Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints

Kiwan Maeng
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
kmaeng@andrew.cmu.edu

Brandon Lucia
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
blucia@andrew.cmu.edu

Abstract

Batteryless energy-harvesting devices have the potential to be the foundation of applications for which batteries are infeasible. *Just-In-Time checkpointing* supports intermittent execution on energy-harvesting devices by checkpointing processor state right before a power failure. While effective for software execution, Just-In-Time checkpointing remains vulnerable to unrecoverable failures involving peripherals (e.g., sensors and accelerators) because checkpointing during a peripheral operation may lead to inconsistency between peripheral and program state. Additionally, a peripheral operation that uses more energy than a device can buffer never completes, causing non-termination.

This paper presents Samoyed, a Just-In-Time checkpointing system that safely supports peripherals. Samoyed correctly runs user-annotated peripheral functions by selectively disabling checkpoints and undo-logging. Samoyed guarantees progress by energy profiling, dynamic peripheral workload scaling, and a user-provided software fallback routine. Our evaluation shows that Samoyed correctly executes peripheral operations that fail with existing systems, achieving up to 122.9x speedup by using accelerators. Samoyed preserves the performance benefit of Just-In-Time checkpointing, showing 4.11x mean speedup compared to a recent possible alternative. Moreover, Samoyed's unique ability to profile energy and to dynamically scale large peripheral operations simplifies programming.

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

Keywords intermittent computing, energy-harvesting

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314613

ACM Reference Format:
Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314613

1 Introduction

The combination of low-power microcontrollers and energy-harvesting power systems enable sophisticated batteryless devices. Such *energy-harvesting devices* collect energy from their environment, charging an energy buffer and operating for a short window using the buffered energy. Because these devices do not need a battery to operate, they can be deployed on an environment where replacing the battery is often infeasible, such as environmental monitoring [34], in-body sensing, or sensing in outer space [71]. While operating, the device computes and manipulates peripherals such as sensors, radios, and specialized architectural accelerators [17, 56]. When the buffer is depleted, power fails, erasing volatile state (e.g., registers and SRAM) but preserving non-volatile state (e.g., FRAM or Flash). The device then recharges until accumulating sufficient energy when it reboots and continues operating. Software executes *intermittently*, only when energy is available [15, 30, 40, 44, 65].

Prior work enables correct intermittent execution with system support. *Checkpointing systems* [6, 32, 40, 45, 54, 65] save the volatile state of the device to non-volatile memory, or *checkpoint*, before the state gets erased by a power failure. On reboot, the checkpointed state is restored, resuming execution from the checkpointed program point. *Static checkpointing* systems rely on the compiler [4, 16, 45, 46, 54, 65] or the programmer [15, 40, 44] to decide where to checkpoint in code. Static checkpointing allows precise control over where to checkpoint, which helps avoid checkpointing during sensitive peripheral accesses that should not be interrupted by a checkpoint. However, these systems require accurate prediction of the energy consumption of code. Poorly placed checkpoints cause non-termination if they are too rare and a performance cost if they are too frequent [16, 45].

Just-In-Time (JIT) checkpointing systems [5, 6, 32, 33], on the other hand, monitor the device's remaining energy and

take a checkpoint only once before a power failure is imminent. JIT checkpointing avoids the checkpoint frequency problem of static solutions, provides a performance advantage, and can simplify programming [6, 32] (claims we empirically validate in Section 2.3).

Despite their benefits, prior JIT checkpointing systems have limitations that render them incompatible with peripheral devices, such as sensors, radios, and architectural accelerators [55] (i.e., "peripherals"). Peripheral devices for sensing and communicating are extremely common in embedded sensor systems, and intermittent systems must safely support these peripherals. Computation accelerators [10, 25] are growing increasingly important, as applications offload more computation (such as machine learning inference) to the edge and beyond [23].

A key design requirement for JIT checkpointing is that the system *must* be able to checkpoint at any moment as power fails. This design requirement conflicts with the need for a peripheral operation to complete *without interruption* by a checkpoint once started. Section 2.4 characterizes this conflict, showing that existing JIT checkpointing systems suffer unrecoverable failures when using peripherals. Moreover, when a peripheral operation requires more energy than the device can buffer (making an application infeasible), no existing system supports safely decomposing the operation into smaller, feasible operations. Due to the limitations, many previous systems instead focus on an alternate non-JIT checkpointing design that allows safe use of peripherals [15, 30, 40, 44, 45], sacrificing performance.

This work introduces Samoyed, [1] the first JIT checkpointing system to correctly and efficiently support multiple different types of peripheral operations in an intermittent execution on an energy-harvesting device. Samoyed provides programmers with a *peripheral atomic function* primitive, the implementation of which supports uninterruptable peripheral operations that are safely compatible with JIT checkpoints. Samoyed statically profiles peripheral operation energy to estimate the viability of the peripheral. Samoyed can then dynamically decompose a long-running peripheral operation into multiple smaller operations, or fall back to a simpler, alternative implementation if energy is insufficient.

We prototyped Samoyed for a modern energy-harvesting hardware platform [17]. The prototype includes a compiler, runtime library and energy profiler that together provide Samoyed's JIT checkpointing and atomic function primitives. We evaluated Samoyed to show that it enables safe peripheral operations, while prior systems fail. Samoyed permits the safe use of computation accelerators, which provide a 10.78x average speedup in computation-heavy microbenchmarks. Samoyed is 4.11x faster on average than a prior, non-JIT checkpointing alternative. Our main contributions are:

- A characterization of peripheral operations that are problematic with JIT checkpoints.
- A JIT checkpointing system that provides safe access to a wide variety of peripherals.
- A mechanism that dynamically scales peripheral operation cost to match energy availability.
- A static energy profiler for estimating the viability of peripherals before deployment.
- An evaluation comparing to prior work and showing that Samoyed provides safe peripheral operation where prior JIT systems fail, with a 4.11x speedup over non-JIT alternatives.

2 Background and Motivation

Improvements in the efficiency of computer systems and the maturation of energy-harvesting have led to the viability of batteryless computing devices built from commodity components. Intermittent energy availability complicates the development and execution of software targeting these devices. Managing the manipulation of hardware peripheral devices adds additional complexity, much of which has not been addressed by prior work on intermittent computing.

2.1 Energy-Harvesting Devices

A typical energy-harvesting device consists of an energy harvester, an energy buffering capacitor, a microcontroller (MCU), volatile and non-volatile memory, and peripheral devices such as sensors, architectural accelerators for specific computations, and radios [17, 39, 56, 72]. A device collects energy from, e.g., radio waves, light, or vibration, storing the energy in its energy buffer. When sufficient energy accumulates, the device operates, quickly draining its stored energy. On depletion, the device turns off and begins charging. Harvested power sources are often too weak to continuously power the device, which instead operates *intermittently*, rebooting frequently [17, 56, 72]. The frequency of reboots depends on the environment and the power consumption of the device and consecutive reboots may be separated by seconds or milliseconds. On each power failure, the device loses volatile state (registers, volatile memory, and peripheral state) and preserves non-volatile state (FRAM and Flash).

2.2 Just-In-Time Checkpointing

A Just-In-Time checkpointing system monitors a device's energy buffer and stores a checkpoint of volatile state in non-volatile memory immediately before a power failure. After checkpointing, the device sleeps [5, 6] or spin-waits [32] awaiting more energy, during which time its power may fail. When there is again sufficient energy, the device reboots, restores the checkpoint and continues from where it left off. Figure 1 shows code that encrypts a string `str` by copying it into `pText` and calling `AES_SW`, a software AES encryption routine, to produce ciphertext, `cText`. The execution

[1] **S**afe **A**tomic **M**anipulation **O**f **Y**our **E**xternal **D**evices

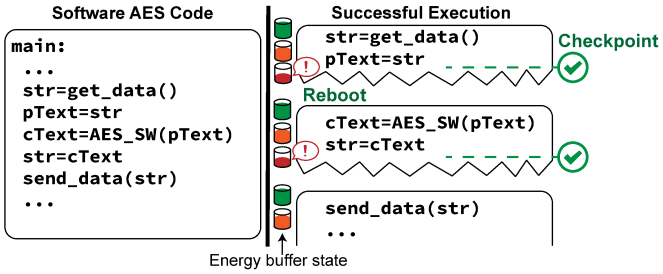

Figure 1. Execution of JIT checkpointing system. The code stores data in `str`, copies `str` into `pText`, encrypts `pText` into `cText`, copies `cText` back into `str`, and transmits (left). An example execution trace is shown (right).

