# Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution

Kiwan Maeng
kmaeng@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Brandon Lucia
blucia@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

## Abstract

Batteryless energy-harvesting devices eliminate the need in batteries for deployed sensor systems, enabling longer lifetime and easier maintenance. However, such devices cannot support an event-driven execution model (e.g., periodic or reactive execution), restricting the use cases and hampering real-world deployment. Without knowing exactly how much energy can be harvested in the future, robustly scheduling periodic and reactive workloads is challenging.

We introduce CatNap, an event-driven energy-harvesting system with a new programming model that asks the programmer to express a subset of the code that is time-critical. CatNap isolates and reserves energy for the time-critical code, reliably executing it on schedule while deferring execution of the rest of the code. CatNap degrades execution quality when a decrease in the incoming power renders it impossible to maintain its schedule.

Our evaluation on a real energy-harvesting setup shows that CatNap works well with end-to-end, real-world deployment settings. CatNap reliably runs periodic events when a prior system misses the deadline by 7.3× and supports reactive applications with a 100% success rate when a prior work shows less than a 2% success rate.

***CCS Concepts:*** • **Computer systems organization** → **Embedded systems**; *Sensor networks.*

***Keywords:*** intermittent computing, energy-harvesting

**ACM Reference Format:**
Kiwan Maeng and Brandon Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3385998


*PLDI '20, June 15–20, 2020, London, UK*

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385998

## 1 Introduction

Ultra-low-power processors and energy harvesters enable batteryless devices that operate using energy collected from their environment. These devices harvest, e.g., radio waves [17, 57] or solar energy [17, 25], avoiding costs associated with batteries, such as thermal limits, short lifetimes, and replacement costs. Batteryless devices are appealing for applications such as long-term environmental and infrastructure monitoring, sensing in adversarial environments, and chip-scale satellites [17, 25, 40, 57]. Harvestable power sources are weak and unreliable, causing an energy-harvesting device to operate *intermittently* when energy is available. A device charges an energy buffer (capacitor) while inactive. After accumulating sufficient energy, the device performs a burst of computation, depleting the accumulated energy [17, 24, 25, 57]. Time spent charging is typically longer than time spent computing and varies with the environment.

Prior work enables safe, efficient software on batteryless devices [6, 8, 10, 15, 34, 45, 46, 51, 64] by saving state across failures. However, these systems lack support for timed events and interrupts, failing to realize *event-driven* execution as in continuously-powered systems [18, 19, 37, 38].

***Importance of an event-driven execution.*** Many modern embedded systems are event-driven, responding to events, such as timer interrupts. Event-driven systems exist to support *periodic* and *reactive* execution [18, 19, 37, 38]. *Periodic execution* allows running a workload with a particular frequency (e.g., sampling temperature at 1 Hz). *Reactive execution* allows a timely response to a signal of interest (e.g., on a smoke detector interrupt). Periodic and reactive execution are widespread and important embedded operating modes.

Periodic and reactive execution on an energy-harvesting device is a challenge. Periodic execution is challenging because the time to harvest sufficient energy for a workload depends on incoming power. If recharge time exceeds the event's period, periodic execution is not viable. Figure 1 (top) shows a failure to run periodically. Reactive execution is challenging because the device may be powered-off or have insufficient energy for an asynchronous event [9, 55, 65]. Figure 2 (top) shows a failure to run reactively.

***Shortcomings of prior approaches.*** Prior approaches [12, 13, 22, 39, 48] schedule tasks and recharges like a real-time operating system (RTOS) on an energy-harvesting device. These schedulers fail in real-world scenarios where incoming

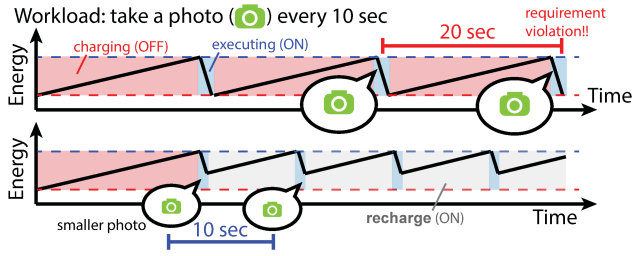



**Figure 1.** Periodic execution. Attempts to periodically take a photo (shaded blue) every 10 seconds is shown. Periodic execution fails with low power (top) because the charging time (shaded red) necessary to take a photo exceeds the desired period of 10 seconds. With degradation (bottom), the periodic requirement is met by the system deciding to start recharging early (shaded gray) after taking a smaller photo.

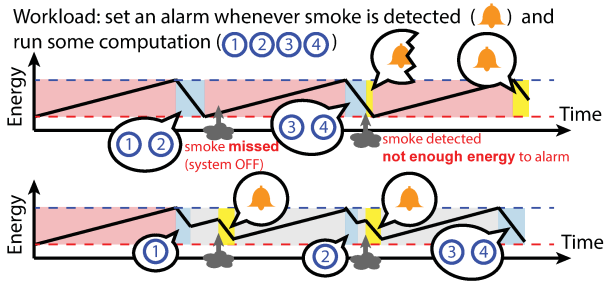



**Figure 2.** Reactive execution. Attempts to reactively set an alarm (shaded yellow) when smoke is detected (gray smoke icon) is shown. Reactive execution can fail (top) if the system is off recharging (shaded red) or have insufficient energy to react when the smoke is detected. With proper recharge scheduling (bottom), the system can postpone less important computations (shaded blue) and recharge early (shaded gray), having enough energy to respond reactively.

power cannot be precisely predicted (Section 2.3). While an RTOS typically schedules based on worst-case execution time, which is predictable, an energy-harvesting device's *charge time* varies with input power and is hard to estimate. The result of mispredicting charge time and overusing energy is catastrophic: misprediction by the lowest-priority task can lead to the *entire system powering off*, halting higher-priority tasks. Avoiding this complexity, most intermittent systems rely on a simplistic execution model, blindly running a workload whenever energy is available [10, 28, 46, 47, 64].

***Our contribution*** To enable robust event-driven execution on an energy-harvesting device, we propose CatNap, a runtime system and programming model that isolates *time-critical events* from *time-insensitive tasks*. The programmer writes a collection of events, containing time-critical operations, and tasks, containing code that need not run immediately. With the event and task definitions, CatNap schedules *only events* like an RTOS scheduler, executing tasks using spare energy. The scheduler also schedules *recharges* to collect energy to execute events. By isolating energy for events and tasks, CatNap tolerates *short-term power fluctuations*, when harvested energy is less than expected. CatNap is simpler and more robust than systems that enforce a global schedule [12, 13, 22, 39, 48], because CatNap schedules events and recharges only, occasionally discarding tasks.

When power decreases for a longer period, CatNap ensures it can collect sufficient energy to schedule events by *degrading* event quality. CatNap detects such a *long-term power fluctuation* by monitoring incoming power directly and estimating the schedule's *feasibility*. If a schedule becomes infeasible because there is no way to collect enough energy to run the events, CatNap falls back to a degraded version of the application that either runs different code or the same code less frequently. Figure 1 (bottom) shows how degradation enables periodic execution, and Figure 2 (bottom) shows how scheduling enables reactive execution.

We implemented CatNap, including its new programming model, scheduler, feasibility test, and quality degradation facility. We evaluated CatNap against state-of-the-art event-driven intermittent systems [48, 65] and show that CatNap has high performance and is correct when prior systems fail.

The main contributions of this paper are:

- A new programming model that allows the user to expose the time-critical parts of the code.
- A scheduler that schedules the time-critical events using a provably correct scheduling policy, while isolating their energy from the time-insensitive tasks for robustness.
- A QoS degradation facility that adjusts the quality to save energy, by directly monitoring input power.
- An evaluation showing that CatNap enables event-driven execution even with low input power. CatNap meets timing requirements while prior state-of-the-art misses its deadlines by 7.3×. CatNap is reactive, capturing 100% of aperiodic events while prior work misses 98% of them.

## 2 Background and Related Work

Prior work enabled computing on embedded and intermittent devices but did not directly target intermittent event-driven execution or incompletely realized it. We discuss several prior systems, including ones that attempt intermittent event-driven execution.

### 2.1 Requirements for an Event-Driven Execution

Intermittent event-driven execution has three *requirements*, which are executing across power failures (**R1**) and supporting timer (**R2**) and hardware interrupts (**R3**). Fulfilling these requirements is, however, insufficient for event-driven execution, additionally requiring three *correctness criteria*. Correctness requires that the system be able to schedule recharges and code properly (**C1**), to maintain a valid schedule even

when incoming power has short-term fluctuations (**C2**) or long-term fluctuations (**C3**). Robustness against short-term power fluctuation is crucial in the real-world: an RF power source configured to emit 15dBm has 10–12% fluctuation [33]. Adapting to a longer-term power variation is also necessary as, e.g., solar-harvester power varies with cloud cover (e.g., Figure 1). Table 1 compares prior work, showing that only CatNap meets all requirements.

