A Difference World: High-performance, NVM-invariant, Software-only Intermittent Computation

Harrison Williams[*]
Virginia Tech

Saim Ahmad[*†]
Amazon

Matthew Hicks
Virginia Tech

Abstract

Supporting long life, high performance, intermittent computation is an essential challenge in allowing energy harvesting devices to fulfill the vision of smart dust. Intermittent computation is the extension of long-running computation across the frequent, unexpected, power cycles that result from replacing batteries with harvested energy. The most promising intermittent computation support strategies combine programmer direction and compiler analysis to minimize run-time overhead and provide programmer control—without specialized hardware support. While such strategies succeed in reducing the size of non-volatile memory writes due to checkpointing, they must checkpoint continuously. Unfortunately, for Flash-based devices (by far the most ubiquitous), writing checkpoints is slow and gradually kills the device. Without intervention, Flash devices and software-only intermittent computation are fundamentally incompatible.

To enable ubiquitous programmer-guided intermittent computation we design and implement CAMEL. The key idea behind CAMEL is the systematic bifurcation of program state into two "worlds" of differing volatility. Programmers compose intermittent programs by stitching together atomic units of computation called tasks. The CAMEL compiler ensures that all within-task data is placed in the volatile world and all between-task data is placed in the non-volatile world. Between tasks, CAMEL swaps the worlds, atomically locking-in the forward progress of the preceding task. In preparation for the next task, CAMEL resolves differences in world view by copying only differences due to the preceding task's updates. This systematic decomposition into a mixed-volatility memory allows—for the first time—long-life and high performance programmer-guided intermittent computation on Flash devices: CAMEL outperforms the state-of-the-art checkpointing system for Flash-based devices by up to 5x while eliminating the need for hardware support, and enables reliable execution of atomic operations where the state-of-the-art fails. Beyond Flash, CAMEL's differential buffer system improves performance by a factor of 2x compared to existing task-based approaches on FRAM platforms.

1 Introduction

Energy harvesting [34, 43, 54] is the key to realizing the vision of ubiquitous computing [22, 48]. Aggressive transistor scaling brings us to an inflection point: computing devices are smaller than a grain of rice and operate on nano-watts of power [53], but the batteries required to power them remain largely unchanged, leaving them large, heavy, expensive, and sometimes flammable [4]. This asymmetric scaling means attaining ubiquity demands that current and future devices shed batteries in favor of harvested energy.

The transition to harvested energy brings a new challenge: how can we support long-running programs in the face of the frequent, unpredictable, power cycles brought on by the relative trickle of energy supplied by energy harvesting? Existing programs and programmers alike assume a continuous supply of energy, while energy harvesters provide only enough energy for short bursts of computation. Attempting to execute unmodified programs on such short bursts of power dooms long-running programs to a never-ending series of restarts. A naive application of existing checkpointing schemes [41] is inadequate as previous work shows that, without careful attention to the memory durability ramifications of power cycles, semantically incorrect executions occur [40]. Thus intermittent computation is born.

There are two classes of intermittent computation:

Just-in-time checkpointing: special-purpose hardware monitors available power, committing a checkpoint and ceasing computation when power dips below a pre-defined voltage threshold [2, 3, 21, 41, 49]. Recent work shows that even on-chip[1] voltage monitors steal between 24% and 71% of a system's energy when using a comparator and an ADC for

[*]Equal contribution.
[†]Work completed at Virginia Tech.

[1]Often on-chip voltage monitoring hardware is subscribed to software functionality and not available for just-in-time voltage monitoring.

voltage monitoring, respectively [51].

Continuous checkpointing: a program is decomposed (by a programmer [6, 26, 29], compiler [30, 52], or hardware [11, 27, 28, 45]) into a series of inherently-restartable sub-computations, glued together by checkpoints. This results in power-failure-agnostic program execution.

Programmer-guided continuous checkpointing intermittent computation systems are favored when programmers require guarantees about execution [6, 26, 29]. Such guarantees are common when using external devices, which pose challenges for just-in-time checkpointing systems: peripheral operations are often atomic (e.g., UART or I^2C transmissions) and, when interrupted by power loss, require that (1) execution is rolled back to the beginning of the operation and (2) the relevant peripheral be reconfigured for the section of code to be re-executed. However, unrestricted state rollback often causes semantically incorrect execution [40], while forcing the programmer to anticipate the effects of a power loss at every point in the code and write reconfiguration routines to mitigate the consequences is unscalable and error-prone.

Programmer-guided approaches relieve programmers of this burden through a combination of compiler analysis and a C programming interface that exposes forward progress atomicity as a first-class programming abstraction. Compiler analysis is comprehensive and error-free, while the programming interface allows programmers to reason about the system-level effects of computation—at a granularity that they are comfortable with or that mirrors the device's interface/protocol. Such approaches divide programs into a series of checkpoint-connected tasks representing atomic, restartable units of computation, i.e., they either complete entirely or not at all. Regardless of power cycles, the result of task execution is semantically consistent with the code. The fundamental principle is that tasks keep their changes private until they complete. Early work conservatively versions all within-task data [26], while follow-on work introduces new classes of cross-task communication channels [6]. The most recent approach further reduces overhead using idempotence analysis to minimize data copying [29].

This paper addresses two limitations in state-of-the-art programmer-guided intermittent computation approaches:

Flash device lifetime: the frequent non-volatile memory writes of continuous checkpointing strategies—no matter the size—exhaust Flash's limited write endurance [16].

Performance: current approaches copy redundant data to within-task buffers due to not reusing existing data.

Fixing the first flaw **makes programmer-guided intermittent computation possible and performant** on Flash-based devices. Fixing the second flaw **increases performance across** all devices.

We design and implement CAMEL, an extension to C and compiler support that enables long-life, low-overhead intermittent computation on Flash-based systems—without hardware support. Our solution leverages the idea of two worlds from ARM TrustZone [35], but replaces security with data non-volatility. Two worlds exist within a CAMEL-instrumented program: a non-volatile inter-task world (for recovery) and a volatile intra-task world. We implement this selective mixed-volatility world abstraction on Flash devices using the time-dependent non-volatility of SRAM (the main memory on Flash-based devices) to create a mixed-volatility domain in SRAM. Unlike previous work which checkpoints all of SRAM, rendering it effectively wholly-non-volatile [49] (§5 shows that this is an untenable solution from a performance standpoint), CAMEL strategically creates both volatile and non-volatile regions within SRAM. Placing within-task data in the volatile world allows partial task execution to be safely forgotten across power cycles. Placing across-task data in the non-volatile worlds allows for task-level, atomic, forward progress. The challenge that CAMEL addresses is the correct and efficient transfer of data between the volatile and non-volatile worlds when power loss can occur at any time. CAMEL's solution of fine-grain idempotence analysis coupled with differential state analysis allows CAMEL to efficiently update and transition between worlds.[2]

We validate CAMEL's ability to extend long-running computation across frequent power cycles using popular microcontrollers in energy harvesting deployments and a superset of benchmarks from previous work. Experiments on hardware under realistic energy harvesting conditions show that CAMEL reduces average run-time overhead by 3–5x over the state-of-the-art Flash-based intermittent system and enables correct execution of atomic operations where prior work fails. On the FRAM-based devices earlier programmer-directed intermittent systems target, CAMEL's advanced compiler analyses cut run-time overhead in half compared to the state-of-the-art.