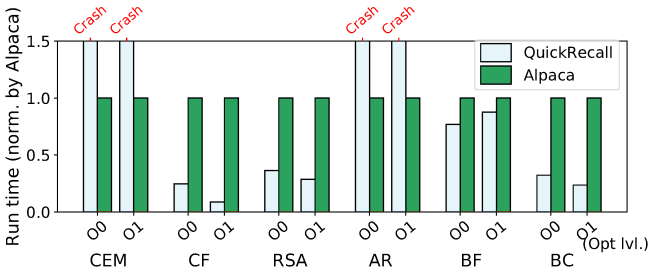

Figure 2. Performance benefit of JIT checkpointing. The plot shows data for optimization levels -O0 and -O1.

depletes the energy buffer forcing the device to checkpoint and continue later after a reboot.

An important correctness property of a JIT checkpointing system is that *the execution stops after the checkpoint.* A JIT checkpointing system eliminates the need to undo- or redo-log updates to non-volatile memory by not executing instructions that may manipulate non-volatile memory after checkpointing. In contrast, a system that collects a checkpoint and continues to execute may update non-volatile memory after the checkpoint and before a power failure. Consequently, such a system must manage non-volatile memory to ensure consistency with the checkpointed program state.

A JIT checkpointing system must monitor device energy, typically with a hardware voltage comparator on the energy storage capacitor [5–7, 32]. Assuming that main memory is non-volatile, which is consistent with common system designs [31, 32, 56, 61, 65], a JIT checkpointing need only contain register state, not stack or heap data. The size of a register-only checkpoint is fixed. Non-volatile main memory remains consistent with the checkpoint as long as execution pauses immediately after checkpointing. With small, fixed-size checkpoints, setting a JIT checkpointing system's voltage threshold is simple [32], and checkpointing has low overhead [32, 65]. Implementing JIT checkpointing checkpointing in a modern platform need not impose a high energy overhead. Using an on-chip comparator present in some microprocessors [61] results in only around 1%-7% energy increase in the microprocessor energy use (Section 4). Compared to the energy consumption of the entire board, the overhead is much lower.

2.3 The Performance Benefit of JIT Checkpointing

JIT checkpointing systems have several benefits compared to alternative approaches. First, a JIT checkpointing system checkpoints only once, immediately before a power failure, leading to low checkpointing overhead. Systems that operate oblivious to remaining energy may collect multiple checkpoints between power failures [15, 40, 44, 65], which is an unnecessary overhead. Second, a JIT checkpointing system need not use memory logging to maintain non-volatile memory consistency. In contrast, many alternative systems require memory logging for correctness incurring additional runtime overhead [4, 15, 40, 44, 45]. Third, a JIT checkpointing system can run mostly unmodified C code, while many alternative software approaches change the programming model [15, 30, 44]. Supporting mostly unmodified C code not only increases programmability but also has a possible performance benefit due to its compatibility with modern compiler optimizations.

Figure 2 emphasizes the performance benefit of a JIT checkpointing by comparing the execution time of QuickRecall [32], a JIT checkpointing system, with Alpaca [44], a state-of-the-art, non-JIT checkpointing system. We ran six benchmarks from the Alpaca paper [44]. We used the released code for Alpaca and implemented QuickRecall based on the author's description. **CEM** reads a temperature sensor and LZW-compresses it. **CF** stores and retrieves pseudo-random inputs in a cuckoo filter. **RSA** encrypts a string using RSA with a 64-bit key. **AR** reads an accelerometer and detects movement with a nearest-neighbor classifier. **BF** encrypts a string using Blowfish. **BC** counts set bits in an input stream. We ran each experiment on two different compiler optimization levels.

The result shows that a JIT checkpointing system, QuickRecall, is much faster than a non-JIT checkpointing system, Alpaca, in many computation-heavy benchmarks (CF, RSA, BF, BC). The speedup of QuickRecall indicates a large potential performance benefit in using a JIT checkpointing. However, the result also shows that a JIT checkpointing system sometimes fails (bars marked "crash") when executing a peripheral operation such as a temperature sensor reading (CEM) or an accelerometer reading (AR).

2.4 The Peripheral Problem for JIT Checkpoints

JIT checkpointing systems may fail in the presence of peripheral operations (i.e., Figure 2), which is an impediment to their adoption. Failures stem from the widespread assumption in JIT checkpointing systems that only register file state

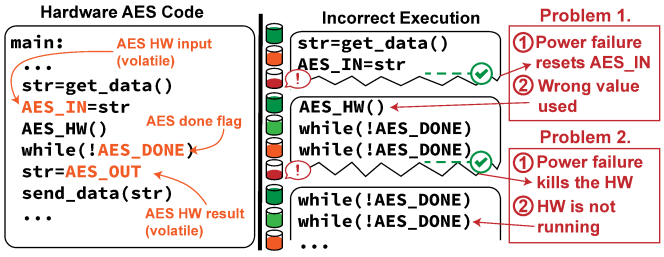

Figure 3. Incorrect peripheral execution. The code uses AES hardware, copying `str` into AES SRAM, `AES_IN`, computing, spin-waiting as AES runs, and copying results from AES SRAM, `AES_OUT`. The problematic execution at right clears the AES SRAM and leaves the module de-configured.

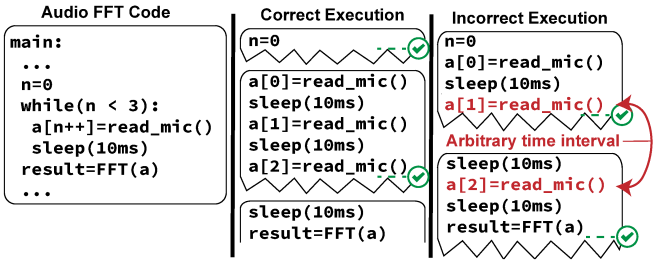

Figure 4. Inconsistent signal sampling. The code samples an audio signal and performs FFT. The middle is a correct execution and the right is a problematic one. A checkpoint and restart during sampling adds an arbitrary recharge time between some samples, rendering the FFT meaningless.

clears on power failure. However, a real system also loses *peripheral* state: sensor and radio configurations, intermediate result buffers in architectural accelerators, and state in the external environment. This paper focuses on *blocking, request-and-response* style peripherals: the MCU configures and activates a peripheral, later awaiting its response while halted, asleep, or spin-waiting. Arbitrary parallel execution and interrupt-driven concurrency are out of scope.

Figure 3 shows how a JIT checkpointing system like QuickRecall [32] behaves incorrectly due to peripherals. The example uses a hardware AES encryption module. The module requires software to populate a volatile input buffer (`AES_IN`), run AES (`AES_HW()`), wait for completion (`AES_DONE`), and collect output from a volatile buffer (`AES_OUT`).

Incorrect results. Unconstrained JIT checkpointing causes the program in Figure 3 to misbehave. Assume the system checkpoints after populating `AES_IN`. On reboot, the code activates the AES module with the volatile `AES_IN` buffer that was cleared by the power failure. The activation produces incorrect output.

Infinite wait. Simply making the AES module's memory non-volatile does not fix the problem. Assuming the module has non-volatile input and output buffers (existing modules do not), the code is still broken. If the system checkpoints while spin-waiting for AES to finish and then power fails, the loop never breaks because the deconfigured AES module never sets `AES_DONE`. The code spin-waits indefinitely.

Timeliness and signal consistency. Figure 4 illustrates another limitation of JIT checkpointing. The code samples an audio signal every 10 ms and computes its fast Fourier transform (FFT). The signal is rendered meaningless if the system checkpoints while sampling and power fails (i.e., the execution at right). Two samples that are consecutive in memory, such as `a[1]` and `a[2]`, should be separated by 10 milliseconds, but may be separated by the arbitrary duration required for the device to recharge. With such non-regular sample periods, the FFT result becomes meaningless. There is no way for existing JIT checkpointing systems to express this *signal consistency* condition because the code cannot distinguish between a buffer of coincidentally contiguous samples and a buffer of samples with inconsistent periods.