**Table 1.** Comparison to prior work. O indicates successfully meeting, X indicates not meeting, and △ indicates partially meeting the requirements. - indicates the comparison is not applicable. (*except InK, which has its own row).

| category | R1 | R2 | R3 | C1 | C2 | C3 |
|---|---|---|---|---|---|---|
| **trad OS** | X | O | O | - | - | - |
| **NI** | X | O | O | - | - | - |
| **intermittent** | O | X | X | - | - | - |
| **dynamic** | O | △ | X | - | - | - |
| **pers clk*** | O | O | X | - | - | - |
| **interrupt*** | O | X | O | - | - | - |
| **InK [65]** | O | O | O | △ | X | X |
| **M-RTOS** | O | O | O | O | X | △ |
| **CatNap** | **O** | **O** | **O** | **O** | **O** | **O** |

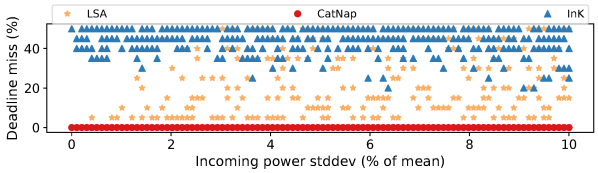
## 2.2 Peripherally-Related Systems

Many prior systems target an orthogonal, though related goal and do not meet the fundamental requirements of R1–3. *Traditional low-power operating systems (trad OS)* [18, 19, 37, 38] do not consider recharges or handle power failures (**R1**). *Non-intermittent energy-harvesting systems (NI)* [20, 32, 61] rely on a large energy buffer that mostly powers the system continuously (e.g., a battery). They do not precisely control recharges or handle intermittent power failures (**R1**). *Simple intermittent systems (intermittent)* [6, 8, 10, 15, 16, 28, 34, 44–47, 51, 64] execute whenever energy is available and do not support timer-based or external interrupts [9, 26, 65] (**R2–3**). *Dynamic energy storage systems (dynamic)* [11, 17] change the size of the energy buffer dynamically to vary the execution interval, which gives only coarse control (**R2**). *Persistent clock systems (pers clk)* [26, 27, 65] propose a specialized clock that runs even when the system is powered off. However, except for InK [65] which we discuss separately below, they do not allow interrupt-based execution (**R3**). *Interrupt-enabled intermittent systems (interrupt)* [9, 55, 65] do not support periodic execution (**R2**), again except for InK [65].

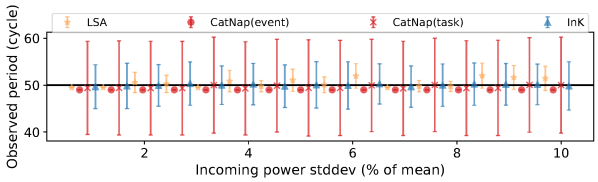
Because the above systems do not attempt to achieve intermittent event-driven execution, we concentrate our comparison against other systems that is more relevant to tackling event-driven execution directly (Section 2.3).

## 2.3 Prior Event-Driven Energy-Harvesting Systems

InK [65] attempts to achieve periodic and reactive execution by adopting a persistent clock and enabling interrupts on an energy-harvesting device. InK *greedily* runs tasks whenever energy becomes available, without ever saving energy for the future. Such greedy scheduling can result in a timing requirement violation as seen in Figure 2 (**C1**). InK can only detect a deadline violation without the capability of preventing it. It is the programmer's responsibility to ensure that there would be no deadline violation by statically selecting system parameters using an oracular knowledge of the energy the system will harvest on deployment. Whenever the incoming energy deviates from the provisioned amount, the system fails in an undefined manner (**C2–3**).




**(a)** Deadlines missed with short-term power fluctuation (C2). The figure shows deadlines missed as power fluctuation increases for LSA, InK, and CatNap.




**(b)** Execution periodicity with short-term power fluctuation (C2). The figure shows the observed execution frequency as power fluctuation increases. For CatNap, we separately show the period for events and for tasks. An interval over the horizontal line is a missed deadline, except for time-insensitive tasks in CatNap which have no deadline.

**Figure 3.** The impact of a fluctuating power source on event-driven energy-harvesting systems.

Others, e.g., LSA [48], modify existing RTOS scheduling algorithms to schedule *both code execution and recharges* [12, 13, 22, 39, 48]. Although they work with an ideal, non-fluctuating power source, RTOS-based systems start to fail when the incoming power fluctuates (**C2**). These systems also do not adapt to long-term power variation; however, some systems (e.g., LSA [48]) offer a feasibility test similar to CatNap (Section 3.2), making it possible to adapt to power variation, *which they currently do not* (**C3**).

We further demonstrate that prior systems show unreliable behaviors when the incoming power fluctuates. We built a simulator that models an energy-harvesting environment, where the harvested power is modeled as a Gaussian distribution instead of a constant value. We varied the standard deviation of the Gaussian up to 10% of the mean and simulated the execution of three systems, LSA [48], InK [65], and CatNap. The systems ran two different workloads, each consisting of 20% time-sensitive activities (e.g., data collection) and 80% time-insensitive activities (e.g., computation).

To program in InK or LSA, it is natural to make the two workloads into two *RTOS tasks*[1]. We assumed each RTOS task was properly sized so as not to experience any deadline violations unless there are power fluctuations.

Figure 3a shows that prior systems miss deadlines when the incoming power fluctuates, even when provisioned with oracular data. LSA started to miss 10% of the deadlines when 1% standard deviation was introduced; it missed up to 50% of the deadlines when the standard deviation of the input power was 3.6%–10%. InK missed 35%–50% of its deadline as soon as any fluctuation was introduced.

LSA and InK assume by design that an expected amount of energy will always be harvested. If the harvested energy is less than what was expected, they end up draining the energy buffer more than planned, leaving insufficient energy for subsequent tasks. The draining of energy can harm other tasks because the energy use of tasks is not isolated from each other; *if the lowest-priority task consumes excessive energy, the entire system shuts down.*

CatNap divides the program into time-critical events and time-insensitive tasks. From our simulation, CatNap never violated the event deadline. Figure 3b further illustrates why CatNap meets the deadline while others do not. The figure plots the observed frequency of code execution (RTOS tasks for LSA and InK, events and tasks for CatNap) on different power fluctuations. We only show the subset of the data points for clarity. LSA and InK have an *average* period that meets the deadline (indicated by the horizontal line); however, they incur high variance in RTOS tasks' execution time. CatNap also has a high variance on code execution time, but crucially, the variance only affects time-insensitive tasks, not time-critical events. CatNap isolates event energy from task energy and prioritizes event execution and the *corresponding recharges* over tasks (as Section 3 explains in detail). CatNap's events always meet the deadline even with a very high standard deviation of incoming power, e.g., up to 100% variation. Unless the mean of the incoming power changes, CatNap accumulates enough energy to run events within their periods, even if power variation is high. CatNap does not run a task until it reserves enough energy for events, ensuring events always run reliably at an occasional cost in the variation of task execution time.

As Figure 3b shows, the downside of CatNap is the risk of unpredictable execution time for time-insensitive tasks, which is the price paid for reliable event-driven execution. Despite the unpredictable task execution time, CatNap practically has higher performance than prior work [65] (Section 7), thanks to several favorable design choices discussed in Section 5.

To summarize, all prior work either does not support intermittent event-driven execution directly (**R1–3**) or was able to partially support it only when the incoming energy does not fluctuate (**C1–3**). CatNap is the first energy-harvesting system that can robustly support event-driven execution even with short- and long-term energy fluctuations.

[1]In RTOS, "tasks" are code regions with a deadline

## 3 The Recharge Scheduler

CatNap's scheduler (1) isolates the energy for time-critical events from time-insensitive tasks, (2) schedules events and the necessary recharges, (3) runs tasks with the remaining energy, and (4) estimates the feasibility of events online.

### 3.1 Events and Tasks

CatNap's design stems from the observation that (1) scheduling all the workloads perfectly is in general impossible with non-ideal incoming power, and (2) many parts of an application do not have a strict scheduling requirement. Consider an application that samples audio and computes its fast Fourier transform (FFT). Audio samples must be collected at a precise frequency, while the FFT computation can be delayed. From this observation, CatNap requires the programmer to separate the time-critical code (i.e., events) from the time-insensitive code (i.e., tasks) explicitly. The two can communicate data or trigger each other. CatNap schedules *events* and the necessary *recharges* using a provably-correct policy similar to that of an RTOS. CatNap runs *tasks* using the remaining energy. CatNap's programming model is similar to TinyOS [37], which uses a two-level priority of events and tasks for reducing overheads.

An *event* is a short code region that must execute responsively, at a specified time. CatNap executes events atomically, without interruption by a recharge or a power failure. Events are a good fit for I/O and peripheral manipulations because they are responsive and atomic. An event may be periodic or posted by an interrupt handler, another event, or a task. As in TinyOS [37], events do not preempt each other and must be short to avoid impeding responsiveness.

A *task* is a long-running computation that need not be responsive, is preemptable, and can run intermittently. CatNap disallows the programmer from specifying any timing requirements (e.g., deadline) for tasks. CatNap exploits the interruptability of tasks, deferring them when events or recharges must run. Tasks run with spare time and energy. A task can be posted by an event or another task. Tasks do not preempt each other. Like TinyOS [37], CatNap does not specify the execution order or a deadline of posted tasks; a posted task finishes *eventually*.

CatNap also schedules *recharges*, inactive periods used to collect energy. A recharge's priority is higher than tasks and lower than events. CatNap schedules recharges to ensure events are timely and avoid power failures.