This paper makes the following contributions:

1. We demonstrate that existing continuous checkpointing approaches have poor performance and kill Flash due to checkpoint-induced Flash memory writes/erases (§2.1).
2. We propose the notion of controlled-volatility worlds to enable high-performance programmer-guided intermittent computation on Flash devices (§3.4).
3. We present an NVM-invariant performance improvement: reusable differential buffers (§3.5).
4. We expose to the designer and quantify the tradeoff between pre-deployment effort and run-time overhead (i.e., canary vs. CRC) (§3.2).
5. We evaluate CAMEL against state-of-the-art just-in-time

[2] Note that swapping data from the volatile world to the non-volatile world is different than double buffering seen in other intermittent computation systems. In double buffering, all software state is written to non-volatile memory to lock-in forward progress. The old checkpoint (and its buffer) is maintained to provide a valid recovery option all the way until the last bit of the current state is written to the new checkpoint [52]. No buffer is modified outside of the checkpoint commit process. In contrast, CAMEL's volatile world acts as a scratch-pad for all state changes of the current task and the non-volatile world acts as a valid recovery option. Idempotence analysis allows CAMEL to avoid double buffering the non-volatile world, reducing memory overhead and increasing performance.

and programmer-guided intermittent systems; results show that CAMEL outperforms previous approaches in lifetime, functionality, and performance—regardless of NVM (§5).

6. We open-source our design and evaluation artifacts to enable further research and deployment of CAMEL on current systems.

2 Motivation

There exists a succession of programmer-guided intermittent computation systems, each refining the interface exposed to programmers and reducing run-time overhead. Why is another approach needed? This section answers this question through analysis that shows that **by ignoring Flash-based energy-harvesting platforms, we exclude the most ubiquitous, most available, lowest cost, and highest performance systems from the benefits of intermittent computation**. Experiments with existing approaches show that due to the performance and lifetime consequences of Flash writes/erases and the high-frequency checkpoints endemic to continuous checkpointing, a new approach is required. Lastly, we show that SRAM retains state for many minutes during the off periods common to energy harvesting devices. This motivates a new, in-place checkpointing, approach to programmer-guided intermittent computation: CAMEL.

2.1 Why Intermittent Computation on Flash Devices?

Frequently checkpointing allows programmer-guided approaches to remove the requirement of dedicated voltage monitoring hardware and reliably execute atomic operations, but exposes any performance limitations associated with the Non-Volatile Memory (NVM) storing checkpoints. For several decades, the only mass-market option for NVM in energy-harvesting-class devices was Flash memory. In the last five years, a new NVM emerged: Ferroelectric Random-Access Memory (FRAM). Following this trend, early energy-harvesting platforms used Flash-based microcontrollers (e.g., WISP 4 [43] and Moo [54]), while more recent platforms use the more esoteric (outside of the lab) FRAM-based microcontrollers (e.g., WISP 5 [34]). According to the WISP 5 developers, the impetus for the transition to FRAM-based devices is the lower cost of writes compared to Flash.

While write latency is one metric to compare NVM technologies, other metrics become important in a world where NVM writes/erases are no longer the limiting factor. Flash-based devices provide several advantages over similar FRAM-based devices: Flash devices are more available and have a larger pool of developers and suppliers, since they have been around for decades, compared to less than a decade for FRAM devices. This trend is unlikely to change soon as Flash-based devices are more available today. Flash also provides a performance advantage, as shown by comparing Dhrystone results [49]. Recent work highlights other advantages of switching to a high-energy, high-efficiency operating point [7], which does not apply to FRAM, due to the additional wait-states required at higher frequencies. Finally, Flash devices tend to contain more SRAM at equivalent NVM sizes [17, 18].

Given the advantages of Flash, why do the most recent energy harvesting platforms [34] and programmer-guided intermittent computation systems [6, 26, 29] target FRAM? Despite the availability and performance advantages of Flash, its slow, high-energy writes/erases are antithetical to the high-frequency checkpoints of continuous checkpointing systems. Programming Flash (i.e., writing) is energy and time-intense as it requires collecting enough charge to raise the voltage of a Flash cell high enough to force charge to flow across the cell's dielectric (e.g., from 2.2V up to 12V). Worse, this process is uni-directional. Thus, to change any single bit of Flash requires copying a segment (512 B) to SRAM, erasing the entire segment, updating the desired bits in SRAM, and writing the updated segment back to Flash. The common-case nature of checkpointing in programmer-guided systems, the cost of writing/erasing Flash memory, and Flash's untimely failure (§2.2) eclipse any benefit Flash offers. Without an alternative to checkpointing to Flash, the vast majority of and most performant microcontrollers will not support software-only intermittent computation. **This paper provides the most performant programmer-guided checkpointing approach invariant of NVM type—without killing Flash.**

2.2 Existing Programmer-guided Systems Kill Flash

Flash cells withstand only a limited number of write/erase cycles before they fail [16]. Each checkpoint requires writing to Flash. Eventually this fills all available Flash memory, requiring an even more expensive erase procedure (because Flash is one-way programmable). Every time a Flash cell undergoes a write/erase cycle its dielectric breaks down. Eventually the dielectric breaks down enough that the cell fails, either not being able to be read or erased correctly. For example, the Flash in the TI microcontrollers that we use is rated to endure 10,000–100,000 write/erase cycles before failure.

