesters.
- A declarative software interface for specifying task energy requirements.
- A runtime system that reconfigures energy storage to meet task energy requirements.
- Two full-system, solar-harvesting platforms: a versatile sensing platform and a board-scale nano-satellite.
- An evaluation on real hardware showing that reconfigurability improves responsiveness and event detection accuracy.

This paper is organized as follows: Section 2 provides background on energy-harvesting and intermittent software challenges. Section 3 is an overview of Capybara's hardware and software. Section 4 describes Capybara's software interface and runtime. Section 5 describes Capybara's reconfigurable power system hardware. We evaluate Capybara in two real energy-harvesting platforms in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 Intermittent Energy-constrained Systems

Energy-harvesting devices collect energy from the environment, buffer a useful quantity of energy, and execute using the buffered energy. The larger the buffer, the longer a span of uninterrupted operation the device can support, and the longer the time required to recharge the buffer. To not make any assumptions about availability of incoming power, the intermittent execution model allows the processor to be completely off while charging, turn on only once the buffer is full, and execute until the buffer is empty. This model is more challenging but also more practical than a non-intermittent model that assumes that incoming power is always sufficient to keep the processor in a (memory-retaining) sleep state while charging the buffer.

The time the device operates before losing power depends on the energy buffer size, not on the incoming power. A device consumes buffered energy much faster than it charges, because harvested power is much lower than active power consumption. The disparity between charging and consumption has two consequences: charging is negligible during operation and charge times may be orders of magnitude

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

longer than discharge times. During active operation, the energy buffer is the only relevant power source, and its size determines the software tasks that can be executed.

Recent work developed support for *intermittent computing* on an energy-harvesting device [4–6, 10, 14, 23, 25, 38]. The key challenge addressed by this prior work is that buffered energy is available only *intermittently*. When power fails, the device loses volatile state, and retains non-volatile state, compromising forward progress and memory consistency. Programming languages [10, 25], software [16, 23, 32, 38] and hardware approaches [4, 5, 14, 24] addressed these issues, ensuring correctness for intermittent computing workloads.

While prior work focused on the correctness of *computation*, energy-harvesting applications may execute tasks that compute, store data, sense, and communicate, using various peripherals. While computational tasks can often be decomposed into subtasks that execute intermittently, other tasks may require a fixed quantum of energy to complete *atomically* without being interrupted by a power failure. For example, successfully collecting a sample from a sensor may require operating atomically at a low power level for only 8 milliseconds; transmitting a 25 byte Bluetooth packet requires operating atomically with a much higher power level for 35 milliseconds. Today's energy-harvesting devices are equipped with a single energy buffer and to support a given application, the buffer must be provisioned at design time to hold enough energy for the *largest* atomic task.

2.1 Task constraints

This work is based on the observation that tasks in an application are not uniform in their demands of the power system. A fixed power system cannot serve all types of tasks, i.e., intermittent, capacity-constrained, and temporally-constrained, because each type of task performs best with a different energy buffer capacity. Energy storage capacity determines a device's operating time and recharge time. The timing of charge and discharge intervals dictates whether a device meets the demands of an application. Operating time determines whether a capacity-constrained task will *complete without interruption*, because the task will fail given insufficient operating time. Recharge time determines whether a temporally-constrained task will *execute reactively* because the system is off (unresponsive) during recharge.

When an application mixes atomic and reactive tasks, a fixed energy buffer is problematic. Low capacity supports reactive tasks with a short recharge interval, but is insufficient for large atomic tasks. High capacity supports large atomic tasks, but causes long, inactive recharge intervals which compromise reactivity. Usually, with fixed energy capacity, a developer must sacrifice reactivity to ensure there is sufficient energy for large atomic operations.

Figure 2 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

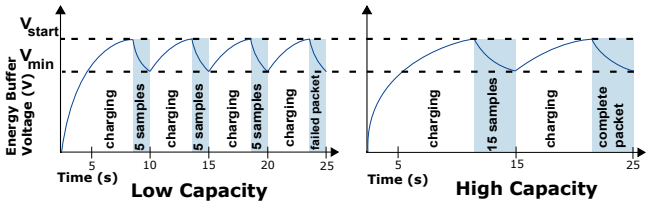

Figure 2. Execution with a fixed-capacity energy buffer. Devices are forced to trade short charge cycles for the ability to complete energy intensive tasks.