### 3.2 Operation of the Scheduler

CatNap adopts an RTOS-style scheduling policy for executing events. We first explain how CatNap schedules events. Then, we explain how tasks run without interfering with the events. Finally, we explain how CatNap estimates the feasibility of the event schedule.

***Scheduler Overview*** To enable *periodic* and *reactive* execution, the system must always have enough energy to execute an event immediately as it arrives. CatNap logically (but not physically) divides its energy buffer into an *event bucket* and a *task bucket*, which stores the energy to be used by the events and tasks, respectively. The event bucket is sized to be bigger than the sum of the worst-case energy consumption of all events.

CatNap tries to always keep the event bucket full to process events even if they arrive all at once. When the event bucket is not full, CatNap refills it by scheduling recharges and (logically) transferring energy from the task bucket.

When an event arrives, CatNap runs the event using the energy in the event bucket. When the event bucket is full and there is no event to run, CatNap runs a task with the excess energy; however, CatNap never allows tasks to use energy from the event bucket. When there are no tasks to execute, CatNap fills up the task bucket for future task executions. Task execution can always be preempted by an event execution or a recharge to fill the event bucket.

CatNap physically uses a single energy buffer, logically partitioned for events and tasks. CatNap monitors the energy in its energy buffer ($e_c$) and compares it with $e_{event}$, the size of the event bucket. If $e_c \leq e_{event}$, the energy is in the event bucket. If $e_c > e_{event}$, the event bucket is full and the task bucket holds $e_c - e_{event}$ energy. Using a single buffer has several advantages over physically-separate buffers: changing either bucket size is simple (e.g., via a programmable comparator IC) and energy transfer between buckets is trivial.

Algorithm 1 summarizes the scheduler's operation. CatNap's scheduler can be in one of three states: running events, running tasks, or recharging. CatNap defaults to recharging when an event or task finishes (Line 1-2). CatNap never interrupts an event execution until it finishes (Line 3). CatNap always runs events with the highest priority (Line 4) and only runs tasks when the event bucket is full (Line 6, 8).

**Algorithm 1** CatNap charge scheduler.

```
1: if state ∈ Events ∪ Tasks then
2:     if Finished then state ← Recharge
3: if state ∉ Events then
4:     if ReadyEvents ≠ ∅ then state ← ϵ ∈ ReadyEvents
5:     else
6:         if e_c ≤ e_event then state ← Recharge
7:         else
8:             if ReadyTasks ≠ ∅ then state ← τ ∈ ReadyTasks
9:             else state ← Recharge
```

***Energy Isolation*** CatNap isolates energy in the event bucket from tasks, allowing reactive event execution despite power fluctuations. Short-term fluctuations affect only tasks because events can rely on the energy reserved in the event bucket. When an input power varies over a longer period, CatNap's feasibility test and degradation generate a degraded set of events that runs even with the lower input power.

***Feasibility Test and Degradation*** When the incoming power decreases, CatNap might not be able to keep the event bucket full. We say a set of events is *infeasible* and CatNap degrades the quality of the events to use less energy. To degrade, CatNap increases the events' period or runs a programmer-provided approximate version of the code. Degrading events yields an alternative, feasible set of events.

CatNap uses a *feasibility test* to check the feasibility of events at a given input power, similarly to how an RTOS would check the schedulability of RTOS tasks. The feasibility test assumes that an event execution time itself is negligibly short compared to charge time, which is true in most energy-harvesting scenarios [17]. Assume an event $\epsilon_i$ has a period $t_i$ and a worst-case energy consumption $e_i$. With an incoming power $P$, the time to charge up $e_i$ is $c_i = \frac{e_i}{P}$. CatNap's feasibility test states that:

**Theorem 1.** *Events* $\{\epsilon_0, ..., \epsilon_n\}$ *are* feasible *if* $\sum_{i=0}^{n} \frac{c_i}{t_i} \leq 1$.

We briefly sketch an informal proof and provide a formal proof as an Appendix (Section **??**). If the energy used by any event is replenished before the next instance of the same event, the system is feasible. In other words, if we define $d_i$ as the deadline to recharge after the event $\epsilon_i$, never violating a deadline of $d_i = t_i$ guarantees feasibility. This deadline formulation allows casting our feasibility test as a typical RTOS task (code regions with a deadline) schedulability test. Prior RTOS literature proved an RTOS task $i$ with an inter-arrival period $t_{RTOS,i}$, worst-case execution time $c_{RTOS,i}$ and deadline $d_{RTOS,i} = t_{RTOS,i}$ is schedulable if and only if $\sum_{i=0}^{n} \frac{c_{RTOS,i}}{t_{RTOS,i}} \leq 1$ [67]. $\sum_{i=0}^{n} \frac{c_{RTOS,i}}{t_{RTOS,i}}$ defines *utilization* and represents the fraction of time spent running RTOS tasks [67]. The schedulability result implies that recharges in our system are schedulable without violating their deadlines (i.e., the system is feasible) if and only if $\sum_{i=0}^{n} \frac{c_i}{t_i} \leq 1$. Following the RTOS's convention, we refer to $\sum_{i=0}^{n} \frac{c_i}{t_i}$ as *utilization* or $U$. An aperiodic event can be treated as periodic, using its minimum expected arrival interval as its period $t_i$, which is a technique widely accepted in prior work [67]. Section 4.3 describes aperiodic events in detail. When CatNap's feasibility test fails, CatNap invokes its degradation logic (Section 4.5).

## 4 The CatNap System

CatNap consists of a programming model with events and tasks, a charge scheduler, a compiler, and a runtime system. CatNap's runtime system includes the facility for evaluating feasibility online based on measurements. CatNap's programming model allows the programmer to define how to degrade

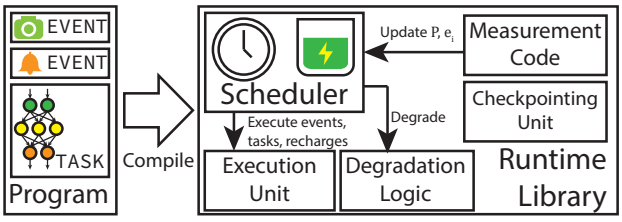



**Figure 4.** Overview of CatNap.

quality to use less energy. CatNap preserves memory state if power fails. Figure 4 summarizes CatNap.

### 4.1 The Assumptions of CatNap

CatNap assumes a single user running a single application with possibly multiple concurrent computations. CatNap assumes a simple, single-core microprocessor with a byte-addressable non-volatile memory and a capability to monitor the buffered energy. The assumptions of CatNap align with most of the prior work [8, 11, 17, 24, 25, 34, 47, 57].

### 4.2 The CatNap Programming Model and Compiler

CatNap asks the programmer to express a program using events and tasks and express guidelines regarding degradation. Figure 5 shows a simplified example of a CatNap program. The programmer defines events and tasks (Line 3-5), the period and the parameter (optional) of events (Line 1-2), and the code for tasks and events (Lines 7-17). The programmer can optionally express how to degrade quality e.g., by increasing the period (Line 1) or the parameter (Line 2). The programmer can also guide the system to use alternative code on degradation by providing more than one function inside the `func={}` argument (not shown). Tasks and events have a structured mechanism for communication, called a channel (Line 6). CatNap's compiler generates an executable instrumented to run tasks and events based on their specifications. Within events or tasks, CatNap can use common C constructs without restrictions, including dynamic memory allocation, recursive calls, and arbitrary pointers, unlike prior work restricting the use of such capabilities [41, 45, 46].

In many cases, programming in CatNap has a reasonable overhead. Dividing events and tasks is mostly straightforward: periodic and reactive I/O operations are good candidates to be an event and compute-intensive code becomes tasks. Many I/O operations have configurable parameters, e.g., resolution of a camera or a transmission power of a radio, that can be directly used for degradation. A CatNap programmer can also leverage approximate computing [3–5, 30, 36, 56, 59, 61] technologies for code degradation. Although CatNap gives programmers the freedom to guide the degradation in a fine-grained manner, programmers can rely on the default parameters unless necessary. In our evaluations, we only specified the core requirements of the applications (in Section 6) and left everything else at its default.

Period, event, task, channel declaration

```
1.PERIOD(name="T",min=1,max=10);
2.PARAM(name="P",min=10,max=100);
3.EVENT(name="capture",period="T"
        type="periodic",
        func={f_capture});
4.TASK(name="compute",
       func={f_compute});
5.EVENT(name="tx",period="T",
        type="aperiodic",
        param="P",func={f_tx});
6.CHANNEL(name="chan",
          src="capture",
          dst="compute",
          struct=img_t);
```

Event, task definition

```
7.void f_capture() {
8. img_t img=getImg();
9. chan.push(img);
10.POST(compute);}
```

```
11.void f_compute() {
12. img_t img
      =chan.pop();
13. if (is_dog(img))
14.  POST(tx);}
```

```
15.void f_tx() {
16. setTxPower(param);
17. send("Dog!");}
```

**Figure 5.** Example CatNap code. The code for an application that captures an image periodically (event `capture`), predicts a presence of a dog (task `compute`), and sends an alarm (event `tx`). Event period is between 1 second to 10 seconds (period `T`) and the system controls the transmission power between 10 to 100 (param `P`, which `f_tx` is accessing via the keyword `param`). A channel holding an image data is defined between `capture` and `compute` (channel `chan`).