2.5 How Do Real Peripheral Accesses Fail?

We studied peripheral failures running microbenchmarks on Capybara [17], powered by radio waves (setup details are in Section 5.1), using on-board and on-chip peripherals. We implemented QuickRecall [32] as faithfully as possible based on the paper to show how JIT checkpointing systems do not handle peripherals. We have 15 tests in five categories.

Basic: *Sleep* waits on a hardware timer for 15ms.

Sensors: *Temp 5*, *Light 5*, and *Mic 5* read an on-chip temperature sensor, on-board light sensor, and on-board microphone, averaging five samples.

Hardware Accelerators: *DMA* uses on-chip Direct Memory Access (DMA) to copy memory. *AES* encrypts data with an on-chip AES accelerator. *LEA vadd* adds vectors with the on-chip TI LEA [64] DSP accelerator. *LEA matmult* multiplies matrices using LEA. *LEA FFT* does FFT with LEA. *LEA conv* low-pass filters data using convolution on LEA.

Communication: *PRINTF* prints data over USB using a hardware UART. *BLE TX* transmits Bluetooth Low-Energy (BLE) packets, using a BLE controller.

Mixed: *Temp BLE*, *Light BLE*, and *Mic BLE* transmit on BLE when the value read from a sensor exceeds a threshold.

Table 1 summarizes the failure behaviors we observed. An **infinite wait** happens when the system checkpoints while waiting for a peripheral operation to complete (e.g., Figure 3). An **incorrect result** happens when peripheral state — often managed through memory-mapped I/O registers (MMIO) — is volatile and erased on reboot. A **timeliness violation** happens when data must be collected or used adherent to a timing condition (like signal consistency).

Table 1 shows that many operations, including simple, common operations like *sleep*, fail with JIT checkpoints. The problem with existing JIT checkpointing is dire: spin-wait,

sleep, and MMIO manipulations all fail, precluding the use of crucially important serial protocols such as UART, I^2C, and SPI that support thousands of sensors, radios, and peripheral devices. This study evaluates our implementation of QuickRecall [32], but the results apply generally across JIT checkpointing and timer-based checkpointing systems [5, 6, 65] that may checkpoint at any program point.

Table 1. Observed failures for QuickRecall [32]. X indicates corresponding failure was observed.

category	app name	infinite wait	incorrect result	timeliness violation
Basic	Sleep	X		
Sensor	Temp 5	X	X	X
	Light 5	X	X	X
	Mic 5	X	X	X
HW Accel.	DMA	X	X	
	AES	X	X	
	LEA vadd	X	X	
	LEA matmult	X	X	
	LEA FFT	X	X	
	LEA conv	X	X	
Comm.	PRINTF	X		
	BLE TX	X	X	
Mixed	Temp BLE	X	X	X
	Light BLE	X	X	X
	Mic BLE	X	X	X

2.5.1 Why Not Just Disable Checkpoints?

JIT checkpointing failures happen when the system checkpoints at an inopportune moment; why not disable checkpoints during inopportune moments? *Simply temporarily disabling checkpoints is insufficient to solve the peripheral problem for a JIT checkpointing system.* There are two reasons: (i) write-after-read (WAR) dependences complicate memory consistency when power fails while checkpoints are disabled and non-volatile memory updates re-execute; and (ii) a limited-capacity energy buffer requires at least one checkpoint per execution period to avoid non-termination.

Memory consistency and WAR dependences Disabling JIT checkpoints during a region of code may compromise non-volatile memory consistency. If power fails with JIT checkpoints disabled, the system is forced to restore the last successful checkpoint. Control flow reverts to a previous point, but updates to non-volatile memory remain because non-volatile memory updates are not logged with the JIT checkpoint. On re-execution, the system may read such an updated non-volatile memory location, consuming an incorrect value. Existing JIT checkpointing systems thus strongly assume that execution stops after a checkpoint and control never flows backward on reboot. Temporarily disabling JIT checkpointing invalidates this key assumption and can lead to the invalid memory state.

The presence of peripheral accesses in code requires a JIT checkpointing system to ensure non-volatile memory consistency. Prior work [15, 31, 40, 44, 45, 65] observed that re-execution can leave non-volatile memory inconsistent. A re-executed code region leaves memory inconsistent if it contains a read followed by a write to the same non-volatile memory location (i.e., a WAR dependence), without a preceding write to the location. While previous JIT checkpointing avoids this WAR dependence problem by avoiding re-execution, peripheral accesses *require* re-execution, reintroducing the risk of memory inconsistency due to a re-executed WAR dependence.

Figure 5 shows how disabling JIT checkpoints renders the use of a hardware AES module incorrect. Checkpoints are disabled in the yellow region, preventing JIT checkpointing. On reboot, the system resumes at the most recently collected checkpoint, which is at the beginning of the yellow region, and re-executes the yellow region. Re-execution is problematic because the yellow region contains a *write* (`str<-AES_OUT`) after a *read* (`AES_IN<-str`) for a non-volatile memory location `str`, i.e., a WAR dependence. The execution re-encrypts the partially encrypted `str`, resulting in an incorrect `str`.

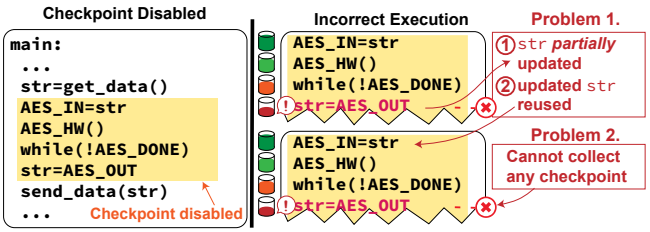

Figure 5. Incorrect execution with checkpoints disabled. AES code from Figure 3 with checkpoints partially disabled in the yellow region of the code (left). With JIT checkpoints disabled, part of the code re-executes on reboot, using a partially updated `str` again as an input. Moreover, the code cannot complete, leading to non-termination (right).

Non-termination Disabling JIT checkpoints during peripheral accesses can lead to non-termination if the energy required to operate for the duration of the peripheral access exceeds the total amount of energy that the device can buffer. In such a case, the region in which checkpoints are disabled will repeatedly attempt to execute, but will never complete.

Figure 5 also shows how disabling JIT checkpoints can lead to non-termination. In the figure, the yellow region is too long to execute using the device's buffered energy. The program repeats this partial execution indefinitely.

Addressing this non-termination problem is a challenge because the duration of the peripheral access can be highly workload- and input-dependent. For example, the AES module consumes different time and energy for different input strings and encryption key sizes, which can vary dynamically, and for different environmental conditions (e.g., temperature), which are outside of the system's control. Asking the programmer to re-write code to ensure termination requires reasoning about possible input sizes, per-operation

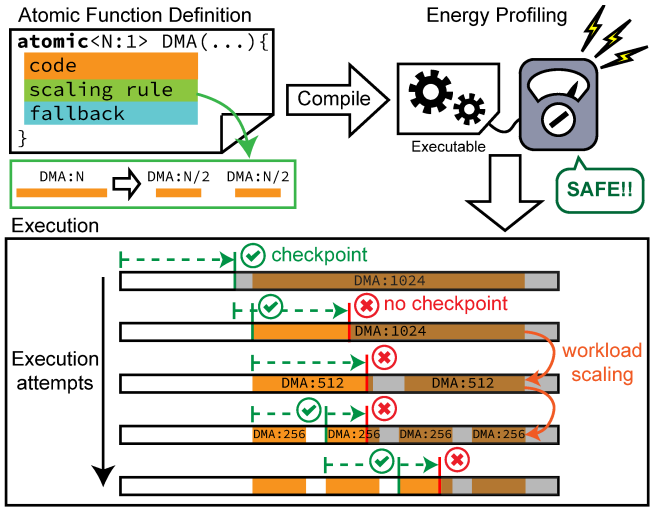

Figure 6. Overview of Samoyed. The figure shows an atomic function for manipulating a hardware direct memory access (DMA) module. After compilation and energy profiling, the code executes. Samoyed uses the atomic function's scaling rule to decrease the size of DMA inputs until the execution completes successfully.

energy cost, and operating conditions, which is extremely difficult. Precisely reasoning about the energy use of arbitrary code is a complicated, unsolved problem [12, 16, 38].