This sensitivity to Flash writes/erases is antithetical to the high rate of checkpoints endemic to continuous checkpointing systems. Continuous intermittent computation systems eliminate the need for special-purpose voltage-monitoring hardware by taking many small checkpoints. For example, existing systems checkpoint on the order of once-per-1000 instructions [6, 26, 29]—limiting the deployment lifetimes of energy harvesting systems to a handful of days at best. If it is not possible to replace batteries in energy harvesting devices, then it is equally unpalatable to replace entire devices every few days. **This paper provides the first long lifetime, continuous checkpointing approach for Flash devices.**

2.3 SRAM's Time-dependent Non-volatility

Many existing intermittent computation works assume SRAM loses state completely as soon as the microcontroller can no longer execute. We observe that, due to capacitance in the system, the voltage of a system's power rail gradually reduces from the microcontroller's brown-out voltage to 0V. Due to the difference in the microcontroller's brown-out voltage (e.g., 1.6V) and SRAM's data retention voltage (~0.4V [14, 39]), **SRAM scavenges the otherwise wasted charge to retain state. We refer to this as SRAM's time-dependent non-volatility**: for a period after computation ceases, SRAM acts as a non-volatile memory. This presents an opportunity to leverage SRAM as a checkpoint storage location—as long as SRAM retains data for longer than the off time.

To verify this opportunity, we first quantify how long SRAM provides perfect data retention for. For this experiment, we use a Flash-based MSP430 development board [20] that is representative of energy harvesting-class devices. The literature indicates that two factors dominate the discharge time of a capacitor: capacitor size and temperature. To explore the impact of these variables on SRAM's data retention time, we modify the development board, replacing its 10μF decoupling capacitor with 47μF, 100μF, and 330μF versions. We select the 47μF capacitor as it represents what the most popular energy harvesting devices use [34, 43]. We use larger capacitor sizes to show how system designers can tune the retention time through the capacitor. To control temperature, we perform the experiments in a Test Equity 123H thermal chamber, varying temperature between 20°C and 50°C.

Because SRAM fails bi-directionally, in a board- and noise-dependent pattern [12], for each temperature/capacitor combination, we perform five trials writing all 1s and five of all 0s, checking for data loss at each trial. Figure 1 shows the retention time of our MSP430 microcontroller across a range of temperatures and the three energy storage capacitor sizes. These results show that—even without system designer awareness of SRAM's time-dependent non-volatility—current energy harvesting platforms provide relatively long data retention times.

2.4 Intermittent Off Times are Short

Given that SRAM provides perfect data retention for between 50 seconds and almost 4 hours, the next question to answer is how long unexpected off-times[3] are for the most common energy sources. To answer this question, we perform a meta-analysis of the energy harvesting literature to identify common energy sources and, for each source, a realistic upper bound for off times. This task is complicated by the fact that

[3]We differentiate between expected and unexpected off times. The challenge for intermittent computation is dealing with unexpected power-cycles and their off times; thus that is our focus. In contrast, solar-powered systems experience long off times at night, but this is (predictable) power loss akin to turning off a desktop computer—not intermittent computation.

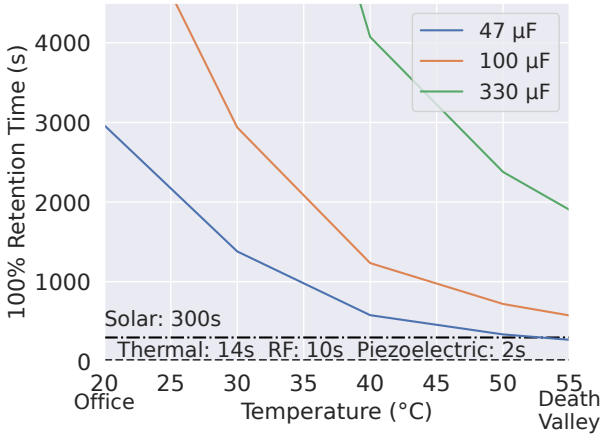


Figure 1: The maximum time (secs) before a single bit fails in SRAM w.r.t. temperature and capacitor size. The horizontal bars represent the maximum off-times from our meta-analysis of off-times reported in the literature.

previous work focuses on on-times due to its reliance on the long-term data retention guarantees of non-volatile memories. Fortunately, by looking at on-times and the frequency of power-on events, we are able to deduce approximate off times. We add the off-times for four energy sources as horizontal lines in Figure 1: RF [9, 27, 41], Thermal [27], Piezoelectric [27, 44], and Solar [9, 27]. When a given capacitor's line is above the horizontal line, the capacitor provides enough perfect data retention time to support operation at the temperature and below. To summarize our meta-analysis: **off-times for most sources are much shorter than the data retention time** provided by existing energy harvesting platforms. In this paper, we design and implement **a system that reliably uses SRAM as a low overhead, long lifetime, non-volatile memory for the short off times common to intermittent computing**, falling back to existing checkpointing to support longer and expected power-off events.

3 Design

We develop CAMEL, a programmer-guided, continuous checkpointing system with the goal of enabling long-life, high-performance intermittent computation on Flash-based devices. CAMEL avoids continuously writing program state to non-volatile memory by preserving in-place SRAM data using differential reusable buffers. Our differential buffer model allows checkpointing just-enough data required to restart program state as opposed to the entire SRAM as implemented previously [49]. We maintain semantically correct execution by ensuring the in-place data remains consistent across power cycles and that tasks always re-execute with known-good data.

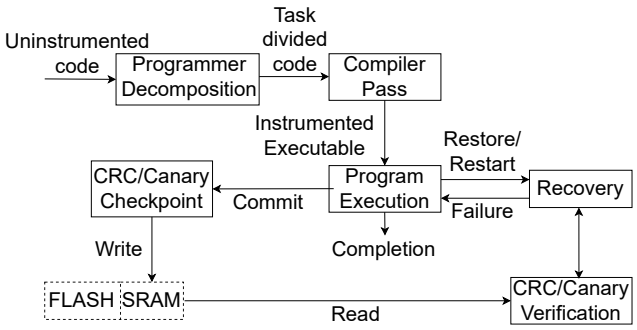


Figure 2: Interaction amongst components within CAMEL.

CAMEL is an amalgamation of three components, working together to guarantee the correct execution of a program on harvested energy. These components are: (1) CAMEL Recovery Routine; (2) CAMEL Tasks; and (3) CAMEL Compiler.