### 4.3 The Scheduler

CatNap's scheduler is the core of the runtime system. The scheduler runs a loop dispatching events, tasks, and recharges, as well as handling aperiodic events (i.e., interrupts). CatNap periodically estimates feasibility by computing utilization. If utilization is over a threshold, CatNap deems the events infeasible and invokes degradation logic.

### 4.4 Tasks, Events, and Channels

Events and tasks contain application code. They can exchange values using a First-In-First-Out (FIFO) channel.

***Tasks*** A task is a potentially long-running region of code without a deadline that can be posted to the scheduler to execute. A task does not run immediately after being posted. Instead, CatNap has a FIFO *task queue* and runs one at a time. The task queue has a fixed size and an overflow loses posted tasks in extreme cases. The `POST` function returns an error code when a task is dropped. While the responsibility to handle dropped tasks can be left to the programmer, CatNap implements a backpressure mechanism to minimize such overflows (Section 4.5.2) to reduce programmer effort.

A task can be interrupted any time by an event or a recharge. When interrupted, execution pauses and resumes later from the point where it was interrupted. Similarly, if power fails during a task, the system reboots to the point where power failed, relying on an existing technique called just-in-time (JIT) checkpointing [8, 34, 47] (Section 4.6).

***Events*** Events are units of computation that must run at a specific time, e.g., periodically or when an interrupt occurs. Events can be periodic or aperiodic, can be statically or dynamically scheduled, and cannot be interrupted by other events, tasks, or recharges. If power fails during an event, the system re-runs the interrupted event from its start on reboot for atomic execution. To rollback any partial results, CatNap

logs every access to the event's channel. Other event-local memory accesses need not be logged.

The scheduler tracks time and dispatches periodic events at their user-defined frequency. Aperiodic events can be posted by an interrupt handler, by another event, or by a task. The scheduler handles aperiodic events similarly to periodic ones by associating each aperiodic event with a minimum inter-arrival period. The programmer specifies this minimum interval based on domain knowledge. During scheduling, CatNap accounts for an aperiodic event as though it were periodic with its period equal to the minimum interval, which is the worst-case provisioning. To avoid over-consuming energy, the scheduler prohibits an aperiodic event being posted at a frequency higher than the frequency defined by the minimum interval.

***Channels*** The programmer can define a FIFO channel that allows a task or event to exchange values with another task or event. A channel allows directional data exchange: the *source* puts data into the channel and the *destination* removes data from the channel. Exchanging data between tasks or events outside of a channel is not allowed. The data structure of a channel element is arbitrary and programmer-defined.

Each channel element stores a logical time value indicating when the element was inserted. The logical time is a counter that increments on each power failure. An event or a task reading an element from a channel can compare the current logical time and the time value of the channel element to determine whether the element was produced after the most recent power failure. The logical time helps to handle data with a "timeliness" requirement [26], i.e., data that become stale and unusable if a certain amount of time passes. If power fails after data are collected and before they are used in a computation, an arbitrary amount of time could pass between data collection and use. The destination of the channel can discard the data in such cases to ensure timeliness, similar to prior work [26]. Channel makes CatNap free from data-races between tasks and events because channel operations execute atomically and other shared memory accesses are forbidden.

### 4.5 Degradation Logic

CatNap allows the programmer to specify rules by which task and event execution can gracefully degrade to decrease their energy use. There are several ways to specify degradation. Period degradation decreases event frequency. Parameter degradation exposes a parameter that, when varied, scales the amount of energy required by a task or event. Code degradation allows the programmer to specify multiple versions of task or event code that consume different amounts of energy. With degradations specified, CatNap gracefully degrades when events are infeasible. CatNap also degrades tasks when the task queue overflows.

#### 4.5.1 Graceful Degradation to Avoid Infeasibility.

Utilization varies depending on incoming power. A previously feasible set of events can become infeasible if the incoming power decreases. The scheduler invokes degradation when its estimate of utilization indicates infeasibility.

By default, the system is deemed infeasible if the utilization goes over 1. CatNap optionally allows the programmer to specify a *threshold utilization* $U_{thres}$ below 1, so that the system is considered infeasible if $\sum_{i=0}^{n} \frac{c_i}{t_i} > U_{thres}$. The computed utilization can be imprecise because it is based on measurements of limited precision; setting a conservative $U_{thres}$ adds a guard band to avoid scheduling insufficient recharges due to imprecise utilization estimates. Specifying $U_{thres}$ is an optional feature which is usually not necessary because CatNap's measurements have low error (Section 7.4).

*Period degradation* increases the period of an event, running less frequently and requiring commensurately less frequent recharges. The programmer can specify an event's *minimum* and *maximum* period and the scheduler tries to run the event at the highest possible frequency. The programmer can also define a *degradation function* specifying how to increase the period allowing, e.g., linear or exponential variation. CatNap allows correlating different events' periods so that they vary together (e.g., periods of a sensing event and a corresponding radio transmission should be degraded together). By default, CatNap doubles an event's period to degrade. When degrading, CatNap calculates the utilization's gradient with respect to each event's period and increases the period that will result in the maximum utilization decrease.

*Parameter degradation* decreases a *parameter* used by an event computation that determines the energy cost in an application-specific way. The programmer can define for each event one parameter variable. As with period degradation, the programmer can specify minimum and maximum tolerable values, a degradation function, and the relationship between different events' parameters. The meaning of the parameter is up to the programmer. It can be a configuration parameter of an I/O device (e.g., the transmission power of a radio) or a knob for an approximate algorithm such as loop perforation [59]. CatNap degrades the parameter associated with the event contributing the most to utilization with the degradation function. Parameter degradation also applies to tasks (Section 4.5.2).

*Code degradation* runs an alternative version of an event's code that uses less energy. The programmer can provide multiple different implementations of the event that provide interchangeable functionality and use the same channel interface. The programmer specifies the expected relative energy consumption of the events and CatNap degrades from one event variant to another by selecting the next highest energy variant. CatNap applies code degradation to the event contributing the most to utilization. Prior work [36, 61] studied

similar algorithmic relaxations for energy efficiency. Code degradation also applies to tasks (Section 4.5.2).

*Degradation memoization* stores the working degradation setting associated with the incoming power level. When the incoming power changes, CatNap reloads the memoized settings associated with the new incoming power if one exists, avoiding the need to search for a feasible setting. If a memoized setting is insufficient, CatNap searches for a new degradation setting, replacing the memoized ones. To avoid high memoization overhead, CatNap quantizes the incoming power into four bins and memoizes for each bin. Our prototype degrades different settings in a fixed order starting with periods, then parameters, then code (the order is not fundamental). Memoization can eliminate unnecessary searches for the degradation parameters, which can have high runtime overhead.

**4.5.2 Handling Task Queue Overflow.** CatNap uses degradation to mitigate task queue overflow as well. When the task queue is full, it applies backpressure to the scheduler which applies parameter and code degradation to tasks. After degrading a task, CatNap lets execution progress for a fixed interval and checks whether the overflow resolved. If not, CatNap degrades the task again. When the task queue overflow resolves, CatNap memoizes the task degradation configuration. With no task to degrade, CatNap reduces $U_{thres}$, which has the effect of increasing the time and energy available for tasks to clear the overflowed queue.

### 4.6 Handling Power Failure

CatNap may still suffer a power failure if incoming power is very low; a solar-powered device cannot run in the dark. CatNap tolerates power failures using JIT checkpoints, a well-studied solution for handling intermittent power failures [7, 8, 34, 47, 53]. JIT checkpointing monitors a device's remaining energy, saves state when a power failure is imminent, and stops executing until more energy accumulates.

As in prior work [34, 47], CatNap uses a fully non-volatile main memory which is commercially available [17, 57, 63] and only saves the register files on a checkpoint. If power fails in the scheduler or a task, CatNap restarts from where it left off. If power fails in an event, CatNap rolls back the event's channel and restarts the event.

## 5 Implementation

We implemented CatNap using a combination of commodity off-the-shelf hardware, some simple custom power system circuits, and its full software stack, including the programming model, compiler, and runtime system.

***Overview*** We implemented CatNap in C on a TI MSP430FR-5994 [63] microprocessor, harvesting RF energy into a 1mF capacitor with a dipole antenna and a P2110-EVB harvester [50]. To monitor stored energy and compare it with $e_{event}$, we used the MCU's built-in configurable comparator.

***Utilization Measurement*** Implementing charge scheduling requires tracking the utilization by measuring the incoming power and the worst-case energy consumption of each event. CatNap uses the MCU's analog-to-digital converter (ADC) to measure the voltage drop on the energy buffer across an event's execution, approximating the event's worst-case energy consumption by selecting the maximum value ever measured. An alternative design could profile worst-case energy statically (e.g., [16]).

CatNap estimates incoming power by measuring how much energy was accumulated during recharge, again using the ADC to measure the energy buffer voltage before and after a recharge. CatNap measures time spent charging using the MCU's scheduling timer, which is already running continuously to support periodic event scheduling.

CatNap must occasionally re-estimate $P$, the incoming power, because harvestable power may change over time. The frequency with which CatNap re-estimates $P$ determines the rate of long-term power fluctuation to which CatNap adapts. Instead of re-estimating $P$ at a fixed interval, CatNap does so on a transition from charging to an event or task execution. This design more frequently re-estimates $P$ when events are frequent or recharging is fast, avoiding unnecessarily frequent state changes while events are infrequent or charging is slow.