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. To successfully meet the demands of such an application, a system requires energy *reconfigurability*, which is central to our main contributions.

2.1.1 Asynchronous tasks and on-demand energy

Some individual tasks have both temporal and capacity constraints and today's devices are especially ill-suited to these tasks. Tasks with both types of constraints are often *asynchronous*, presenting a need to react to an unpredictable event, such as an environmental signal or interaction with a user. To handle unpredictable events, a device's power system 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. Moreover, a capacity-constrained, reactive task incurs a long recharge latency before executing, rendering latency-sensitive applications (e.g., human-computer interfaces), largely inapplicable to existing energy-harvesting devices. A concrete example of such an application samples a sensor, detects a specific event, and sends a radio alert. The radio alert task is both capacity- and temporally-constrained. 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.

2.2 Design space

The power system demands of intermittent, energy-constrained systems define a new hardware/software design space. The dimensions of this space are: required *atomicity* of a task (i.e., minimum operating time), the required *reactivity* of a task (i.e., maximum recharge time), the total energy buffer capacity (in μF), and the board area or volume consumed by energy buffering capacitors. We characterize the space to show that the problem is fundamentally one of hardware/software co-design and that it is essential to avoid design points that violate application requirements.

2.2.1 Atomicity, reactivity, and energy capacity

Device energy capacity determines the longest possible span of operations that can execute atomically, and the minimum time between operations, i.e. maximum reactivity. The minimum interval between operations is also limited by hardware capabilities, e.g. sensor warm-up time, however this interval may be *unnecessarily* inflated at design points where these limits are below the charge time. Figure 3 illustrates the trade-off between provisioning for atomicity and reactivity from a perspective of a task in a multi-task system. The figure shows how either implementability or efficiency of a task may be compromised by another task. In this experiment, we connected a MSP430FR5969 microcontroller to capacitors of different size and type. For each capacitor, we measured the longest span of ALU operations that the device could execute before a power failure, modeling a generic atomic operation consuming a particular amount of energy.

The curve in Figure 3 represents design points that are *optimal*, perfectly matching the span of atomic operations to buffered energy. In configurations to the left of the curve, the task is *infeasible*: the atomicity requirement of the task exceeds the maximum possible with the given energy buffer size. In configurations to the right of the curve, the task is *not reactive* and *inefficient*: the buffer and the time to charge it are larger than needed by this task. In these inefficient configurations, the device executes in unnecessarily long bursts, and spends an unnecessarily long time recharging leaving the device powered off and not reactive. Assuming the atomicity requirement of the task indicated by the dashed line, in Design A the task is infeasible, and in Design B the task is not reactive. However, other tasks might only be feasible in Design B. Unless all tasks are the same, no fixed-capacity design will satisfy the requirements of all tasks.

2.2.2 Atomicity and capacitor volume

After determining the energy buffer capacity required to meet a task's atomicity requirement, the designer must provision a buffer with that capacity. Figure 4 illustrates the non-uniform relationship between capacitor volume and atomicity. In this experiment, the microcontroller was powered by a bank of one or more capacitors of the same type in

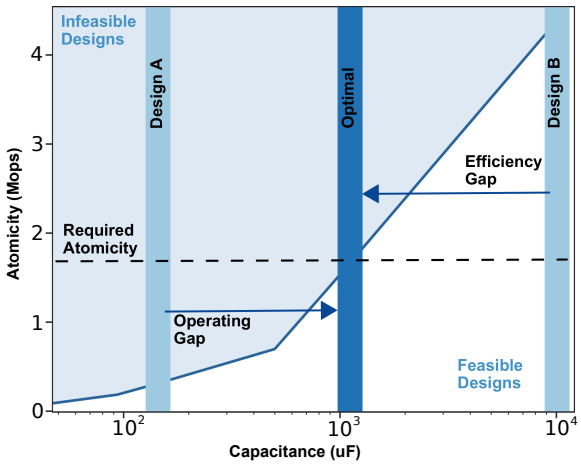

Figure 3. Design space for energy buffer capacity. Systems with insufficient energy storage capacity will not be able to complete workloads with atomicity requirements, but systems with overprovisioned energy storage will spend more time charging than is necessary.