3.1 System Overview

CAMEL supports C programs (including dynamic memory[4] and pointers) where `main()` consists of task function calls connected by control-flow logic (i.e., no assignment statements). Tasks may call other functions, but not other tasks. Figure 2 gives a high level overview of how different components interact within CAMEL. The CAMEL programming model allows the programmer to ensure forward progress of applications on any energy harvesting platform by decomposing source code into a set of individually re-executable tasks. The CAMEL compiler analyzes how tasks interact with shared data in the differential buffers to ensure in-place SRAM data is consistent at run time, despite re-executions after power failures. CAMEL performs idempotence analysis and produces a ready-to-run executable that can be flashed to a board of the programmer's choosing. CAMEL implements the volatile and non-volatile world concept using two differential, swappable buffers—at any given point in execution, a volatile world and a non-volatile world exist. **Tasks interact with data exclusively in the volatile world, whereas recovery pulls data exclusively from the non-volatile world.** Between tasks, CAMEL atomically swaps which buffer represents each world—locking-in forward progress by rendering the updated buffer effectively non-volatile (§3.2). After the swap and before the next task, CAMEL resolves the differences between the up-to-date newly-non-volatile world and the outdated newly-volatile world by copying only the data that the preceding task modified. Following a power failure, CAMEL invokes the recovery routine which continues execution either (1) from the most recent checkpoint in the common case when SRAM retains its data, or (2) from the beginning of the program, when an uncommonly long power failure causes SRAM to lose its data.

[4]The current implementation of CAMEL supports dynamic memory within a task. Supporting dynamic memory between tasks requires dynamically-sized worlds; an engineering challenge that we are addressing as future work.

3.2 Detecting Unexpectedly Long Off Times

SRAM transitions from non-volatility to semi-volatility, gradually approaching full-volatility as supply voltage falls. For any stage beyond non-volatility, the SRAM cells begin losing state, jeopardizing recovery. We employ two methods to detect unexpectedly long off times.

Canary Values: The SRAM cells that fail first upon power loss, and the direction of failure, are decided by manufacturing-time process variation; each device exhibits unique yet consistent failure patterns [12, 13]. We leverage this predictable failure pattern for a low-overhead check of SRAM data retention by writing a pre-determined value to canary memory and checking for it after a power failure. If the first-to-fail cells retain their canary values, we know all SRAM data is intact and can restart from a checkpoint; otherwise, data may be corrupted and we must restart execution from the beginning. SRAM canary values require chip characterization to identify (1) the cells that fail first and (2) the value those cells fail to, which prevents silent failures from the cell failing into the chosen canary value.

Pre-deployment characterization works for three reasons: (1) given a device, SRAM cells lose state in a mostly totally ordered fashion—especially the tail cells [13]; (2) SRAM cell failure ordering is preserved across temperature and voltage fluctuations [10]; and (3) The first-to-fail cells have a reliable power-on state [50]. To determine for a given device the location and values of the canary cells, the user searches through off-times to find the first cells to fail at the shortest off-time. The cells are set to all 1's, then the power is disconnected, then reconnected after the desired off time, then the SRAM state is read-back, looking for failed cells. The user does the same for all 0's. If a cell fails for either case, then it is marked as a failure for that time. Our experiments perform 3 such (Bernoulli) trials at each time step to eliminate noise. We then store the addresses of the first-to-fail cells in non-volatile memory and the value that exposes their failure. The checkpoint routine writes the value to the addresses, while the recovery routine validates it before resuming execution. Programmers can use multiple canary cells for increased resilience at the cost of a few more memory comparisons.

Cyclic Redundancy Checks (CRC): For cases where chip characterization represents unacceptable pre-deployment effort, we explore a second approach based on Cyclic Redundancy Checks. CRCs are common mechanisms for communication systems and applications that need to verify the integrity of received data with high confidence. Hardware sup-

port for CRCs is common to low-powered micro-controllers that send and receive data.

The CRC algorithm divides data by a predetermined generator polynomial using repeated shifts and XOR operations. The output of the CRC algorithm is the remainder of this division, which is stored alongside the data in volatile memory. The state of the application data in the SRAM changes after every task, hence we recompute the CRC between tasks. To verify data integrity after recovery, we recalculate the CRC over the trusted in-place application data and compare the result to the one previously stored in memory. The two remainders must match to conclude that data remains integral.

CRCs guarantee up to G bits of error detection, where G is determined by the variant of CRC used; a 16-bit CRC detects up to 3 flipped bits whereas a 32-bit CRC detects up to 5 flipped bits. Both CRC variants additionally detect all odd-bit errors. For other errors, CRCs provide probabilistic error detection with a chance of missing an error of $1/2^m$, where m is the width of the CRC [23]. For a multi-bit error to go undetected, the checksum of the corrupted and un-corrupted must be the same. The probability of undetected data corruption is further reduced because there is a 50% chance that a failing cell will fail into the correct value; thus, CRCs provide a high-confidence solution to verify SRAM's data integrity.

3.3 Bimodal Recovery Routine

As shown in Figure 2, following a power failure during program execution, the recovery routine passes control to the program from either the start of either the *main* function or the last executed task. We arrive at this decision by recomputing the CRC or checking the canary value, depending on the variant of CAMEL deployed.

To resume execution from the last in-place checkpoint (1) the recomputed CRC should match the one stored in the SRAM or (2) the canary value must be correct, indicating that the data in the non-volatile world was preserved over the power cycle. After passing the integrity check, CAMEL copies the data in the non-volatile world over the volatile world, ensuring that the first task begins with correct data. CAMEL restores control to the program at the beginning of the last partially-completed task by copying the saved register values back to the register file, restoring the program counter last. In the uncommon case—when SRAM fails to retain data due to an unexpectedly long power failure—CAMEL passes control to the beginning of the program and restarts execution.

3.4 CAMEL Tasks

The CAMEL programming model is task-based and depends on the programmer to divide source code into independent atomic tasks. This division enables tracking within-task idempotence [52] by the compiler. We label a section of code as

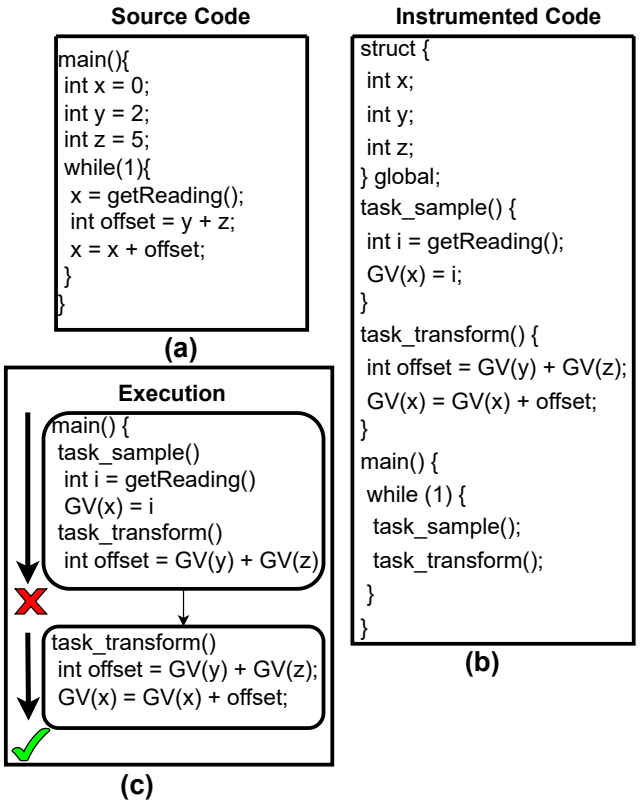


Figure 3: (a) Shows unmodified source code (b) shows the task divided code according to the conventions described §3.4 (c) shows the execution of the code in (b) after it has been instrumented by the compiler.