By measuring incoming power and the worst-case energy consumption of events dynamically, CatNap adapts to long-term incoming power fluctuations (e.g., less sunlight on a cloudy day) and variation in system power consumption (e.g., component degradation, temperature changes). As a result, CatNap's measurement-based utilization estimate closely matches real utilization (Section 7.4).

## 6 Benchmarks and Methodology

### 6.1 Evaluation Setup

We conducted end-to-end evaluations on a real energy-harvest-ing setup with a full prototype of CatNap, comparing with InK [65]. We could not compare against the original implementation of LSA [48] on real hardware because the implementation was not available and its multi-level priority scheme makes context switching expensive and challenging to implement efficiently for the platform we target. A simulated comparison with LSA was presented in Figure 3.

For the evaluations, we ran CatNap and InK harvesting 915MHz radio waves generated by a ThingMagic Astra-EX RFID reader. We used InK as published with minor changes to support time-keeping. Rather than using a custom persistent clock [65], we emulated a persistent clock with an MCU clock powered by a dedicated energy buffer.

### 6.2 Event-Driven Benchmarks

We ran mixtures of event-driven benchmarks with each designed to have distinct, representative characteristics.

***Audio Sampling 0 (AUD0)*** represents a class of *time-critical, energy-hungry, low frequency, degradable, and periodic* workloads. AUD0 samples audio at 12kHz using a SPU0414HR5H MEMS microphone. The program has a strict timing requirement to collect a block of audio samples every 2.3 seconds. Each block ideally consists of at most 500 samples, while the number can be degraded to accommodate the deadline of 2.3 seconds. Other energy-hungry, time-critical workloads that are degradable, e.g., cameras with adjustable resolution or long-range (LoRa) radios with adjustable transmission power [31] also fall into this category. *Requirement: (1) sample a block every 2.3 seconds, (2) collect as many samples as possible, up to 500.*

***Audio Sampling 1 (AUD1)*** represents a class of *time-critical, energy-hungry, low frequency, non-degradable, and periodic* workloads. AUD1 is similar to AUD0 except it is not degradable, i.e., it has to always sample exactly 300 audio data points every 5.7 seconds. AUD1 represents the events without a degradable knob. *Requirement: (1) sample a block every 5.7 seconds, (2) collect exactly 300 samples per block.*

***Audio Sampling 2 (AUD2)*** represents a class of *time-critical, less energy-hungry, high frequency, non-degradable, and periodic* workloads. AUD2 continuously collects audio data every 0.1 second. AUD2 is different from AUD0 or AUD1; instead of collecting a burst of high frequency signals, it continuously monitors a low frequency signal. *Requirement: (1) sample an audio signal every 0.1 second.*

***Temperature Monitoring (TEMP)*** represents *time-critical, less energy-hungry, middle frequency, non-degradable, and periodic* workloads. TEMP measures temperature using an on-chip sensor [63] every 0.57 seconds and cannot be degraded. Unlike AUD0 and AUD1, AUD2 and TEMP consumes less energy per execution but is more frequently executed. Typical sensor readings usually fall into this category. *Requirement: sense every 0.57 second.*

***Button Press Detection (BTN)*** represents *time-critical, less energy-hungry, middle frequency, non-degradable, and reactive* workloads. BTN detects a button presses at most every 0.57 seconds and cannot be degraded. We emulate button presses from a second MSP430FR5994 MCU's GPIO. BTN represents a reactive workload whose arrival time cannot be precisely estimated. Peripherals that generates a hardware interrupt usually fall into this category. *Requirement: detect the randomly occurring GPIO signal that can occur as frequently as every 0.57 second.*

***Downsampling (DSP)*** represents *time-insensitive and degradable* workloads. DSP performs downsampling over 300 data points with a square filter, downsampling at a 20:1 ratio. The quality of the filter can vary, using a convolution approximation from ParaProx [56]. Many signal processing workloads with a well-studied approximation can fall into this category. *Requirement: (1) downsample 300 samples with 20:1 ratio, (2) use the highest filter quality possible.*

***Fast-Fourier Transform (FFT)*** represents *time-insensitive and non-degradable* workload. FFT takes 16 data points as an input to compute an FFT. Although a degradable FFT algorithm might exist, we did not adopt such degradation to represent a set of time-insensitive workloads that cannot be degraded. *Requirement: (1) Compute FFT over 16 samples.*

The benchmarks and their characteristics are summarized in Table 2. Since DSP and FFT are time-insensitive, they are tasks in CatNap, while others more as an event. Because CatNap does not consider the energy use of a task, we do not characterize or vary the energy use of DSP and FFT (marked as -). Tasks also do not have an inherent frequency or periodicity constraints. We ran all the benchmarks with the device placed 65cm away from for the RF supply.

**Table 2.** Summary of benchmarks. DSP and FFT are tasks, whose energy use is not an important concern and do not have an inherent frequency or periodicity (marked as -).

| name | time-critical? | energy-hungry? | frequency? | degradable? | category |
|---|---|---|---|---|---|
| **AUD0** | O | O | low | O | periodic |
| **AUD1** | O | O | low | X | periodic |
| **AUD2** | O | O | high | X | periodic |
| **TEMP** | O | X | med | X | periodic |
| **BTN** | O | X | med | X | reactive |
| **DSP** | X | - | - | O | - |
| **FFT** | X | - | - | X | - |

### 6.3 Performance Benchmarks

We also ran intermittent performance benchmarks used in prior work [15, 41, 45, 64] to evaluate the overhead of CatNap. A subset of the benchmarks was also used in InK's evaluation [65], allowing direct comparison. The benchmarks consist of six compute-intensive IoT programs. **CEM** LZW-compresses data. **CF** stores and searches numbers in a cuckoo filter. **RSA** encrypts a string using a 64-bit key. **AR** classifies accelerometer data with a nearest-neighbor classifier. **BF** encrypts a string using Blowfish encryption. **BC** counts the number of 1s in a bitstream.

We use four applications (CEM, CF, AR, BC) written for InK by its authors and ported ones that were omitted from InK's evaluation [65]. For CatNap, we ran plain C versions of the benchmarks from prior work [45], running code in a task. We again ran all tests 65cm away from the RF supply.

## 7 Evaluation

Our evaluation uses the benchmarks from Section 6 to answer the following four questions:

- Does CatNap support periodic and reactive events?
- What are CatNap's main overheads?
- Does CatNap precisely estimate utilization?
- Is CatNap practically applicable?

### 7.1 CatNap Supports Event-Driven Execution

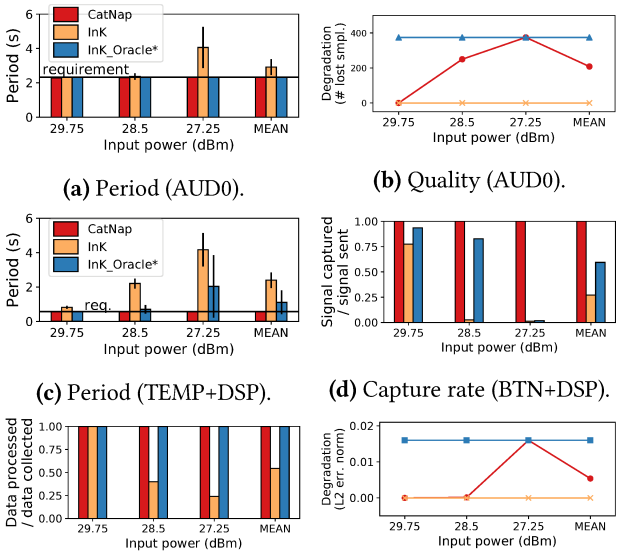
We compared CatNap's ability to execute periodic and reactive programs with InK [65], a state-of-the-art event-driven kernel for energy-harvesting devices whose implementation is available. We evaluated whether the system can support periodic and reactive execution when a mixture of events and tasks with different characteristics run together (Section 6.2). *We only present the subset of the evaluations where the specific mixture of the benchmarks and the power level resulted in interesting behavior.* The omitted data points showed trivial results (e.g., all systems running or failing) or trends similar to those included.

**7.1.1 CatNap Degrades Events Properly (AUD0).** CatNap properly degrades the quality of the event on a long-term variation of the incoming power. Figure 6a summarizes how the period of AUD0 changed as we vary the input power level with both CatNap and InK. As the result shows, CatNap always executed AUD0 with constant, user-specified frequency. However, InK violated the timing requirement by 177% on average at 27.25dBm RFID reader power. Figure 6b shows quality degradation for each power level, reporting samples lost out of 500. CatNap scales quality to meet the timing requirement, while InK cannot scale.

To avoid the deadline violation, an InK programmer must have an oracular knowledge of the possible incoming power that the device may ever experience and manually tune the event parameters before deployment. While such an assumption is unrealistic, we additionally compare to such a manually-optimized InK by an oracular programmer (InK_Oracle). CatNap is still better than InK_Oracle; CatNap collects 2.33× more samples by dynamically increasing collected samples with abundant power while InK_Oracle is statically provisioned for the worst-case.