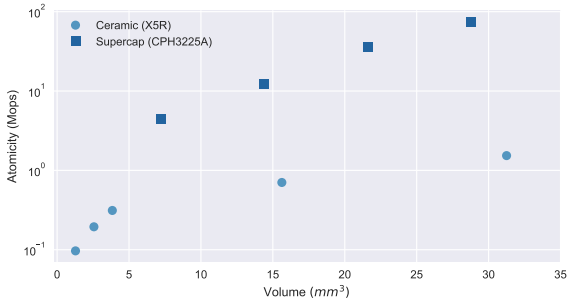

Figure 4. Design space for provisioning a given atomicity requirement with a capacitor by volume and type.

the highest density package connected in parallel. The first observation is that differences between capacitor technologies produce inefficient design points. An equal or larger volume of ceramic capacitors provides less atomicity than a smaller volume of supercapacitors, due to the low density of ceramic capacitors. The second observation is that for the supercapacitor model we evaluated, the atomicity sees a diminishing increase with volume. This effect is due to the high Equivalent Series Resistance (ESR) of this ultra-compact supercapacitor model, which is inversely proportional to the number of capacitors connected in parallel. A high ESR limits the amount of useful energy that can be extracted once stored, and renders the capacitor useless in power systems without the capability to boost voltage, as we explain in Section 5. When provisioning, the designer should avoid inefficient design points and consider the capacitor type along with power system capabilities.

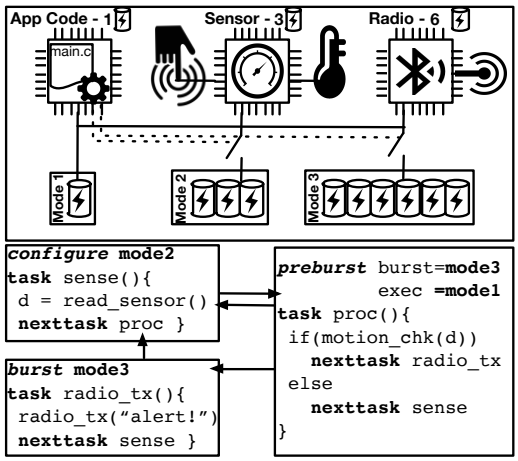

Figure 5. Overview of Capybara. The platform has resources for computation, sensing, and communication, and includes three energy storage configurations with different capacities. An example program has tasks annotated with energy mode requirements.

2.2.3 Versatility

Existing energy-harvesting power systems have limited versatility. Factors limiting their versatility are over-specialization to a particular load (e.g. an MCU or sensor) [33], to an expected input power level [31], or to a particular capacitor ESR and voltage rating [30]. Capybara is inspired by the need for a power system that is reusable across a variety of applications and dynamically reconfigurable to support applications with varied energy demands.

3 Capybara 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 5 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.

Task-based programming model. The application depicted shows software tasks expressed in the style of recent task-based intermittent programming and execution models [10, 25]. 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 4 describes how a programmer conveys task energy requirements through Capybara's programming interface. Section 5 how Capybara's hardware implements a reconfigurable energy reservoir using an array of capacitor banks.

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 4). Capybara's power system (described in Section 5) 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 [13]). 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.

4 Capybara 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 5 discusses how energy modes are implemented in hardware.

4.1 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 capacity constraint, a temporal constraint, or both. Recall from Section 2 that a task with a capacity constraint requires a specific minimum amount of energy to complete without being interrupted by a power failure, a task with a temporal constraint 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 5 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 3.

The programmer can use a `config (mode)` annotation to indicate either a capacity or temporal task constraint. For a capacity 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 a temporal 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 5 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 `mode2` box as a result of the `config (mode2)` 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()`.