idempotent if no variable undergoes a *Write-after-Read* dependency [24] within that section. A variable is subject to the *Write-after-Read* dependency when it is first read and then later written in the same task. The CAMEL compiler identifies this sequence by a load followed by a store to the same memory location. Variables within a task are tracked to ensure the consistency of the differential buffers upon task (re)entry post power failure.

Programmers define functions serving as volatile tasks using the *task_* keyword to mark them for tracking by the compiler. CAMEL expects all variables that are to be used by multiple tasks or reused by multiple executions of the same task to be declared as *task-shared* variables. All *task-shared* variables are declared as part of a global structure which serves as the buffer for the volatile and non-volatile worlds. *Task-shared* variables need not be passed to every task individually; instead, they are directly accessible by the use of the *GV()* keyword.

After dividing the application into different worlds using tasks, the flow of the program must be described in *main*—each task must be called in an order that would result in an

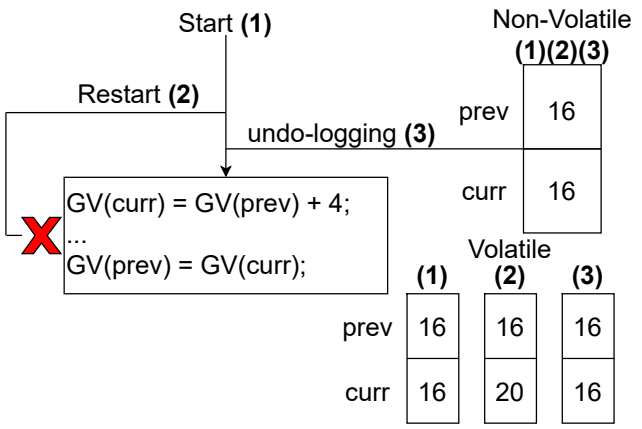


Figure 4: (1) Shows the start of a task (2) Shows a power failure midway execution of a task (3) shows undo-logging before any tasks begins execution. The state of the **non-volatile** and **volatile** buffers is shown after each of the three steps.

identical execution to that of the unmodified version of the program. We limit the use of `main()` to calling tasks and read-only conditionals that determine the next task, as CAMEL does not track idempotency outside of tasks.

3.5 CAMEL Compiler

Our static analysis ensures (1) idempotency of tasks across power cycles and (2) consistency of the *differential shared buffers* across tasks. Any premature writes to the non-volatile buffer by a task persist across power failures, causing the re-execution of the task after recovery to yield different results than expected of it. The compiler statically tracks data in the volatile world and inserts code between tasks to ensure data idempotence upon (re-)entry into a task. This ensures the system-level atomicity of tasks—the results of a task are never committed to the non-volatile world until the task is complete. To achieve idempotency and atomicity of a task, we implement a differential double buffer solution, using the difference between the two buffers to ensure forward progress and re-execute tasks after a power failure.

Two versions of the buffer exist at any time, deemed volatile and non-volatile. Tasks work on global variables in the volatile buffer and do not address the non-volatile buffer, which serves as the fail-safe against memory corruption due to power loss. Between tasks, CAMEL calculates the CRC of registers and the non-volatile buffer and ensures memory consistency through *undo-logging*, whereby CAMEL copies the validated non-volatile buffer over the volatile buffer to undo partial work. Upon task entry, all global variables are in a consistent known state. We illustrate this process in Figure 4.

The successful execution of a task is marked by a commit, which involves swapping the volatile buffer (which contains

```
1. struct {
2.       int x;
3.       int y;
4.       int z;
5.       int temp;
6.       int result;
7. };
9. void task_compute() {
10.      GV(temp) = GV(x) + GV(y);
11.      GV(result) = GV(result) + GV(temp);
12.}
```

Figure 5: CAMEL programming interface. `result` undergoes a write-after-read sequence in line 11.

updated state after task completion) and non-volatile buffer and re-calculating the CRC on the newly non-volatile buffer. Crucially, we implement the swap as a pointer re-assignment, which reduces data movement: instead of copying data from a dedicated non-volatile buffer, modifying it in a private volatile buffer, and re-committing it to the non-volatile store, tasks work *directly on the data in the volatile world*, which is later rendered non-volatile by CAMEL as part of the commit process. We discuss enforcing the atomicity of the commit in §4.3 to prevent incorrect execution stemming from an interrupted commit. To ensure correct forward progress, the CAMEL compiler resolves differences between the two worlds to keep program state consistent between tasks.

The compiler makes tasks idempotent by only undo-logging variables that undergo the *write-after-read* dependency in a task. These variables cause memory inconsistencies and incorrect execution if a power failure and re-execution occur after the write and before the commit, as the preceding read will read a different value from the last execution. We refer to these variables as non-idempotent; non-idempotent variables must be undo-logged following a power failure to ensure idempotence. Between tasks, we atomically swap the worlds and ensure consistency using a partial update of the newly-non-volatile world. Each changed variable must be updated in the volatile buffer to ensure the two buffers remain consistent between tasks; we use different functions to update each variable type.

CAMEL implements data copying between buffers using several internal functions to cover the three types of variables supported by C: scalars, compounds, and unions. The `copy_scalar` function copies scalar values between buffers; for contiguous variables (i.e., compounds and unions), CAMEL uses one of the following functions as appropriate:

- `copy_array`: Logs an entire contiguous variable using `memcpy`.
- `copy_array_scalar`: Logs an element of a contiguous variable based on a scalar index stored in the differential buffers.

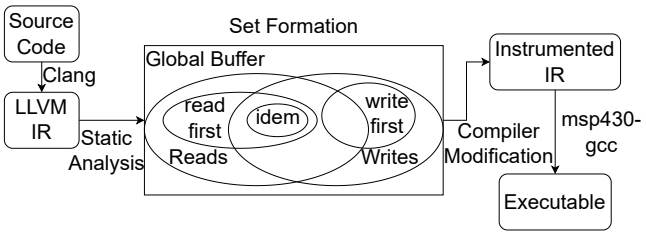


Figure 6: Pipeline for the generation of a CAMEL executable.