**7.1.2 CatNap Isolates Events and Tasks (TEMP+DSP).** CatNap can keep the periodic event's schedule even when there is a concurrently running task. Figure 6c shows the period of TEMP while running DSP concurrently. We again included InK_Oracle, which manually degraded DSP as much as possible to save more time and energy for TEMP. CatNap always met TEMP's requirement of 0.57 second, while even InK_Oracle abundantly violated the timing requirement. InK and InK_Oracle violate the timing requirement by 732% and 358%, respectively, at 27.25dBm. InK_Oracle meets TEMP's deadline better than InK because it uses less energy for DSP and hence more for TEMP. However, even InK_Oracle still fails due to the lack of energy isolation; when the time-insensitive computation (DSP) uses up all the energy, InK and InK_Oracle fail to run a time-critical measurement (TEMP) properly.

**7.1.3 CatNap Supports Reactive Events (BTN+DSP).** CatNap can reserve and isolate energy for the possible future reactive event (BTN), even when there is a concurrently running task. Figure 6d shows the percentage of the button press events (BTN) that were captured and processed with different incoming power level, while DSP is running concurrently. We again included InK_Oracle, which manually degraded DSP as much as possible to save more time and energy for BTN. While CatNap captured all button presses, InK and InK_Oracle missed most of the button presses, capturing only 1.4% and 1.8% respectively on low input power (27.25dBm). On average across power levels, InK and InK_Oracle only captured around 25% and 50% of the button presses. CatNap reserves energy for BTN and isolates it from DSP. InK and InK_Oracle allow DSP to use too much energy, leaving insufficient energy for BTN when a button press arrives.




**(a)** Period (AUD0). **(b)** Quality (AUD0).

**(c)** Period (TEMP+DSP). **(d)** Capture rate (BTN+DSP).

**(e)** Process rate (AUD1+DSP). **(f)** Quality (AUD1+DSP).

**Figure 6.** Result of different benchmarks. The (a) period and (b) quality for AUD0, the (c) period of TEMP+DSP, the (d) capture rate of BTN+DSP, and the (e) portion of processed data and the (f) quality for AUD1+DSP is shown. User-specified requirement is shown as a horizontal line (a, c). *InK_Oracle was manually tuned for best performance.

**7.1.4 CatNap Degrades Tasks Properly (AUD1+DSP).** CatNap can properly degrade the quality of the task when the task queue overflows, using its backpressure mechanism 4.5.2. A long-running task like DSP poses a risk of overflowing the task queue if events post tasks more quickly than the task completes. Such a task queue overflow loses data. InK uses *pipes* [65] that act as a task queue, exposing a similar problem. Figure 6e shows the fraction of collected data processed by DSP's filtering task without being lost due to the task queue/pipe overflow. Figure 6f shows the degraded performance of the filter, which we report by calculating the L2 relative error norm ($\frac{||degraded-original||}{||original||}$) of

downsampled data. CatNap's backpressure avoids task queue overflow by degrading the tasks, sacrificing the quality of the downsampling on low input power (Figure 6f). In contrast, InK loses 76% of the collected data at 27.25dBm.

InK_Oracle represents an oracular programmer statically configuring the filter accuracy, assuming knowledge of future input power (InK_Oracle). In our experiment, InK_Oracle was able to process all the data collected; however, InK_Oracle suffered from 2.97× higher mean quality degradation because it cannot adaptively increase quality when power is abundant.

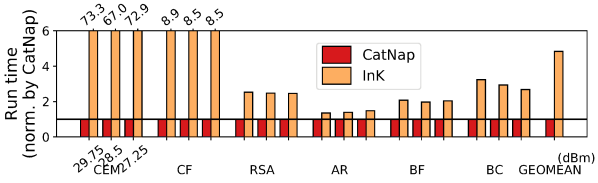
**7.1.5 CatNap Fails Gracefully (AUD2+FFT).** While running AUD2 with FFT, we observed that CatNap fails gracefully with an infeasible setup. AUD2 + FFT requires that FFT with 16 samples be calculated every 1.6 seconds. With a power level of 29.75dBm, such parameters were too demanding for any system to reliably meet the deadline.

While operating on such an infeasible operating point, even CatNap experienced **31** power failures because the system is neither feasible nor degradable. CatNap still collected samples at 10Hz (AUD2) when there was no power failure, correctly computing **17 of 17** FFTs attempted. InK, however, *incorrectly* computed 19 FFT results, because InK could not consistently sample at 10Hz. The interference of the FFT computation with AUD2 led InK to have an average sampling period of **0.4s** and a high standard deviation of **0.47s**. FFT results on such non-uniformly sampled data are not meaningful. The comparison shows that CatNap meets timing requirements (i.e., AUD2) better than InK because of energy isolation, even when both fail to run perfectly.

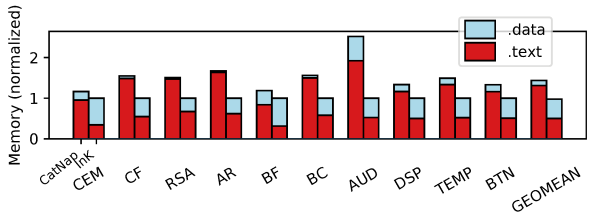
## 7.2 CatNap Has High Performance

CatNap has a higher performance than InK. Figure 7a shows the end-to-end execution time of the energy-harvesting performance benchmarks (Section 6.3) with different input power levels. CatNap outperforms InK by 4.84× on average, with up to 73.3× performance benefit. The main performance benefit of CatNap comes from using JIT checkpointing (Section 4.6) to keep memory consistent on a power failure. Instead, an InK programmer decomposes a program into multiple small code chunks and logs data shared between them at their boundaries [65]. Prior work showed that JIT checkpointing has a 2.7×-4.1× run time benefit over a user-specified, boundary-based approach [47], which is consistent with our results. InK is even slower in some cases (e.g., CEM) because InK logs *all* shared memory at each boundary using DMA, scaling overhead with memory use.

CatNap's scheduler overhead is also reasonably low. To measure the scheduler overhead that varies with the number and period of events, we ran sets of one, ten, and thirty dummy events with frequencies between 1 second to 8 microseconds for a single benchmark, CEM. We only report the result with the RF power at 29.75dBm for brevity, although




**(a)** Execution time.




**(b)** Memory use.

**Figure 7.** Performance and memory use of CatNap.

the trend was not sensitive to input power. The overhead was only **0.77%** of the total execution time with thirty events running every 0.25 seconds. With ten events running every second, the overhead was **0.12%**. The data show that the overhead is reasonably low for practical use.

## 7.3 CatNap Has Modest Memory Overhead

We evaluated CatNap's memory overhead, inspecting our performance benchmarks (CEM–BF) and event-driven benchmarks (AUD0–FFT). Figure 7b shows that CatNap incurs 1.5× higher memory overhead than InK on average, most of which is due to its runtime library code (i.e., .text). The 50% memory overhead is the cost of CatNap's precise event-driven execution and high performance.

## 7.4 CatNap Precisely Estimates Utilization

CatNap assesses the feasibility of a schedule based on its utilization estimates. We assessed the accuracy of CatNap's utilization estimates on real energy-harvesting hardware by studying its behavior with different static input power levels, then with time-variant power.

***Fixed Input Power*** To evaluate the accuracy of the utilization estimate, we ran a large collection of parameterized microbenchmarks with degradation disabled. Each microbenchmark has 1–3 events, each running a spin-loop consuming a small, medium, or large amount of energy, with a period ($T$) of 0.5 one second. The RF transmitter 30cm away was configured to emit one of twelve power levels between 29.75dBm and 23.75dBm for one minute. We logged CatNap's utilization estimates and compared them with the measured utilization. Utilization is essentially the portion of time spent replenishing the energy used, which was measured with a voltage comparator. We plotted measured utilization over 1 (i.e., the energy never fully replenishes) as 1.1.

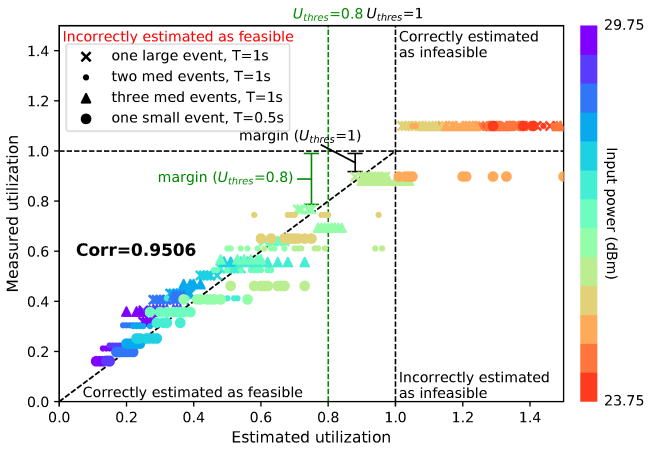
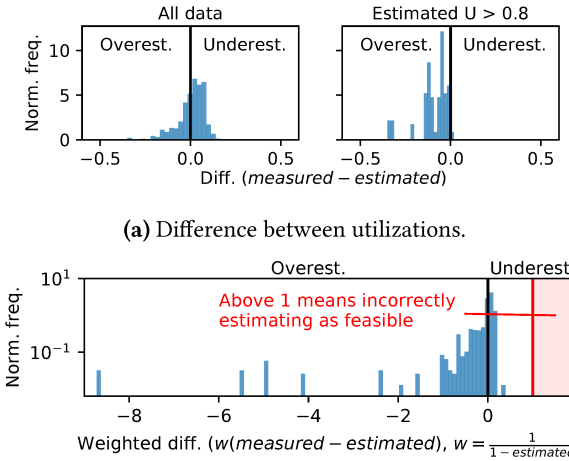



**Figure 8.** Accuracy of the utilization calculation. The estimated utilization versus the measured utilization is plotted. Different shapes indicate different applications and different colors indicate different incoming power.