Existing JIT checkpointing systems fail to execute common peripheral operations correctly or efficiently, making them impractical despite their benefit in performance and programmability. In this work, Samoyed supports peripheral operations safely and efficiently on a JIT checkpointing system, making them viable for real-world deployment.

3 Safe Peripheral Operations with Samoyed

Motivated by the limitations of JIT checkpointing that we identified in Section 2.4, we propose a new JIT checkpointing system called Samoyed that supports safe and efficient peripheral operations, free of these limitations.

Samoyed allows the programmer to define an *atomic function*, inside which the system selectively captures checkpoints and maintains memory consistency with undo-logging. If the work in an atomic function requires more energy than a device can provide, Samoyed *dynamically scales* the atomic work by iteratively sub-dividing the function into multiple smaller function invocations, each containing a subsequence of the original function's work. Samoyed additionally provides an *energy profiler* to ensure that at least the maximally sub-divided atomic function (*i.e.,* minimum work) can run on a given platform, with a fallback routine to handle the profiling error.

Figure 6 summarizes Samoyed's operation. The programmer defines an atomic function containing peripheral operations, optionally providing a *scaling rule* and a *fallback*. Samoyed compiles the atomic function and runs the binary through the profiler, which estimates the energy consumption of each atomic function for a given device. The profiler determines whether the smallest sub-division of the atomic function will execute to completion, given the device's total available energy. If the profiler determines that the atomic function does not exceed the energy budget, Samoyed deems it safe and runs it, applying the scaling rule dynamically to adapt to energy availability as needed. If the profiled atomic function exceeds the device's energy budget, the programmer must change the platform or the peripheral. Profiling is imperfect and may underestimate an atomic function's energy cost. To handle profiling error, Samoyed allows a software fallback that executes if the smallest possible sub-division of an atomic function exceeds the device's energy budget on deployment. Unlike the atomic peripheral operation, the fallback safely spans failures. Samoyed's atomic function facility, compiler, profiler, and runtime system support safe, efficient peripheral access alongside the performance and programmability benefits of JIT checkpointing.

3.1 Baseline JIT Checkpointing System

Samoyed assumes an underlying JIT checkpointing system with fully non-volatile memory, as in prior work [32, 45, 65]. We assume the system can measure the device's remaining energy and interrupt the MCU when buffered energy hits a lower threshold (as in [5, 6, 32]). The system stops executing after checkpointing, restarting when the energy buffer is again refilled. We only target peripherals that do not have internal non-volatile state and are arbitrarily restartable after a power failure. Although restricted, many peripherals used in the embedded domain fall into this category, including all peripherals that we studied (Table 1). Samoyed is applicable only to blocking peripheral operations during which the MCU waits; Samoyed does not support concurrency of computation and peripheral accesses. While not inclusive of all code that manipulates peripherals, low-power embedded systems and peripheral computation accelerators frequently rely on a blocking interface. Samoyed is relevant to an important class of intermittent systems code.

3.2 Samoyed Atomic Functions

An atomic function contains code that should execute without checkpoints, permitting safe peripheral accesses.

3.2.1 Syntax

Table 2 summarizes the syntax of an atomic function. **atomic<$knob:$val,...>** declares a function as atomic. An atomic function has no return type because Samoyed requires passing outputs via output parameters. A programmer declares knob variables inside angle brackets (<>), which are

Table 2. Summary of atomic function syntax.

syntax	meaning
`atomic<$knob:$val, ...>`	define atomic functions with their knob variable
`input[]`	define a parameter as an input with its size
`output[]`	define a parameter as an output with its size
`inout[]`	define a parameter as both input and output
`scaling rule{}`	provide a scaling rule (if needed)
`fallback{}`	provide a fallback routine (if needed)

parameters Samoyed can vary to change the amount of work in an atomic function. Samoyed permits an optional knob minimum after the colon (`:`), defaulting to one.

The input, output, and inout parameter qualifiers declare that an atomic function's parameter is used for input, output, or both. An array parameter requires an additional size argument in a square brace (e.g., `input[size]`).

A scaling rule defines an atomic function's scaling rule, which tells Samoyed how to *recursively decompose* an atomic function that consumes too much energy into a sequence of smaller function invocations, the sequential execution of which is equivalent to the original function's execution.

A fallback is a software-only routine associated with an atomic function that Samoyed runs if recursive decomposition fails to produce a terminating execution. For a compute accelerator, the most common fallback is a software-only (*i.e.*, without peripherals) implementation that can run intermittently, tolerating arbitrarily-timed JIT checkpoints.

Samoyed treats knob variables, the scaling rule, and the software fallback as optional. An atomic function should not access any non-volatile global memory directly without specifying it as an `input`, `output`, or `inout`.

3.2.2 How to Write an Atomic Function

This section shows how to write an atomic function, using a hardware AES encryption accelerator as a running example. Figure 7 shows the original atomic function definition (left) and the simplified code transformed by Samoyed's compiler (right). The programmer annotates the function as `atomic` and marks its `inout` array parameter (Line 1). The function has a knob variable (`N`) with a minimum size of 16 (Line 1), which is the minimum allowed by the algorithm. The programmer then implements the peripheral control code (Line 2-5), the function's scaling rule (Line 7-9), and the software fallback (Line 11). In the example, the scaling rule is expressing that two consecutive AES invocations each operating on a distinct half of the original string are equivalent to the original AES invocation for this program. Expressing a recursive scaling rule is similar to writing a recursive function. The programmer should make sure that applying the scaling rule does not affect the correctness (e.g., Line 7-9 may not work with a different encryption algorithm).

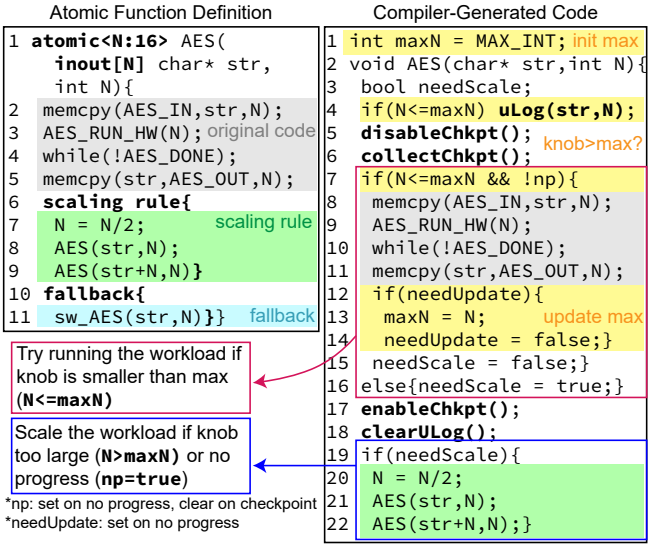

Figure 7. An atomic function. The hardware accelerated AES encryption code (left), and code transformed by the compiler (right).

3.2.3 How Samoyed Executes an Atomic Function

Samoyed executes compiled atomic function code, automatically invoking the scaling rule and fallback as necessary. The compiler-transformed code on the right of Figure 7 has two main parts. The code inside the upper red box (Line 7-16) contains the atomic function's *main workload*, including the original peripheral operations (gray, Line 8-11) and the state management code (yellow, Line 7, 12-14). The code in the lower blue box (Line 19-22) contains the scaling rule.

On entering the atomic function, Samoyed disables automatic JIT checkpoints (Line 5) and collects a checkpoint (Line 6), ensuring the atomic function restarts from the beginning when power fails with checkpoint disabled. Then, Samoyed runs either the atomic function's main workload (Line 7-16) or applies its scaling rule (Line 19-22), based on the prior execution history. The decision is made based on the two state variables, `maxN` and `np`. Each knob (e.g., `N`) has a corresponding *max knob value* (e.g., `maxN`) that holds the largest known value of the knob for which the function completed successfully. `np` is a flag that Samoyed sets if the most recent two attempts to execute the main workload made **n**o **p**rogress. `np` is cleared on a checkpoint. If the knob value is the same or smaller than the max knob value (`N<=maxN`) and the system is not currently stuck (*i.e.*, `!np`, Line 7), Samoyed tries running the main workload with checkpointing disabled. Otherwise, Samoyed re-enables checkpointing and applies the scaling rule, which decreases the knob value.