- `copy_array_scalar_local`: Logs an element of a contiguous variable based on a task-local scalar index. This function saves the value of the local variable when it is used to index into the shared array and uses it to perform only the required copies during logging.

4 Implementation

4.1 Compiler Analysis

We implement CAMEL as a compiler pass in LLVM 10.0.0 [25], which ultimately produces MSP430 assembly; we use *msp430-gcc* to generate the corresponding executable. The compiler's aim is to populate sets of read and written variables to find non-idempotent memory accesses. Figure 6 illustrates the pipeline of our analysis from source code to an executable. Our pass statically analyzes the structured, architecture independent LLVM IR generated using the task-divided code, written by the programmer using the conventions highlighted in §3.4. LLVM provides interfaces to traverse, interact and change the IR. Our pass analyzes every function declared with the prefix *task_*. We focus our analysis on instructions in IR that are directly involved in interacting with memory locations, namely load, store and memcpy. Furthermore, we are only interested in said instructions if their operands are a part of the volatile world global buffer as only that buffer is impacted by task execution.

Our pass begins by performing intra-module static analysis on the LLVM IR, examining all function declarations to determine whether they are tasks. Once a task is identified, our pass traverses the control-flow graph of the function, searching for loads, stores and memcpys. After identifying the instructions of interest in the control-flow graph, we backtrack from the operands of the instructions to their first use in a task. At this stage, we only add variables backed by the global buffers to their respective read/write set. In addition to a set of read and written variables, we maintain a set of read-first variables—variables that are read before they are written in a task. Once all sets are populated, we take the intersection of the read-first and write sets. This produces a set with the variables subject to a *write-after-read* dependency, which Figure 5 demonstrates. Note that our analysis is context-insensitive: when the compiler cannot predict which branch of a conditional statement will execute and one of the path would mark the variable as read-first, the compiler considers the variable read-first regardless of the execution path. This static analysis guarantees the detection of all idempotent violations within a task by pessimistically analyzing all execution paths.

4.2 Compiler Modifications

The compiler inspects `main()` to locate task call sites and inserts code to (1) undo-log data before a task and (2) copy data between the volatile and non-volatile worlds to ensure buffer consistency after completing a task. Undo-logging copies variables in the *write-after-read*-vulnerable set from non-volatile to volatile using the functions implemented in §3.5, while the inter-task buffer swap/update following successful task completion copies all modified variables.

For arrays, the user may choose to update a specific index of the array using a variable defined in the volatile world buffer or a local, task-defined variable. The compiler can insert logging code for any of these two variants. If the index is part of the volatile world buffer, the compiler loads the value of the variable from the buffer, uses the built-in LLVM `getelementptr` instruction to get the array from the buffer and logs the variable. If however, the index is a local variable (i.e., not part of the buffer), we insert code in tasks to store the value of the index in the volatile world buffer at the time of change. We then use this global variable to log the specific index of the array.

4.3 Recovery

The recovery component has two major elements: the commit and the recovery functions. Algorithm 1 describes our commit procedure. Volatile and non-volatile worlds are implemented as pointers to global buffers, which allows us to swap the values of each pointer to swap worlds. To ensure the atomicity of commits, the decision of which pointer points to which buffer is determined by a flag value that is inverted at the end of each commit. The commit procedure saves all registers to a protected region in the non-volatile buffer then updates the canary value or re-calculates the CRC. We implement the function to save registers and calculate the CRC in native MSP430 assembly; the argument to the `SAVE_REGISTERS` function is the memory location to place the registers in. The CRC, when used, is calculated over the saved register file and non-volatile world buffer and excludes the CRC result itself.

We modify the MSP430 reset vector to point to the recovery function when the device regains power. The recovery function passes control to the program by either restarting from the beginning of the program or from the last in-place checkpoint in the SRAM, based on whether the SRAM integrity check passes. The recovery routine first reads the flag value,

Algorithm 1 Task Commit Routine.

```
1: NON_VOLATILE ← FLAG ? &BUF1 : &BUF2
2: VOLATILE ← FLAG ? &BUF2 : &BUF1
3: SAVE_REGISTERS(NON_VOLATILE->reg_file)
4: Guard_BUFFER_REGS_integrity(CRC_MODE)
5: FLAG ← not FLAG
```

which determines which global buffer represents which world. Then, depending on the integrity check strategy, the routine either recomputes the CRC over the non-volatile world buffer or validates the canary value's integrity. If the non-volatile world's contents are valid, recovery commences: 1) the non-volatile world is copied to the volatile world; 2) the platform is re-initialized; 3) register values are restored from the non-volatile world buffer; and 4) the program counter is restored, resuming execution from the last in-place checkpoint.

4.4 Correctness

Correctness is a first-class part of our design and implementation processes. Our correctness measures validate that our CAMEL implementation (1) generates programs capable of maintaining semantically correct execution on harvested energy and (2) that the CRC and Canary strategies both detect unexpectedly long off times that corrupt SRAM. To obtain a ground truth to compare compiler-instrumented output against, we manually instrument all benchmarks and manually compare the generated assembly against the CAMEL-instrumented assembly. Our comparison shows that there are no differences between the two, meaning CAMEL is capable of inserting fault-free code to log data used across different benchmarks. Our second line of defense is a set of regression tests that capture corner cases, the data types available in C, and bugs in earlier versions of CAMEL. Third, we conduct 10 trials of execution for every benchmark to ensure correctness. In each trial, we execute the uninstrumented and instrumented benchmarks, comparing the final state of the system after completion. While executing the instrumented programs, we introduce approximately 20 random power failures, reflective of real-wold energy harvesting traces [9]. Finally, we simulate the uncommon case of extended off times to validate the effectiveness of the CRC and the canary.

5 Evaluation

We evaluate CAMEL's effect on end-to-end system performance on a physical MSP430G2955 [15], a Flash-based energy-harvesting-class microcontroller comparable to those found in past work [29, 49], running on harvested energy emulating a typical RF-powered batteryless device (§ 5.1). For comparison to state-of-the-art Flash-based intermittent computing, we implement TotalRecall [49] on the same device.[5] We evaluate the canary-based implementations for both CAMEL and TotalRecall in order to focus on the highest-performance, hardware-invariant (i.e., no need for hardware CRC) systems available. Finally, we simulate CAMEL for the FRAM-based MSP430FR6989 [19] to evaluate its runtime, binary size, and checkpointing behavior against comparable task-based intermittent systems: DINO [26], Chain [6], and Alpaca [29]. For Flash-based devices, our evaluation demonstrates that CAMEL (1) provides performant, long-life intermittent computation where it was previously limited by voltage monitoring hardware, and (2) enables reliable atomic operations where the state-of-the-art fails due to erroneous checkpoint placement. On FRAM devices, CAMEL outperforms state-of-the-art task-based systems using a novel buffer design which minimizes data movement.