**(a)** Difference between utilizations.

**(b)** Weighted difference between utilizations.

**Figure 9.** Additional characterization of Figure 8. The histogram shows the difference between utilizations from Figure 8 (a) without and (b) with a weight of $w = \frac{1}{1-estimated}$.

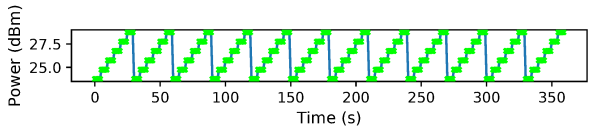
Figure 8 plots 2496 pairs of estimated (x-axis) and measured (y-axis) utilizations. We distinguish microbenchmarks with shapes and RF power levels with colors. CatNap's estimates are accurate: most points lie on or slightly below the line, $y = x$, showing a *high correlation of 0.9506* when estimated utilization is below 1 ($x < 1$). We could not calculate the correlation above $x >= 1$ because the measured utilization clamps at 1.1. Points below the line mean utilization *may have* been overestimated. Points may still lie below $y = x$, even with perfect estimation because the estimate assesses *worst-case* utilization. Overestimating may degrade an already feasible schedule unnecessarily, if estimated utilization is above one and actual utilization is below one ($x >= 1$, $y < 1$). Overestimation may thus degrade quality, but will *not* lead to insufficient recharging that causes a timing violation.

The data show that CatNap rarely *underestimates* utilization: few points lie above $y = x$. Underestimation is potentially problematic if an infeasible schedule is incorrectly considered feasible, which can cause a power failure or event timing violation (i.e., if points in $x < 1$, $y >= 1$). We never observed an instance where CatNap incorrectly assumes an infeasible environment is feasible.

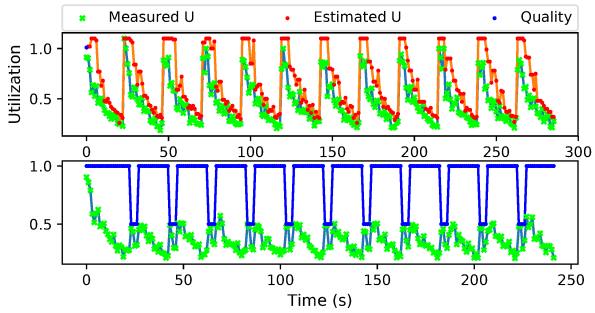
Points in $x <= 1$, $y <= 1$ and $x > 1$, $y > 1$ are instances correctly estimated as feasible or infeasible. As long as points are within these regions, overestimation or underestimation is still safe. The data reveal a 10.3% margin between the dangerous $x <= 1$, $y > 1$ region and its nearest point — an empirical guard-band in the utilization estimates. The green line shows that with a conservative $U_{thres}$ of 0.8 instead of 1.0, there is a margin increase to 23.3%. With the conservative threshold, the chances of unnecessary degradation also increased (points in $0.8 <= x < 1$, $y < 1$ additionally triggers unnecessary degradation).

***Detailed Characterization*** To further evaluate data in Figure 8, we look into two metrics representing how accurate the estimated utilization is compared to the measured utilization. We plot the **difference** between the measured and estimated utilization ($measured - estimated$, Figure 9a). We preserve the sign which additionally gives us information on whether the system is overestimating (negative sign) or underestimating (positive sign). Figure 8 shows the histogram of the difference of (1) all data (Figure 9a, left) and (2) the data whose estimated utilization is over 0.8 (Figure 9a, right). Although CatNap sometimes overestimates and sometimes underestimates utilization by a modest amount (left), the plot shows that, crucially, CatNap rarely underestimates near high utilization (right). CatNap tends to underestimate by less when utilization is high, avoiding an incorrect judgment that an infeasible schedule is feasible (recall that underestimation leads to a scheduling failure only when real utilization goes above one). The relatively frequent underestimation at low utilization (left) is due to the assumption that event execution time is negligible compared to recharge time (Section 3.2); low utilization invalidates this assumption. However, underestimation in low utilization is safe.

We also define the **weighted difference**, which is the difference (as above) multiplied by a weight, $w = \frac{1}{1-estimated}$. The weight grows larger as estimated utilization approaches one, reflecting the fact that an error is critical when the utilization is near one, but that low utilization tolerates more error. A negative value represents an overestimation and a positive value is an underestimation. Among the underestimated cases, *values over one mean the system is incorrectly assessing an infeasible set of events to be feasible*, which is a critical failure for CatNap. Figure 9b plots the weighted




**(a)** Incoming power.




**(b)** Utilization.

**Figure 10.** Utilization estimation over time. (a) shows an example patterns of an input power. (b) shows the utilization estimated (red) and measured (green) with degradation disabled (top), and the utilization measured (green) and the output quality (blue) with degradation enabled.

difference for all trials. The figure shows that even when the system underestimates ($x > 0$), the weighted difference is far from one: its maximum is below 0.4. The result implies CatNap's robustness against incorrectly estimating feasibility and failing to schedule recharges properly. The values span from below −8 on the low end, showing that CatNap is prone to overestimation that may lead to unnecessary degradation. Occasionally imposing such conservative degradation is the main drawback imposed by CatNap as it achieves the highly reliable schedulability demonstrated in Section 7.1.1–7.1.5.

***Time-Varying Incoming Power*** We evaluated CatNap's ability to adapt to long-term power variation, running a microbenchmark (large energy cost, $T = 1s$). We varied power from 29.75dBm to 23.75dBm following several power traces. We show one example trace of a sawtooth in Figure 10a. We also tested with a triangle wave and a reverse sawtooth wave. We tested each trace with a period of 30, 60, and 120 seconds.

Figure 10b shows estimated and measured utilization with varying power with and without degradation. Degradation halved the spin-loop iteration count in the event and we model quality loss as $\frac{reduced\ loop\ size}{original\ loop\ size}$. Results from other waves are omitted; the trends are similar to Figure 10b The upper plot shows that the estimated utilization (red) is always above actual utilization (green line) and the estimate updates rapidly as power changes. CatNap estimates perfectly or overestimates, but never dangerously underestimates, even as power drops sharply. The lower plot shows that the measured utilization (green) is consistently below 1.0, while the quality (blue line) degrades in response to sharp drops in input power. The data show CatNap's ability to adapt to varying input power by degrading the quality.

### 7.5 Limitations of CatNap

Although CatNap successfully ran periodically and reactively on our benchmarks (except for FFT), there are scenarios where CatNap can fail. We summarize cases where CatNap may fail. First, if CatNap cannot further degrade and the utilization is above one, the situation is infeasible and CatNap may fail to run periodically or reactively. Such a situation is like *starvation* in traditional scheduling, where some events use too much energy for others to run. Even in such an infeasible situation, however, CatNap fails gracefully compared to prior work (Section 7.1.5). Second, if CatNap underestimates an infeasible system as feasible, CatNap may fail to schedule sufficient recharges, although our empirical study shows that such underestimation is unlikely (Figure 8–9). To mitigate these possible failures, the programmer can adjust $U_{thres}$ to be more conservative (Section 4.5). Third, if the incoming energy rapidly changes, CatNap may schedule recharges with an outdated utilization value. However, from Figure 10, we show that with reasonable energy fluctuation this error mode is not an issue. Finally, under heavy load, CatNap may cause tasks to run with a large execution time variance, as shown in Figure 3b. In an extreme case, this high variance may lead to a task queue overflow. It is the programmer's responsibility to write a program to tolerate these overflows.

### 7.6 Case Study: Particle Filter Orbit Estimation

As a case study demonstrating the applicability of CatNap, we built a *satellite orbit estimator* appropriate for chip-scale satellites [17, 66] in low-earth orbit. The estimator uses a particle filter [52] with 200 particles to determine the chipsat's orbital position by fitting sequential magnetometer readings to a stored map of Earth's magnetic field [49]. The application goal is to frequently enough run the particle filter, which is complicated by orbiting into and out of Earth's eclipse. In eclipse, harvestable energy is about 10%-20% of sunlight energy, and CatNap degrades the number of filter particles to reduce energy cost. Every two minutes, an event reads the magnetometer and posts a particle filter task. We emulated magnetic field readings by simulating the chipsat's orbital dynamics (i.e., position and velocity) and reporting magnetic field data at its position, using a field map [49]. We simplistically assume an equatorial low-Earth orbit, a 90 minute period, 2.5mW harvestable power in sun, and 0.5mW in eclipse. We used the RFID reader to emulate solar energy. In general, simulating solar energy with RF energy is not ideal because their characteristics, e.g., the I-V curve, can be different. We use RF power to emulate the proportion of incoming energy in the sunlight/eclipse cycle experienced by the solar-harvesting system only and we do not depend on the similarity of low-level power characteristics.