4.2 Responsive asynchronous bursts

Capybara allows tasks to have a capacity 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 of the energy 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 5, `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.

4.3 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 5. 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.

5 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 architecture is illustrated in Figure 6(a). The hardware design consists of two parts: (i) the power distribution circuit, and (ii) the capacity reconfiguration circuit.

5.1 Power distribution

Capybara's power distribution circuit accepts energy from the harvester, charges energy buffering capacitors, and generates a usable output voltage to power the load. Our design is versatile, because it can operate with a wide range of input voltage and power, it is compatible with high-ESR capacitors (e.g. small dense super-capacitors), and supports loads with voltage requirements that may exceed the harvester voltage output or the capacitor voltage rating. 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 *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. Capybara's particular 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 reduces charge time by at least an order of magnitude.

The *output booster* allows Capybara to extract more stored energy from the energy buffering capacitors than a direct connection to the load. The booster produces stable output voltage, despite decreasing capacitor voltage until the capacitor is discharged nearly completely (down to about 10% of capacity on our devices). Output boosting is required especially for high-density high-ESR supercapacitors to compensate for the voltage droop induced by the ESR under load. The regulated voltage of the output booster also allows Capybara to power sensors, actuators, and radios with a high minimum operating voltage (e.g. 2.5v gesture sensor or 2.0v for BLE radio).

5.2 Reconfigurable energy storage circuit

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 (1.6v) before any *useable* 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 low atomicity 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,

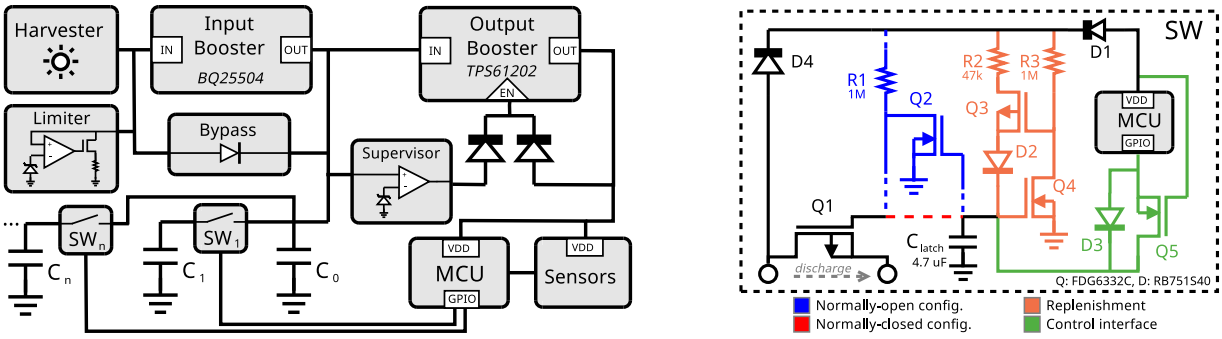

Figure 6. Capybara power system hardware: system architecture and capacitor bank switch replicable module.

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* (bottom-left of Figure 6(a), marked *SW*). 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 3 discusses how to determine an application's energy modes. Figure 6(b) shows the switch circuit that *activates* a bank. The figure includes two design variants, "normally-open (NO)" (blue) in which is open by default and "normally-closed (NC)" (red).

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 switch interrupting a bank's charge path is implemented as a P-channel MOSFET (Q1) in a high-side switch configuration. 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.

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.

Voltage thresholding alternative Reconfigurable energy storage is also possible by configuring the voltage to which capacitors are charged. We studied this design alternative by including in our prototype a non-volatile, variable threshold circuit based on a digital potentiometer and a voltage supervisor. Our study revealed that compared to switched capacitor banks, the threshold circuit occupies twice the area and has 1.5x higher leakage current (according to component specifications). The threshold design also limits device lifetime, because the write endurance of the EEPROM potentiometer is limited. In a switch design wear can be reduced on high-density capacitors by dedicating them to a separate bank that cycles less frequently than the small bank.