We first evaluate CAMEL's performance on five software benchmarks developed in past work [6, 26, 29], representing applications commonly found in energy harvesting systems:

Activity Recognition (AR): uses simulated samples from a three-axis accelerometer to train a nearest neighbor classifier and determine whether a device is moving.
Bit Count (BC): uses seven different algorithms to count the set bits in a given sequence.
Cold-Chain Equipment Monitoring (CEM): simulates input data from a temperature sensor and later compresses the data using LZW compression [46].
Cuckoo Filter (CF): A Cuckoo filter is a data structure used to efficiently test for set membership [8]. This benchmark stores random data in a Cuckoo filter and later queries the filter to recover the data.
Data Encryption (RSA): RSA is a widely used public-key cryptosystem [42]. This benchmark encrypts a string using a user-defined encryption key stored in memory.

We also evaluate CAMEL on three peripheral-focused benchmarks to explore how the existing options for intermittent computation on Flash-based systems interact with such systems:[6]

Transmit: Transmit data over the on-chip UART to a desktop computer.
Actuate: Generate a sine wave using an external Digital-to-Analog Converter [32] (DAC) connected via I^2C.
Sense: Use the on-chip ADC and a timer to sample the output of an external light sensor [47] twice per second.

5.1 Experimental Setup

Our experimental setup draws motivation from previous work [9] on emulating environmental conditions for real-world energy harvesting use cases in experimental and in-

[5] The MSP430G2955 on-chip ADC is software-configurable to monitor supply voltage, enabling direct comparison without hardware modifications.

[6] We evaluate the peripheral-focused benchmarks primarily for correctness rather than performance, as each benchmark is a "minimal working example" for the corresponding peripheral. As a result, end-to-end performance is dominated by peripheral power consumption (rather than software behavior).

lab setups. We use a secondary MSP430 controlling a high-drive Digital-to-Analog Converter (DAC) to deliver a programmable power trace to the device under test; the secondary microcontroller monitors output voltage and supply current across a sense resistor and varies the DAC voltage to deliver the programmed input power. We evaluate each system running on the MSP430G2955 under three input power scenarios previously recorded using a 915 MHz RF transmitter/harvester pair [36–38] in an active office environment:

- **Mobile**: The transmitter is mounted in a room, while the harvester is carried in, around, and through the room. Average power input: 0.5 mW.
- **Cart**: The transmitter is on a table, while the harvester is on a cart moving towards, around, and away from it. Average power input: 2.12 mW.
- **Obstruction**: The transmitter and harvester are on a table approximately 1 meter apart, while various obstructions (laptops, metal water bottles, etc.) periodically move between the two. Average power input: 0.227 mW.

These power scenarios provide a range of realistic power levels for an RF energy harvesting system. The power-control system charges a 2 mF supply capacitor connected to an intermediate power gate that connects the MSP430G2955 under test to the supply once capacitor voltage reaches 3.3V and disconnects it once it reaches 1.8V. We instrument each benchmark to drive a digital output pin high upon completion, and monitor this output to record total runtime. To minimize randomness, we perform 10 trials of each benchmark at 20°C.

5.2 Programmer Effort

CAMEL requires the programmer to reason about the energy consumption and data flow of their code and divide it into tasks accordingly. This has the potential to introduce a significant design-time burden if the programmer needs to refactor existing code; the specific level of effort depends on the original code base. Porting code from prior intermittent task-based models (e.g., Alpaca) introduces minimal burden: the only addition our current CAMEL implementation requires is the use of the GV keyword to access task-shared variables (which we plan to automate in the future). Porting plain C code requires more effort as programmers must decompose code into atomic, self-contained tasks. Fortunately, task-based programming is already a common motif in embedded system design as many applications consist of series of tasks with natural boundaries (e.g., sensing, computing on, and transmitting data) [1]. Thus, we expect that programmers porting existing code bases would often only need to convert code to the CAMEL *syntax* rather than significantly restructure the program.

Shifting responsibility to the programmer also has potential performance consequences. While the range of appropriate task sizes depends mainly on the size of the storage capacitor (and thus is known at design time), the programmer is still ultimately responsible for keeping task size within this range. Undersized tasks reduce performance as Camel takes checkpoints at each task boundary, increasing the number of unused checkpoints if software hits many task boundaries during a single on-period. Oversized tasks threaten non-termination as Camel must complete at least one task per on-period to make forward progress. While we do not include them here, existing techniques to optimistically skip checkpoints [30] or automatically scale task size [31] are interesting avenues for expanding on CAMEL.

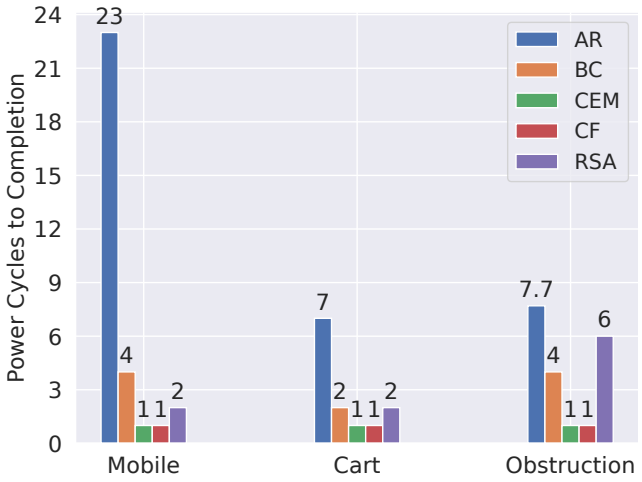


Figure 7: Relative power cycles to completion for benchmarks using TotalRecall compared to CAMEL.