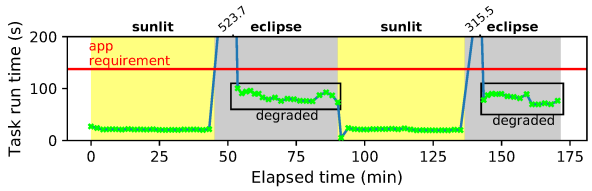



**Figure 11.** Orbit detection using a particle filter.

Figure 11 shows a time series illustrating how CatNap adapts to the changing orbital power environment. In sunlight, tasks are fast. In eclipse, power drops. The first task's time violates the deadline because CatNap does not degrade tasks in progress. CatNap degrades subsequent tasks, which finish within the deadline. The data suggest that CatNap practically supports sophisticated state estimation problems.

## 8 Additional Related Work

***Intermittent and Energy-Harvesting Systems*** Section 1 and Section 2 covered prior intermittent systems [6, 8–11, 15, 16, 26, 28, 34, 41, 44–46, 51, 55, 64, 65], energy-harvesting platforms and hardware [17, 24, 25, 27, 32, 57, 61, 68], and energy-harvesting RTOS-based systems [12, 13, 22, 39, 48]. Section 2 also introduces low-power embedded systems [18, 19, 37, 38, 61] that influenced the design of CatNap. They all fail to satisfy requirements for event-driven execution on an energy-harvesting device (Table 1).

Others target specific application domains on energy-harvesting devices, including neural networks [23], communication [42], wildlife tracking [35], and multi-tenant sensing [2]. Incidental computing [43] and WN [21] approximate intermittent computation in the architecture. Some work helps develop intermittent systems, including software updates [1, 62], debugging [14, 69], and modeling [58].

***Approximate Computing*** A large body of work has been done on trading the quality of the system and the system performance by tuning a parameter [29, 30, 56, 59] or selecting between a user-provided code [3–5, 36]. These work can be applied to CatNap in building degradable events and tasks.

***Energy-aware OS*** Prior energy-aware systems bear similarities with CatNap in that they dynamically adjust the energy use of applications. These systems, however, mainly concentrate on energy-efficiency, while CatNap's main focus is also in deciding *when* to use energy or recharge.

Eon [61] degrades application quality by monitoring the incoming power. Using a rechargeable battery, Eon has the luxury to operate on a coarse time scale and does not assume frequently being interrupted due to low energy. CatNap, whose design choice was to exclude the battery, must make much more fine-grained decisions on when to recharge or execute with an RTOS-style recharge scheduler.

Koala OS [60] degrades system parameters, e.g., clock speed, to achieve energy-efficiency. Adopting such a system-level degradation is an interesting future work for CatNap.

Cinder OS [54] allows the programmer to delegate portions of the available energy as a resource to different applications, isolating energy between them. The concept of isolating energy is similar to CatNap's. Cinder OS assumes a reliable power supply and applications continuously consuming a portion of the supply. The model is not directly applicable in the intermittent event-driven context, where the incoming power is weak and events and tasks must *take turns* in consuming energy.

LAB [36], JouleGuard [29], and PowerDial [30] automatically find the optimal approximation balancing performance and energy. CatNap's trial-and-error-based parameter finding can be improved by adopting such theoretical models. These systems, however, concentrate on improving energy efficiency rather than scheduling energy use.

## 9 Conclusion

This paper presents CatNap, a system with a new programming model enabling reliable event-driven execution on energy-harvesting devices in environments where prior systems failed. CatNap truly achieves event-driven execution on energy-harvesting devices, opening up new possibilities for energy-harvesting applications.

## Acknowledgments

We thank the anonymous reviewers for the insightful feedback and to our shepherd Daniel Barowy for the help in refining our final manuscript. This work was funded in part by a KFAS scholarship and by National Science Foundation Award #1751029.

## References

[1] Henko Aantjes, Amjad Y Majid, Przemyslaw Pawełczak, Jethro Tan, Aaron Parks, and Joshua R Smith. 2017. Fast downstream to many (computational) RFIDs. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 1–9.

[2] Joshua Adkins, Bradford Campbell, Branden Ghena, Neal Jackson, Pat Pannuto, and Prabal Dutta. 2016. The Signpost Network: Demo Abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM (SenSys '16)*. ACM, New York, NY, USA, 320–321. https://doi.org/10.1145/2994551.2996542

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland. http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 85–96.

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

[6] Sara S Baghsorkhi and Christos Margiolas. 2018. Automating efficient variable-grained resiliency for low-power IoT systems. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 38–49.

[7] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. 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 (2016), 1968–1980.

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

[9] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (2018).

[10] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 209–219.

[11] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-aware Runtime for Computational RFID. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 197–210.

[12] Wei-Ming Chen, Tai-Sheng Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. 2016. Value-Based Task Scheduling for Nonvolatile Processor-Based Embedded Devices. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*. IEEE, 247–256.

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

[14] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 577–589. https://doi.org/10.1145/2872362.2872409

[15] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

[16] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 116–127.

[17] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. 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)*. ACM, New York, NY, USA.

[18] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462.

[19] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. 2005. Nano-rk: an energy-aware resource-centric rtos for sensor networks. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, 10–pp.

[20] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh Gupta. 2018. Pible: battery-free mote for perpetual indoor BLE applications. In *Proceedings of the 5th Conference on Systems for Built Environments*. ACM, 168–171.

[21] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. 2019. The WhatâĂŹs Next Intermittent Computing Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 211–223.

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

[23] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2018. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. *arXiv preprint arXiv:1810.07751* (2018).

[24] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/2809695.2809707

[25] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 19.

[26] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Conference on Embedded Networked Sensor Systems (SenSys 2017)*. ACM, New York, NY, USA.

[27] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P Burleson, and Jacob Sorber. 2016. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 4 (2016), 77.

[28] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 228–240.

[29] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 198–214.

[30] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 199–212.

[31] Hoperf Electronic. 2019. RFM95/96/97/98(W) - Low Power Long Range Transceiver Module. https://cdn.sparkfun.com/assets/learn_tutorials/8/0/4/RFM95_96_97_98W.pdf. (2019).

[32] Neal Jackson, Joshua Adkins, and Prabal Dutta. 2019. Capacity over Capacitance for Reliable Energy Harvesting Sensors. (2019).

[33] JADAK. 2017. ASTRA-EX 2-Port Integrated UHF RFID Reader. https://rfid.atlasrfidstore.com/hubfs/1_Tech_Spec_Sheets/Thingmagic/ATLAS%20ThingMagic%20Astra-EX%20Integrated%20RFID%20Reader%20JADAK.pdf. (2017), 2 pages.

[34] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. 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*. IEEE, 330–335.

[35] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. 2002. Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/605397.605408

[36] Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola. 2013. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 661–676.

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

[38] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

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

[40] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[41] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[42] Kaisheng Ma, Xueqing Li, Mahmut Taylan Kandemir, Jack Sampson, Vijaykrishnan Narayanan, Jinyang Li, Tongda Wu, Zhibo Wang, Yongpan Liu, and Yuan Xie. 2018. NEOFog: Nonvolatility-Exploiting Optimizations for Fog Computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 782–796.

[43] Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2017. Incidental Computing on IoT Nonvolatile Processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 204–218. https://doi.org/10.1145/3123939.3124533

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

[45] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2017)*. ACM, New York, NY, USA.

[46] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *OSDI*.

[47] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints. In *Proceedings of the 40 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM.

[48] Clemens Moser, Davide Brunelli, Lothar Thiele, and Luca Benini. 2007. Real-time scheduling for energy harvesting sensor nodes. *Real-Time Systems* 37, 3 (2007), 233–260.

[49] National Centers for Environmental Information (NOAA). 2017. Enhanced Megnetic Model (EMM). https://www.ngdc.noaa.gov/geomag/EMM/. (2017).

[50] Powercast Inc. 2010. Evaluation Board for P2110 PowerharvesterâĎć Receiver. http://https://datasheet.octopart.com/P2110-EVB-Powercast-datasheet-15540333.pdf. (2010), 3 pages.

[51] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. (2011), 159–170. https://doi.org/10.1145/1950365.1950386

[52] Branko Ristic, Sanjeev Arulampalam, and Neil Gordon. 2003. *Beyond the Kalman filter: Particle filters for tracking applications*. Artech house.

[53] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V Merrett, and Alex S Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18, 1 (2018), 172.

[54] Arjun Roy, Stephen M Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. 2011. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*. ACM, 139–152.

[55] Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency for Intermittent Systems. In *Proceedings of the 40 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM.

[56] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 35–50.

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

[58] Joshua San Miguel, Karthik Ganesan, Mario Badr, and Natalie Enright Jerger. 2018. The EH model: Analytical exploration of energy-harvesting architectures. *IEEE Computer Architecture Letters* 17, 1 (2018), 76–79.

[59] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 124–134.

[60] David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. 2009. Koala: A platform for OS-level power management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 289–302.

[61] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/1322263.1322279

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

[63] TI Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf. (2017), 19 pages.

[64] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 17–32.

[65] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 41–53.

[66] Zac Manchester. 2015. KickSat. http://zacinaction.github.io/kicksat/. (2015).

[67] Fengxiang Zhang and Alan Burns. 2009. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Comput.* 58, 9 (2009), 1250–1258.

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

[69] Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob Sorber. 2011. Ekho: Bridging the Gap Between Simulation and Reality in Tiny Energy-harvesting Sensors. In *Proceedings of the 4th Workshop on*

*Power-Aware Computing and Systems (HotPower '11)*. ACM, New York, NY, USA, Article 9, 5 pages. https://doi.org/10.1145/2039252.2039261