6 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. We also perform a case-study of applying Capybara to a low-earth-orbit board-scale satellite [40] to show the versatility of our system.

6.1 Applications and methodology

We implemented three complete applications characteristic of the embedded domain in the Chain programming language [10] and deployed them onto the Capybara board. The applications depend on tasks with distinct atomicity and temporal requirements. 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. We describe each application and the hardware setup used to drive the applications with real environmental input.

6.1.1 Wireless Gesture-Activated Remote Control

We implemented a batteryless, wireless, touch-free gesture-activated remote control (GRC) using the APDS-9960 gesture sensor, a phototransistor and the CC2650 wireless MCU. Each time the MCU turns on, the application samples the phototransistor 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 BLE radio.

We implemented two variants of the GRC application. In GRC-Compact the atomicity requirements of the application are: (1) acquire one sample from the phototransistor, (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 atomicity 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 temporal requirement of the gesture recognition task is to execute immediately after proximity was detected, before the motion finishes. The temporal requirement of the proximity sensing is to minimize inter-sample times to avoid missing proximity events.

For the Fixed-Capacity system, a capacity of 400 uF ceramic + 330 uF tantalum + 67.5 mF EDLC[2] is provisioned to meet the maximum atomicity requirement, i.e. (2). For Capybara, two configurations are provisioned, one per energy mode. In both gesture variants Capybara uses a 400 uF ceramic + 330 uF 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 atomicity 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, as seen in Figure 7. The board is powered using a harvester built from a voltage regulator and an attenuating resistor that supplies at most 10 mW of power.

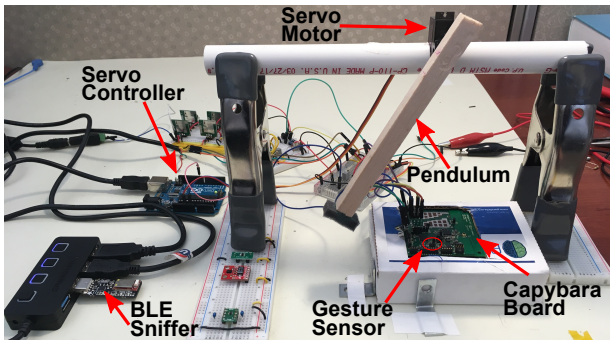

Figure 7. Experimental setup for gesture-activated remote control application.

6.1.2 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 atomicity requirements of the application are: (1) acquire one temperature sample and (2) transmit a 25 byte BLE packet. The temporal requirement of the sampling task is to minimize charging intervals to not miss over- and under-temperature events. The temporal requirement 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 uF ceramic + 1100 uF tantalum + 7.5 mF EDLC capacity.The Capybara systems use one configuration with 300 uF ceramic + 100 uF tantalum to support energy mode (1), and another with 1000 uF tantalum + 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 60W heating element, and a 60W 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, illuminated with a 20W halogen bulb with brightness controlled by

[2]Energy capacity is not fungible due to different equivalent-series-resistance of capacitor types that affects the effective extractable energy.

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.

6.1.3 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 uF ceramic + 330 uF 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.

6.2 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 8 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 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.

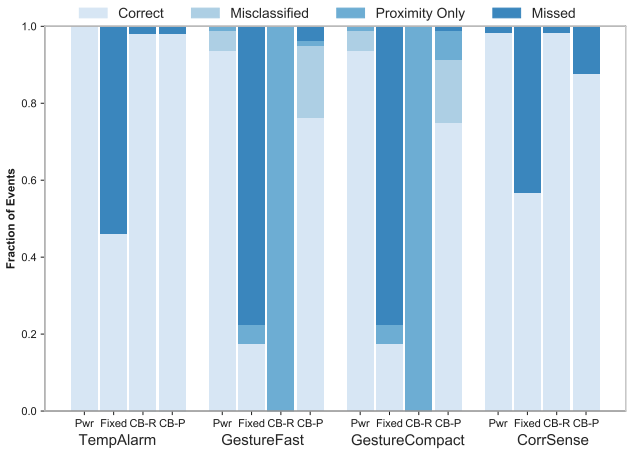

Figure 8. Event detection accuracy.