Running the main workload. If the knob value (`N`) is less than or equal to the max knob value (`maxN`) and `np` is unset, the main workload (Line 7-15) executes. Samoyed undo-logs `inout` parameters (Line 4) before the main workload *only if*

the main workload will execute (i.e., N<=maxN). If power fails during the main workload, Samoyed rolls back the changes made to the inout parameters (using its undo-log) and re-runs the operation from the last checkpoint (Line 6). If the main workload fails to complete twice, it means the current knob value is too large; Samoyed indicates that the current knob value is invalid by setting a needUpdate flag along with the np flag. On the following reboot with np set, Samoyed skips the main workload and instead runs the scaling rule by setting needScale (Line 16). On successful execution, Samoyed updates the max knob value (Line 12-14) if the previous value was invalid (i.e., needUpate is set) and exits the function. Updating the max knob value spares future executions the attempts and failures to calibrate the knob. Samoyed clears its undo-log after the peripheral operation is done (Line 18).

Scaling an atomic function. Samoyed runs the scaling rule in two cases: (1) after np is set, which indicates a failure to make progress on two consecutive runs; or (2) when the knob value is larger than the max value, which indicates that the current knob value is known to be too large based on prior executions (Line 7). Samoyed recursively invokes atomic functions with a decreased knob (Line 19-22) until each invocation is small enough to be executed on a given platform. If the scaling rule scales the knob beyond the specified minimum, the system runs the software fallback instead of the atomic function, whose code is omitted for brevity.

Concrete example invocation. An invocation of AES with a 1024-entry array (N=1024) will fail to execute if the system can only buffer enough energy to perform an AES encryption on 300 entries (N=300) at most. Initially, maxN is too high, at MAX_INT (e.g., 32,767 for 16-bit int). Samoyed compares the knob (1024) to maxN (MAX_INT) at Line 4, and undo-logs str. Samoyed then checks the np flag, which is unset, and attempts to execute the peripheral operations (Line 8-11). The knob value is too large and power fails before the operation completes. On failure, Samoyed rolls back the partially updated str. After two failed execution attempts, Samoyed sets np, indicating a lack of progress, and needUpdate, indicating the current maxN is incorrect. The next execution enters the else on Line 16 and the if on Line 19, decreasing N to 512 and recursively calling AES twice, each processing its half of the input (Line 20-22). N=512 remains too large; after two more failed attempts, Samoyed scales N to 256, which is small enough to complete with a given energy budget. After completing, the runtime updates maxN to 256 (Lines 12-14), equipping future executions with a viable starting knob value. This execution successfully hardware-accelerates an infeasibly large AES encryption call on an input of size 1024 by *automatically, dynamically* dividing it into *four* AES encryption calls, each with N=256. The knob converges to an amount of work possible using a full energy buffer, guaranteeing that amount of work even with no incoming energy.

3.2.4 Correctness

Samoyed avoids memory inconsistency. On a power failure during a peripheral operation, control reverts to the last checkpoint placed at the start of the atomic function (Line 6). On re-execution, memory may become inconsistent if the code has a WAR dependence that is not preceded by a re-initializing write (Section 2.5.1). Samoyed assumes that function local variables and peripheral-internal state clear on a restart and are initialized in the function before use, always having a re-initializing write. This assumption leaves inout parameters the only variables that can be involved in a problematic WAR dependence, since they are the only non-volatile data that may be both written and read in the atomic function. With all the inout parameters annotated correctly, Samoyed protects the parameters from becoming inconsistent with undo-logging (similarly to [4, 45]). Correctly annotating the inout parameters requires careful programming; the burden is, however, reasonable because most peripherals have well-defined inputs and outputs.

Samoyed also avoids non-termination. Samoyed can scale an atomic function's knob value down to a programmer-specified minimum. If the function can complete with at least the minimum knob value, Samoyed makes progress with scaling the knob. Samoyed's profiler helps determine whether an atomic function's minimum knob value is sufficiently small before deployment (Section 3.3). If the profiler mispredicts and the atomic function fails to complete even with its minimum knob value on deployment, Samoyed safely executes the software fallback instead to avoid non-termination. It is the programmer's responsibility to correctly define the knob and the scaling rule. The complexity of writing a recursive scaling rule is similar to writing a recursive function.

3.2.5 Generality

Samoyed supports sophisticated, multi-knob scaling rules, asking the programmer to define each knob and to define a scaling rule that uses reduced knob values. Figure 8 shows a scaling rule for matrix transpose using the LEA DSP module on the TI MSP430FR5994 microprocessor [63]. To use the module, the programmer populates the input buffer, TR_IN, and the module fills the output buffer, TR_OUT. The example's knobs are row and col. The scaling rule bisects the larger of the input matrix's row (Line 8-9) or column (Line 11-12) size. Illustrating Samoyed's capabilities, we implemented complex scaling rules, including a hardware accelerated FFT and a hardware accelerated matrix multiplication with three different knob values.

The scaling rule also need not be a recursive decomposition in general. Samoyed is flexible: a scaling rule could instead invoke a lower-precision (lower-energy) peripheral routine determined by the knob value.

```
1 atomic<row,col> tp(
    output[row*col] int* out,
    input[row*col] int* in,
    int row, int col){
2  memcpy(TR_IN,in,row*col*2);
3  TR_RUN_HW(row, col);
4  while(!TR_DONE);
5  memcpy(out,TR_OUT,row*col*2);
6  scaling rule{
7   if (row > col){
8    tp(out,in,row/2,col);
9    tp(out+row/2,in+(row/2)*NUM_COLS,row/2,col);
10  }else{
11   tp(out,in,row,col/2);
12   tp(out+(col/2)*NUM_ROWS,in+col/2,row,col);}}}
```

original code

column size of the original matrix

divide row dimension

row size of the original matrix

divide column dimension

Figure 8. A more complex scalable atomic region. A matrix transpose that dynamically scales the larger of its row or column dimensions.

3.3 Energy Profiler

Samoyed uses its energy profiler to directly measure the energy consumed by each atomic function to estimate the viability of the function on a given platform. The compiler generates a *measurement binary* that includes the original device initialization code (e.g., clock setup), and repeatedly invokes the atomic function with its smallest possible knob variable and a randomized input. The binary checks whether the most aggressively scaled function can run on a given platform. The generated measurement binary is programmed onto a real energy-harvesting platform, with the measurement hardware attached.

The measurement device is a custom hardware circuit similar in design to EDB [14]. The device manages a power source and a control signal to the target device. The hardware repeatedly (1) fills the device energy capacitor, (2) detaches the power source, and (3) signals the device to run the instrumented measurement binary. The signaled device repeatedly executes the atomic function until the energy is depleted and sends the resulting number of successful execution back to the measurement hardware. Aggregating across multiple runs, the programmer can look at the statistics to assess the viability of the function on a given platform. If the system cannot run a single instance of the function with a fully-charged capacitor, the programmer must increase the capacitor size or consider a peripheral with different energy requirements. If the system runs hundreds of peripheral operations in one energy cycle, the peripheral is likely appropriate for the device.

The purpose of Samoyed's energy profiler is *not* to guarantee correctness by proving the absence of non-termination. Samoyed avoids guarantees because peripheral energy consumption can vary unpredictably with different input, dynamic variation in hardware configuration, and the deployment environment (e.g., temperature, RF). Instead, the profiler is a tool to aid the programmer in understanding whether the peripheral's energy demand exceeds the device's energy supply in a particular environment, in a particular software, hardware, and input configuration.

Samoyed's profiler is practically useful. Most peripherals that we studied have low variance in their energy consumption. The profile uniformly reported a high margin between the peripheral's measured energy consumption and the device's buffer capacity (Section 5.6). This margin is an additional, empirical assurance that Samoyed will eventually find a knob assignment that successfully completes.

4 Implementation

We implemented Samoyed's annotations, compiler, runtime, and energy profiler targeting the Capybara platform [17].

Samoyed hardware. Capybara [17] includes a TI MSP430FR-5994 MCU, which has an AES accelerator, a LEA DSP accelerator [61], a Bluetooth Low-energy (BLE) radio controller, and sensors connected via I^2C. We used the MCU's internal comparator [61] to monitor the energy buffering capacitor and generate an interrupt at 1.8V, a voltage at which the system reliably checkpoints. According to the MCU's datasheet, the energy the comparator uses is 1-7% of the MCU energy [61].

Samoyed runtime library. Samoyed's runtime library includes checkpointing, undo-logging, restore, and dynamic scaling. Checkpointed register data and undo-logs reside in a pre-allocated FRAM buffer. To undo-log multiple elements in an array, Samoyed uses an atomic function using DMA. Registers restore on reboot, eventually restoring the program counter and redirecting control flow to the checkpointed program point. The system also restores any undo-logged data. The restore routine sets the `np` flag and the `needUpdate` flag, as explained in Section 3.2.3. All the control code inserted by the compiler is engineered to work correctly (e.g., respecting atomicity constraints) with arbitrary power failures.

The MSP430FR5994 MCU has both SRAM and FRAM as its main memory [61]. We wrote a custom linker script forcing the MCU to use only the FRAM during program execution.

Samoyed compiler. We implemented the Samoyed compiler as a source-to-source transformation using LibTooling [60], a Clang-based frontend that analyzes the abstract syntax tree (AST) and generates instrumented C code that we compile with GCC. Source-to-source transformation avoids inefficiencies in the experimental [4] MSP430 LLVM backend in favor of more efficient GCC backend.

Samoyed energy profiler. We used another MSP430FR5994 MCU that controls a SIP32431 load switch to measure energy, collecting data via UART.

5 Evaluation

We evaluated Samoyed, showing that it enables correct peripheral operations, comparing to QuickRecall [32], a prior system that does not. We show that Samoyed allows the use

of hardware accelerators that substantially improve performance. Comparing computational performance to an alternative system, we show that Samoyed maintains the extremely low run time overheads of JIT checkpointing systems. We also show that the energy profiler simplifies programming. The evaluation also studies a full-system, room-scale sensing prototype that senses audio, performs DSP and statistical inference computations, and communicates using BLE. Finally, we show an analytical study of the applicability of Samoyed to Eyeriss [10] a recent CNN accelerator.

5.1 Experimental Setup

We evaluated Samoyed using a real, full hardware energy-harvesting setup. We attached a dipole antenna and a P2110-EVB Powercast harvester [51] to Capybara [17], harvesting 915 MHz radio waves generated by a ThingMagic Astra-EX RFID reader positioned 75cm apart and set to a power level of 30dBm. We ran benchmarks repeatedly until the confidence interval was under 10% of the mean.

5.2 Samoyed Makes Peripheral Accesses Correct

We compared Samoyed's correctness to a previous JIT checkpointing system, QuickRecall [32], with the benchmarks from Section 2.5. We implemented QuickRecall based on the authors' description. While some details may differ, QuickRecall's correctness issues arise from the high-level system design, not its implementation. Table 3 reports data showing Samoyed avoids all failures that QuickRecall experienced.

Table 3. Correctness of Samoyed vs. QuickRecall. O indicates success, X indicates failure.

category	app name	Samoyed	QuickRecall [32]
Basic	Sleep	O	X
Sensor	Temp 5	O	X
	Light 5	O	X
	Mic 5	O	X
HW Accel.	DMA	O	X
	AES	O	X
	LEA vadd	O	X
	LEA matmult	O	X
	LEA FFT	O	X
	LEA conv	O	X
Comm.	PRINTF	O	X
	BLE TX	O	X
Mixed	Temp BLE	O	X
	Light BLE	O	X
	Mic BLE	O	X

We also empirically observed that peripheral failures are not unique to JIT checkpointing. Many recent non-JIT checkpointing systems, e.g., Ratchet [65], Alpaca [44], and Chinchilla [45], also experienced failures while using certain peripherals: Ratchet [65] experienced failures similar to QuickRecall's when the compiler inserted checkpoints during regions of code that manipulate peripherals. Ratchet [65], Alpaca [44], and Chinchilla [45] all failed in protecting the WAR dependences introduced by peripherals, corrupting memory (mainly because their compilers are unaware of the peripheral manipulations). We do not discuss the failure of these systems in detail, as each is highly system-specific.

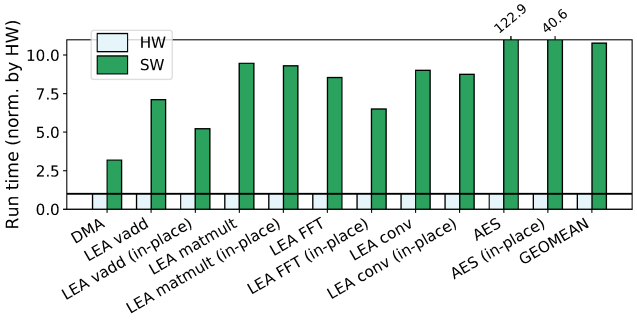

Figure 9. Comparing accelerators vs. software.

5.3 Samoyed Enables Hardware-Acceleration

Unlike prior JIT checkpointing systems, Samoyed enables the use of extremely efficient architectural accelerators that substantially improve performance. To emphasize the benefit, we compared six programs written to use a hardware accelerator with a behaviorally identical software implementation. **DMA** copies 50,000 memory words. **LEA vadd** adds two 600-entry vectors using TI's on-chip LEA DSP accelerator. **LEA matmult** multiplies two 16-by-16 fixed-point matrices. **LEA FFT** computes the fast Fourier transform (FFT) on a 1024-entry array. **LEA conv** convolves a 16-entry filter with a 2048-entry input. **AES** encrypts a 2048-character string with a 128-bit key, using an on-chip AES accelerator. For each benchmark except DMA, we ran two different versions, one with a separate input and output buffer and one that reuses the input buffer to store the result (labeled "in-place"). All software implementations replaced the invocation of accelerators with software code from official TI libraries [62, 63].

Figure 9 shows that even compared to highly-optimized vendor libraries, both accelerated versions yield dominant performance, with speedups up to 122.9x and on average 10.78x. The in-place versions with undo-logging remain faster than their software counterparts.

5.4 Samoyed has Low Overheads

We compared Samoyed's performance against Alpaca's, as we did in Section 2.3 with QuickRecall. We used the same benchmarks as in Section 2.3. Figure 10 shows that Samoyed is faster than Alpaca in all cases, with up to a 13.9x speedup. Samoyed's mean speedup over Alpaca is 2.74x with compiler optimization level `-O0` and 4.11x with `-O1`.

Samoyed is faster than Alpaca for three reasons. First, Alpaca executes multiple task boundaries (i.e., checkpoints) per execution period because tasks are often conservatively defined. Samoyed collects exactly one checkpoint per power failure. Second, Alpaca uses redo-logging to prevent WAR dependences from corrupting memory throughout the code.

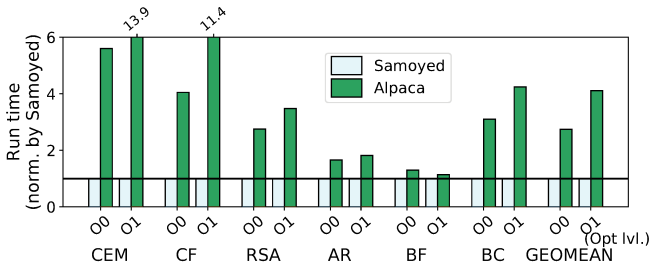

Figure 10. Performance of Samoyed and Alpaca. The plot shows data for gcc optimization levels -O0 and -O1.

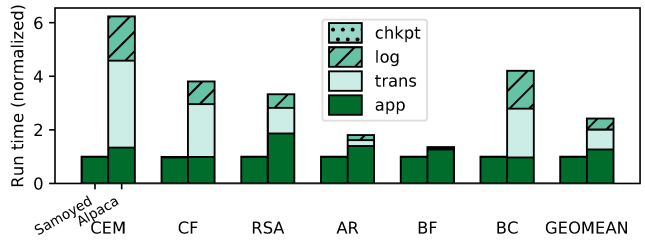

Figure 11. Comparing the overheads of Samoyed and Alpaca. Execution time breakdown for each system compiled with -O0. Alpaca's main overhead is task transitions (trans) and redo-logging (log). Samoyed's main overhead is checkpointing and restoring (chkpt), which is less than 0.01% on average.