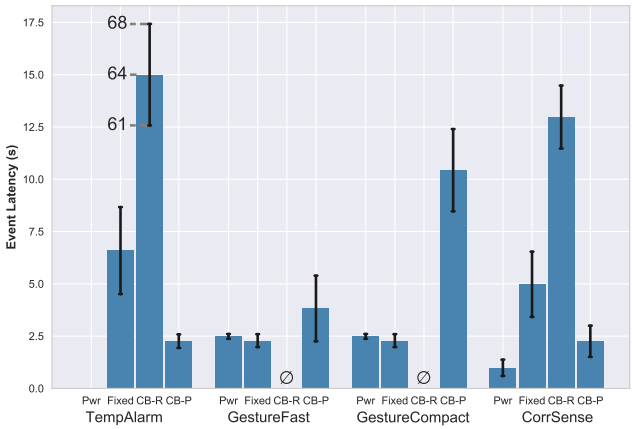

Figure 9. Report latency for detected events.

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 10 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.

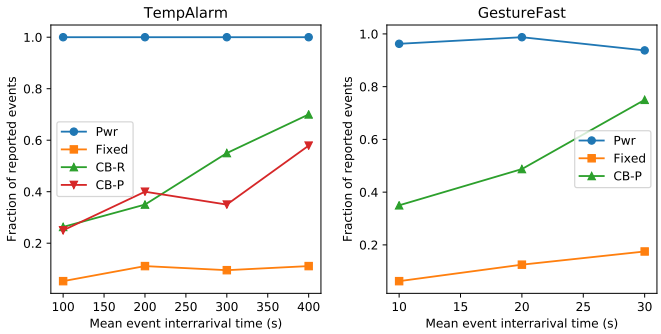

Figure 10. Sensitivity of accuracy to event inter-arrival.

6.3 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 9 shows the latency of each event that was successfully reported in the experiment of Section 6.2. 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 9. As explained in Section 6.2 Capy-R reports no events for GRC.

6.4 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 11 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 6.2.

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

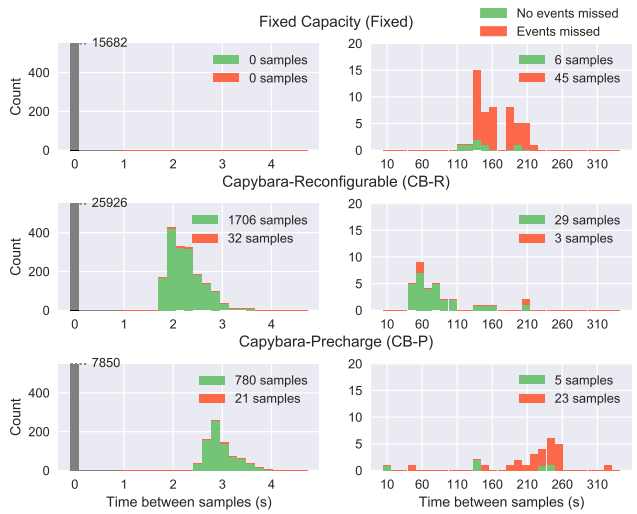

Figure 11. Distribution of times between samples in TempAlarm application. Total counts of *non-back-to-back* samples show that sampling is *denser* with Capybara compared to a fixed capacity.

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 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 10, but is compensated by the order of magnitude improvement in latency in Figure 9.

6.5 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, solar panels occupy 700 mm^2, the Capybara power system 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 hardware design files and source code for our prototype will be released.

6.6 CapySat case study

We specialized Capybara to build a solar-powered board-scale low-earth orbit satellite deployable via a KickSat carrier satellite [40]. Unlike our terrestrial sensors nodes, the satellite board has strict constraints on volume (1.7x1.7x0.15in, including the solar panels) and must withstand temperatures as low as -40C. The volume and temperature constraints severely limit eligible energy-storage technologies, disqualifying all batteries, including thin-film, and many supercapacitors. However, the sensing application, tasked with collecting samples from an on-board magnetometer, accelerometer, and gyroscope, has an extreme atomicity requirement and requires two energy modes. The energy modes are sampling and communication back to Earth. To transmit a 1-byte radio packet to Earth the satellite must keep the radio on for 250 ms while draining 30 mA of current, due to a redundant encoding with a 1064x bit length overhead.