Samoyed only incurs undo-log overhead when an atomic function with `inouts` is present. Third, Alpaca's tasks limit the scope of compiler optimizations, while Samoyed permits optimization across all code outside an atomic function.

Figure 11 breaks down each system's execution time, revealing its major overheads. We measured each time overhead by toggling a GPIO before and after each type of operation. In addition to application code (app), Alpaca adds overhead for each task boundary (trans) and for redo-logging (log). Samoyed's main overhead is checkpointing and restoring (chkpt). Our benchmarks did not contain any `inouts`, incurring no undo-log overhead.

The data make clear why Samoyed is faster than Alpaca. Relative to application code execution time, Alpaca has 59% task boundary overhead and a 32% redo-logging overhead. Samoyed's checkpoint and restore overhead is a vanishingly small 0.9% of its application code execution time. The result shows that Samoyed still preserves the low-overhead characteristics of JIT checkpointing, while correctly running the benchmarks for which QuickRecall failed (CEM, AR). Other recent non-JIT checkpointing systems, Ratchet [65] and Chinchilla [45], perform similarly to Alpaca for these benchmarks [45]. While we do not directly compare to those systems, Samoyed is likely to similarly outperform them.

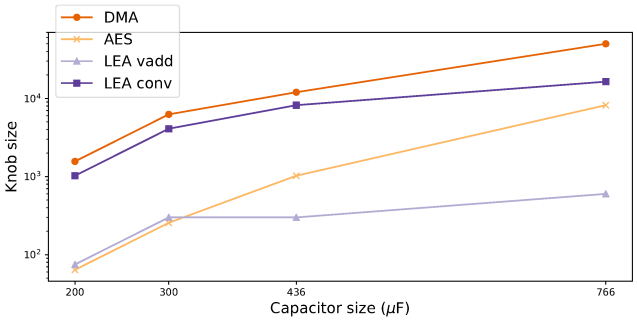

Figure 12. Size of a knob for various benchmarks in different capacitor size.

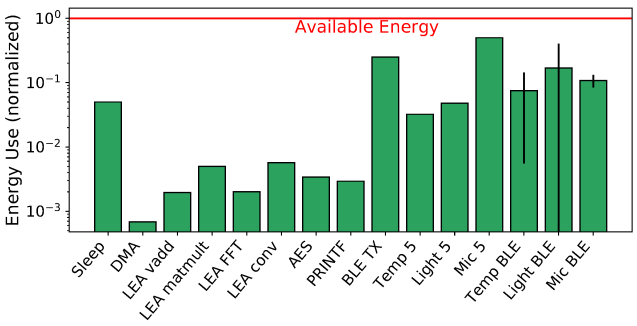

Figure 13. Profiled energy use of each atomic function.

5.5 Samoyed Effectively Scales Atomic Functions

Samoyed effectively scales knob values in atomic functions. Figure 12 shows how knob values vary with different capacitor sizes. From our seven benchmarks that have a scaling rule (DMA, AES, LEA vadd, LEA matmult, LEA conv, LEA FFT, BLE TX), we plot only four because LEA matmult has three knob values, LEA FFT only fails with a very small capacitor, and BLE TX requires a very large capacitor. The data show that all benchmarks dynamically scale a region's work to match the available capacitor size. Samoyed's scaling rules free the programmer from thinking about the capacitor size when programming.

5.6 Samoyed's Energy Profiler is Informative

Figure 13 shows the energy use of each atomic function for the benchmarks in Table 3 *with the smallest knob size*, automatically measured by our energy profiler. The profiler collected 30 sample executions each and plotted the mean and the standard deviation. The atomic functions between DMA and PRINTF on the x-axis use less than 1% of the available energy, with very little variation. Similarly, Sleep and the atomic functions between BLE TX and Mic 5 used around 1–10% of the energy. It is likely that these functions will safely make progress with an appropriate Samoyed-selected knob value.

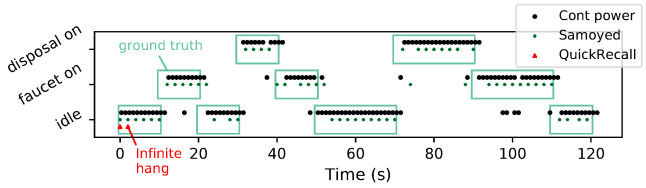

Figure 14. Data received from the sensor. The classified state of the kitchen received at the server is plotted in black (continuously-powered sensor), dark green (Samoyed, harvested energy) and red (QuickRecall [32], harvested energy). The light green box shows the ground truth.

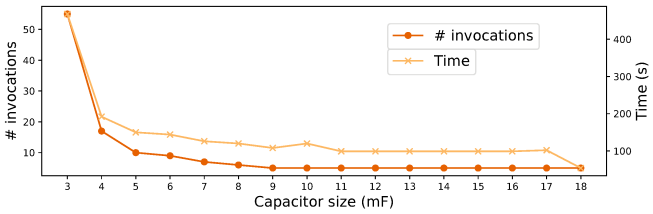

Figure 15. Estimated Eyeriss behavior on Samoyed. Estimated number on invocations of the atomic functions and the end-to-end execution time is plotted.

Other benchmarks (Temp BLE through Mic BLE) have high variation in their energy consumption because their atomic functions contain statically unpredictable control flow. In such cases, the programmer may have to manually reason that the function is safe to deploy.

Although the profiler cannot provide a progress guarantee, its output helps determine whether the atomic function is likely to fail on the profiled platform, simplifying both programming and platform design.

5.7 Case Study 1: Intelligent Synthetic Sensor

We demonstrated the practicality of Samoyed, using it to build a room-scale, end-to-end application prototype: an energy-harvesting version of the *Synthetic Sensor* proposed by Laput, et al [36]. A synthetic sensor uses machine learning on sensor inputs to classify environmental signals. Our prototype collects microphone data from a fixed location. The device uses a pre-trained logistic regression classifier to detect kitchen events (faucet on, garbage disposal on, quiet) and sends results to a server via BLE. The application collects a window of 64 microphone samples at 167Hz, converts them to 16-bit fixed-point, performs a LEA FFT, processes FFT results, classifies using a LEA matrix multiply, and sends results over BLE. Sensing, BLE transmission, FFT, and matrix multiplication were each written as an atomic function.

Figure 14 shows classifications received via BLE from three system variants. Black points show continuously-powered sensor, dark green show Samoyed, and red show QuickRecall [32], with Samoyed and QuickRecall running on harvested energy. The light green box shows the ground truth. The data show that Samoyed accurately reports events approximately once every two seconds which is acceptable, although less frequently than with continuous power. In contrast, QuickRecall fails to send data after two samples. Assuming a reporting frequency of two seconds is acceptable, Samoyed's prediction rate is 87%, including as errors prediction error, packet loss, and late packet delivery. This study shows Samoyed supports end-to-end applications that can be applied to a real-world context.

5.8 Case Study 2: Deep Neural Network Accelerator

We analytical studied the applicability of Samoyed to Eyeriss [10], a recent convolutional neural network (CNN) accelerator. Eyeriss reports power consumption of 200mW [10], which is a feasible power level for a burst-mode [17] intermittent system.

Without access to a real Eyeriss chip, we estimated the behavior of Samoyed for Eyeriss performing a prediction using AlexNet [35], by borrowing reported measurement numbers from the Eyeriss paper [11]. We assume that main memory is FRAM instead of DRAM as in the original design, and concentrate on the five convolutional layers, which consume most of the total energy [10]. We use the reported power and the end-to-end latency number to calculate the chip energy use. Then, we multiply the DRAM access numbers given by the paper [11] with an FRAM access energy of 0.4nJ per byte [53] to get the FRAM access energy. If the sum of the two exceeds the device's energy capacity, Samoyed will have to recursively break the convolution layer down into multiple smaller convolutions. For each application of Samoyed's scaling rule, we add the cost of reconfiguring Eyeriss (provided by the paper) including the additional FRAM access cost. We conservatively estimate the extra FRAM access cost due to Samoyed scaling by multiplying the original FRAM access cost by the number of recursive division. We measured the charging time for each capacitor size from 3mF to 18mF in the same setup as in Section 5.1 and used it to estimate the end-to-end execution time.