Capybara power system features described in Sections 3-5 are vital for meeting these requirements. For example, without the input and output boosters, energy storable and extractable from a capacitor bank that would fit on the board would be insufficient for the radio transmission. Because the application runs on two MCUs concurrently, each exercising one of the two energy modes, we are able to support energy modes with a simplified version of the capacitor bank switch from Section 5.2. We transform the switch into a diode-based splitter that always connects both banks to the harvester but only one bank to each of the MCUs. Like the general-purpose Capybara, the resulting configuration matches the energy storage to the application demands, but at 20% of the area.

7 Related Work

Energy-harvesting platforms and power systems Recent energy-harvesting research produced platforms, re-usable power modules, and power system designs. Compared to battery-powered motes, e.g. Synergy [1], Amulet [12], Eco [29], devices based on Capybara eliminate the volume and maintenance overhead of batteries through energy-harvesting. Energy-harvesting motes available today harvest light [26, 37, 39], RF [33, 41] and other sources surveyed in [35]. The power systems in these devices are more efficient but less versatile than Capybara, because they specialize for a particular device and target deployment. Capybara places few constraints on platform components and capacitor technology, and provides reconfigurable energy storage.

Federated energy storage [13] dedicates separate capacitors to the MCU and peripherals, charging them in a cascade. Federation, like Capybara, eliminates the need to charge a large capacitor provisioned for the worst-case workload before performing other work. However, federation rigidly allocates energy buffering to a *hardware peripheral*, not a *software task* making it less capable and flexible than Capybara. UFoP focuses on energy partitioning, while Capybara

addresses other challenges: programmable reconfigurability, full capacitor charging with low harvester voltage, and supporting load voltages that exceed capacitor ratings.

Dynamic Energy Burst Scaling (DEBS) [11] programmatically configured energy bursts to minimize the total energy required to execute a complete sequence of tasks. Using reconfiguration, Capybara satisfies atomicity constraints of tasks, as does DEBS, but Capybara also identifies and addresses temporal constraints of tasks, including asynchronous tasks, which motivate a mechanism for pre-charging energy bursts (Section 4.2). The DEBS reconfiguration mechanism is implemented by controlling the V_{top} threshold, whereas Capybara's mechanism controls capacity C (Section 5.2).

Unlike Capybara, Ambimax [28] partitions capacitors across sources not loads, to charge at the maximum power point of each source. Capybara leverages maximum power point tracking in its input booster. eShare [42] allows sensor nodes to share energy via wires to extend network lifetime, and is orthogonal to Capybara.

While Capybara is batteryless, some energy-harvesting systems combine batteries and capacitors. Prometheus [17] increases rechargeable battery lifetime by operating from a supercapacitor when solar energy is available. Unlike Capybara, Prometheus supports only a single, fixed energy buffer. Heliomote [31] is a generic solar-harvester with a permanent battery that informs an application of instantaneous battery and panel voltage. ZebraNet collars [18] were one of the earliest devices to use a solar harvester to charge a battery, and supported peripherals with voltage requirements above battery voltage through voltage boosters, as does Capybara. Software-defined batteries [2] allow the OS to control charge flowing into and out of each battery in an array and to change the power source properties. Batteries avoid the challenges of intermittent execution faced by Capybara, but constrain device size, temperature range, and lifetime.

System support for intermittent computing Software intermittent programming and execution models are complimentary to the support for energy modes provided by Capybara. Capybara's software interface complements task-based systems [10, 23, 25], allowing the programmer to express atomic operations as tasks and annotate tasks with energy requirements. Dynamic checkpointing approaches are less amenable to use with Capybara because checkpoints occur arbitrarily, on energy changes (Hibernus [4, 5], Idetic [27], QuickRecall [16]), at backedges [32], idempotent regions [38], or when detected by custom hardware [14].