Figure 15 plots the estimated number of atomic function invocations (i.e., the number of sub-divisions) and the estimated execution time. The result suggests that Samoyed's atomic function specification and scaling rule facility can be applicable to Eyeriss. We did not model a software fallback (it was unnecessary), but recent work [23] proposed a software CNN implementation that is applicable if Eyeriss exceeds the device energy budget. Samoyed makes using Eyeriss in an intermittent system easier. The scaling rule effectively divides the entire network computation into many, smaller atomic functions to run on a given platform, which is not straightforward to do manually. The data show that execution time

decreases as available energy increases because decomposing the peripheral invocation imposes a configuration and FRAM overhead. With an 18mF capacitor, the end-to-end execution time is less than a minute, suggesting that Samoyed and Eyeriss together make it viable to run AlexNet on an energy-harvesting device. This study emphasizes Samoyed's applicability to modern architectural accelerators, establishing the feasibility of their use in intermittent systems.

6 Related Work

Several prior research relates to Samoyed. We discuss the most related work on energy-harvesting devices and intermittent computing, idempotent computation, and work on memory consistency and persistency.

Intermittent and energy-harvesting devices Section 1 covered prior JIT checkpointing systems [5, 6, 32, 33] and static checkpoint systems relying on a compiler [4, 7, 16, 45, 46, 54, 65], or the programmer [15, 30, 40, 44, 70]. Other approaches [31, 43] change the microarchitecture to efficiently provide checkpointing, suffering some problems of software approaches, and requiring custom silicon.

Concurrently with our work, RESTOP [55] developed support for automatic reinitialization after power failures of peripherals using MMIO registers or I^2C protocol. RESTOP is a partial solution for simple peripherals that require only reinitialization (e.g., simple sensors), but does not support complex peripherals, like architectural accelerators (e.g., LEA, AES). RESTOP also does not provide timeliness or workload scaling. Mayfly [30] uses an external hardware timer to avoid timeliness violations. Mayfly is a complementary work that specifically addresses timeliness violations, but does not handle other peripheral manipulation issues. NVRF [68] is an RF chip that saves configurations in non-volatile memory. NVRF is not a general peripheral solution and requires custom hardware. UFoP [28] allocates to each peripheral a capacitor that stores sufficient energy to use that peripheral. The solution is insufficient when the energy use of the peripheral varies. In contrast, Samoyed's scaling rules, energy profiling, and fallback routines prevents non-termination for a broad class of peripherals.

Samoyed can be applied to a variety of intermittent platforms [17, 29, 56, 72], and multi-tenant energy-harvesting systems [2]. Wisent [59] and Stork [1] enable wireless software updates for these devices, while Ekho [73] and EDB [14] help with full-system debugging. Incidental computing [42] optimizes latency-insensitive code via approximation and NEOFog [41] models communication between hypothetical intermittent devices that can directly communicate. Other prior work targets energy-harvesting devices but does not support intermittent operation [8, 34, 37, 58].

Idempotent compilation Prior work on idempotent code compilation [19, 20, 74] uses a compiler to generate idempotent code, making the system robust to failures. Idempotent compilation also features in other intermittent systems [44, 45, 65]. Our prototype ensures idempotent execution for atomic functions by undo-logging `inout` parameters.

Automatic algorithm tuning Prior work on algorithm tuning [3, 9, 52, 69] dynamically selects the optimal algorithm and execution parameters, optimizing performance similarly to Samoyed's dynamic knob selection.

Consistency and non-volatile memory Prior work on transactions [24, 26, 27, 57] bears similarity to Samoyed's atomic regions, although targeting concurrency, not intermittence. Work on non-volatile memory persistency [13, 18, 21, 22, 47–50, 66, 67, 75] also relate to Samoyed in their purpose, but differ in their mechanism and in that they target large-scale parallel systems, not intermittent microcontrollers.

7 Conclusion

This paper presented Samoyed, a Just-In-Time checkpointing system that can safely and efficiently manipulate peripherals in programmer-defined atomic functions, leveraging compiler support, energy profiling, and a runtime library for safe, efficient intermittent execution. Samoyed safely executes code that manipulates peripherals that cause failures in prior JIT checkpointing systems. Samoyed provides a 10.78x mean speedup over prior work by enabling the use of architectural accelerators. Samoyed preserves the performance benefit of JIT checkpointing systems, with a 4.11x speedup on average compared to an alternative non-JIT system. Samoyed dynamically scales atomic function work that is too large, profiles energy consumption, and defaults to a software fallback if safe use of a peripheral is impossible. Together, these features of Samoyed significantly simplify the task of writing code for an intermittent system that manipulates peripherals. Using Samoyed, we built and evaluated an end-to-end home sensing application that we tested in a deployment, demonstrating Samoyed's practicality. We also demonstrated quantitatively that Samoyed is crucial to the use of emerging architectural accelerators in intermittent devices.

Acknowledgments

We thank the anonymous reviewers for the valuable feedback and we thank Jennifer Sartor for shepherding our work. We thank Emily Ruppel for the insightful early discussion on these ideas. This work was supported in part by National Science Foundation Award #1751029. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Kiwan Maeng was partially supported by the Korea Foundation for Advanced Studies (KFAS).

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. Helix-up: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 235–245.

[10] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[12] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Determining application-specific peak power and energy requirements for ultra-low power processors. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 3–16.

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

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

[19] Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–12.

[20] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2254064.2254120

[21] Kshitij Doshi and Peter Varman. 2012. WrAP: Managing byte-addressable persistent memory. In *Memory Archiecture and Organization Workshop.(MeAOW)*.

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

[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] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, Vol. 32. IEEE Computer Society, 102.

[25] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 243–254.

[26] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[27] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free synchronization. In *Proc. of the 20th Annual International Symposium on Computer Architecture*. 289–300.

[28] 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

[29] 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.

[30] 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.

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

[32] 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.

[33] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Trans. Embed. Comput. Syst.* 16, 3, Article 65 (April 2017), 23 pages. https://doi.org/10.1145/2983628

[34] 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

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[36] Gierad Laput, Yang Zhang, and Chris Harrison. 2017. Synthetic sensors: Towards general-purpose sensing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 3986–3999.

[37] 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

[38] Umer Liqat, Zorana Bankovic, Pedro Lopez-Garcia, and Manuel V Hermenegildo. 2016. Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)*.

[39] 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.

[40] 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

[41] 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.

[42] 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

[43] 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.

[44] 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.

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

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

[47] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 1.

[48] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2150976.2151018

[49] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276.

[50] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131.

[51] Powercast Inc. 2010. Evaluation Board for P2110 Powerharvester™ Receiver. http://https://datasheet.octopart.com/P2110-EVB-Powercast-datasheet-15540333.pdf. , 3 pages.

[52] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[53] Masood Qazi, Michael Clinton, Steven Bartling, and Anantha P Chandrakasan. 2012. A low-voltage 1 Mb FRAM in 0.13 μm CMOS featuring time-to-digital sensing for expanded operating margin. *IEEE Journal of Solid State Circuits* 47, 1 (2012), 141.

[54] 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

[55] 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.

[56] 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.

[57] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

[58] 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

[59] 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.

[60] The Clang Team. 2018. Clang 7 documentationg: LibTooling. https://clang.llvm.org/docs/LibTooling.html.

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

[62] TI Inc. 2018. Advanced Encryption Standard. http://www.ti.com/tool/AES-128.

[63] TI Inc. 2018. Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. http://www.ti.com/tool/msp-dsplib.

[64] TI Inc. 2018. Low-Energy Accelerator (LEA) Frequently Asked Questions (FAQ). http://www.ti.com/lit/an/slaa720/slaa720.pdf. , 8 pages.

[65] 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.

[66] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5.

[67] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[68] Zhibo Wang, Fang Su, Yiqun Wang, Zewei Li, Xueqing Li, Ryuji Yoshimura, Takashi Naiki, Takashi Tsuwa, Takahiko Saito, Zhongjun Wang, et al. 2017. A 130nm FeRAM-based parallel recovery nonvolatile SoC for normally-OFF operations with 3.9× faster running speed and 11× higher energy efficiency using fast power-on detection and nonvolatile radio controller. In *VLSI Circuits, 2017 Symposium on*. IEEE, C336–C337.

[69] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE, 38–38.

[70] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors *(SenSys '18)*.

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

[72] 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).

[73] 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

[74] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. 2013. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2451116.2451129

[75] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744