Programming language and system support for energy management Energy-aware programming systems allow energy management. However, it is difficult using existing languages to express burst and temporal energy requirements, which is central to Capybara's design. In Eon [36], task are associated with abstract energy states and executed when the system is in the corresponding state. Unlike Eon, Capybara actively changes energy modes by reconfiguring energy capacity, rather than adapting to device changes. Energy Types [9] attribute energy to application phases via a type system. ENT [7] introduces dynamic types that resolve to cause different behavior based on device energy state. Type-systems use energy modes to prevent high-energy code from running while the device is in a low-energy state. In contrast, Capybara uses modes for versatile reconfigurability, reactiveness, and efficiency.

The LAB abstraction [20] 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. By forgoing such high-level application-specific abstractions in its software interface, Capybara supports low-level general-purpose embedded programming. 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 [3, 15, 19, 34]. These systems rely on a mechanism for estimating available energy at runtime, which is an open problem for an energy-harvester that will likely require additional hardware and software. OS-level power management that keeps unused resources in sleep states [22] is complimentary to Capybara. Methods for estimating worst-case energy consumption of software tasks [8, 21] apply for provisioning capacity for Capybara banks given an application.

8 Conclusion

Energy-harvesting platforms free applications from batteries, which are large, heavy, and fragile. To handle environmental triggers responsively, application tasks place capacity and temporal constraints on a device's power system. 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, a hardware mechanism for reconfiguring energy storage capacity and pre-charging capacitors for on-demand energy bursts, and a runtime system that supports reconfiguration. Our evaluation of Capybara in a solar energy harvesting device showed that reconfigurability improves application responsiveness and event detection accuracy. Future work should automate energy capacity estimation for application tasks and find an allocation of capacitors to banks for a set of task energy requirements.

9 Acknowledgments

Thanks to the anonymous ASPLOS 2018 reviewers for their valuable feedback. This work was funded by gifts from Disney Research and Google, by NSF grant CNS-1526342 and NSF CAREER Award CCF-1751029.

References

[1] Michael P. Andersen, Gabe Fierro, and David 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.

[2] Anirudh Badam, Evangelia Skiani, Ranveer Chandra, Jon Dutra, Anthony Ferrese, Steve Hodges, Pan Hu, Julia Meinershagen, Thomas Moscibroda, and Bodhi Priyantha. Software defined batteries. pages 215–229. ACM Press, 2015.

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

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

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

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

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

[8] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Determining Application-specific Peak Power and Energy Requirements for Ultra-low Power Processors. pages 3–16. ACM Press, 2017.

[9] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

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

[11] Andres Gomez, Lukas Sigrist, Thomas Schalch, Luca Benini, and Lothar Thiele. Efficient, long-term logging of rich data sensors using transient sensor nodes. *ACM Trans. Embed. Comput. Syst.*, 17(1):4:1–4:23, September 2017.

[12] Josiah Hester, Sarah Lord, Ryan Halter, David Kotz, Jacob Sorber, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, and Kevin Freeman. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. pages 216–229. ACM Press, 2016.

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

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

[15] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

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

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

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

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

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

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

[22] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and others. TinyOS: An operating system for sensor networks. *Ambient intelligence*, 35:115–148, 2005.

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

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

[25] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2017.

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

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

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

[29] Chulsung Park, Jinfeng Liu, and Pai 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.

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

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

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

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

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

[35] Faisal Karim Shaikh and Sherali Zeadally. Energy harvesting in wireless sensor networks: A comprehensive review. *Renewable and Sustainable Energy Reviews*, 55:1041–1054, March 2016.

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

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

[38] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 17, 2016.

[39] Lohit Yerva, Brad Campbell, Apoorva Bansal, Thomas Schmid, and Prabal 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.

[40] Zac Manchester. KickSat. http://zacinaction.github.io/kicksat/, 2015.

[41] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. Moo: A batteryless computational rfid and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011.

[42] Ting Zhu, Yu Gu, Tian He, and Zhi-Li 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.