5.3 Runtime: Flash Platforms

For an energy-constrained batteryless system where the power draw of active execution eclipses harvester input power, execution time is dominated by charging periods during which the system is inactive. Energy efficiency determines the number of charge-discharge cycles to complete a workload, which dictates overall performance for typical batteryless devices. Figure 7 shows the number of power cycles for TotalRecall to complete each benchmark under various power scenarios running on the Flash-based MSP430G2955, normalized to the power cycle count for CAMEL under the same conditions. Although CAMEL incurs higher software overhead than TotalRecall by creating and maintaining inter-task checkpoints, the energy savings from eschewing voltage monitoring with the ADC enable the CAMEL-based system to complete each benchmark in fewer power cycles under all circumstances. ADC power consumption means that TotalRecall has less energy available for software execution in each power cycle, forcing the system to power down and wait for more energy—

up to 23 times more than when using CAMEL, as in the AR benchmark on both the Mobile and Obstruction power traces.

Less time spent waiting for incoming power means that CAMEL ultimately completes energy-constrained benchmarks well before TotalRecall: CAMEL reduces the overall time to completion for the AR, BC, and RSA benchmarks by an average of 82.1%, 78.2%, and 66.9% across all power scenarios, respectively. CAMEL marginally underperforms TotalRecall on the CEM and CF benchmarks because these benchmarks both complete within a single power cycle under all conditions—i.e., they are not energy-constrained, and the higher energy cost of TotalRecall is not reflected in their execution time. For typical batteryless applications running across multiple power cycles, however, CAMEL outperforms the existing solution for Flash-based systems by a factor of 3-5x.

5.4 Peripheral Operations

While efficient software execution is essential to performant intermittent systems, many batteryless system deployments must interact with the physical world using on-chip or on-board peripheral devices. To compare CAMEL with the state-of-the-art for these systems, we first run the three peripheral-focused benchmarks described in § 5 on TotalRecall under both continuous and intermittent power. While each benchmark completes normally under continuous power conditions, all fail when subjected to intermittent power: TotalRecall fails to correctly re-execute atomic operations (UART and I^2C transmissions, respectively) for the **Transmit** and **Actuate** benchmarks after checkpointing, and infinitely hangs during the **Sense** benchmark after failing to reconfigure the sample timer. Applications like **Sense** present an additional problem because they compete with TotalRecall for use of the ADC, preventing checkpointing during measurement and introducing the potential for semantically incorrect code re-execution.

We demonstrate how CAMEL enables peripheral-focused intermittent operation on Flash platforms by porting each benchmark to the CAMEL task model, which ensures that peripheral devices are always correctly reconfigured after a power loss and atomic interactions are correctly re-executed. Porting existing code to the task-based model requires some one-time programmer effort (e.g., combining sensor configuration and sampling into a single atomic task), but ensures atomic operations complete successfully regardless of power conditions. Under both continuous and intermittent power, our CAMEL-based implementations of each peripheral-focused benchmark complete successfully—enabling peripheral operations on Flash-based batteryless systems using an intuitive and scalable programming model.

5.5 Runtime: FRAM Platforms

CAMEL's software-only approach to intermittent computing using SRAM data retention yields significant improvements

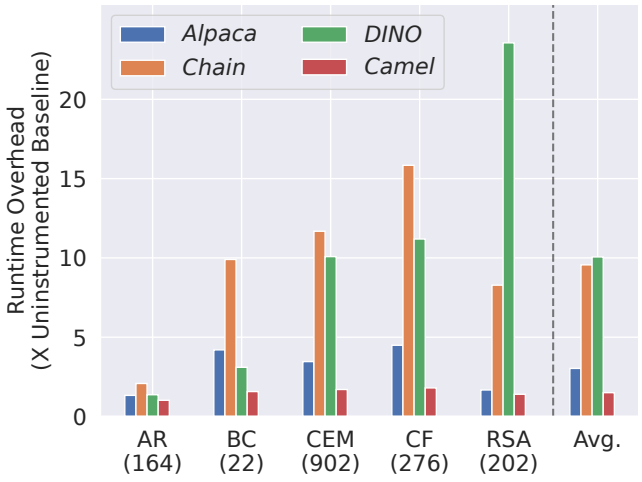


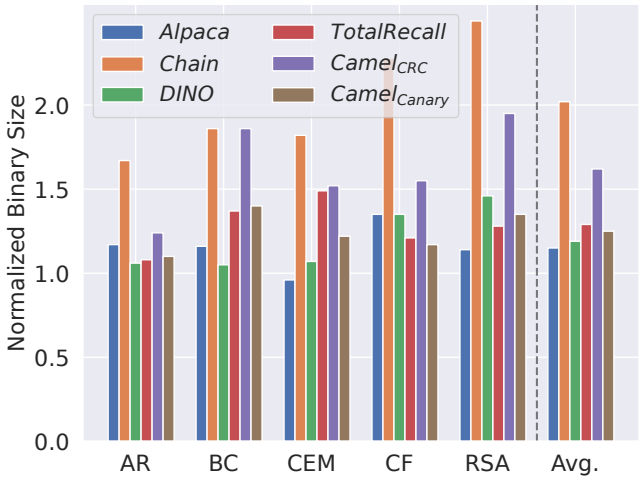
Figure 8: CAMEL run-time overhead for FRAM-based devices. The global buffer size in bytes for each benchmark is shown in parenthesis.

	AR	BC	CEM	CF	RSA	avg.
DINO [26]	1136	717	259	324	1830	788
Chain [6]	2008	717	231	452	315	744
Alpaca [29]	2008	717	225	452	315	743
CAMEL	1999	709	114	385	254	692

Table 1: Number of checkpoints recorded by each task-based system across all benchmarks.

over state-of-the-art just-in-time techniques, but how does it compare to previous task-based programming models? We contextualize CAMEL's improvements by comparing it to DINO [26], Chain [6], and Alpaca [29], three task-based systems designed for FRAM platforms. Running each system on an FRAM device allows us to isolate the effects of CAMEL's differential buffer design without the overhead of an SRAM integrity check. We compile all task-based software benchmarks for the FRAM-based MSP430FR6989 [19] and run them in a modified version of the mspdebug MSP430 simulator [5], which provides fine-grain performance statistics such as CPU cycle count and number of checkpoints recorded.

Figure 8 and Table 1 illustrate the normalized runtime overhead and number of checkpoints taken by each task-based system, respectively. CAMEL's design marginally reduces the total number of checkpoints taken—approximately a 7% average reduction compared to Alpaca, the next-best system—but reducing data movement overhead using CAMEL's differential buffer system means that CAMEL ultimately outperforms Alpaca by an average factor of 2x, demonstrating the power of CAMEL's approach even on FRAM-based devices.



Inter-task Commits: The commit routine following each task is a primary source of runtime overhead for CAMEL and varies with CAMEL implementation. CAMEL$_{Canary}$'s commit has a constant runtime overhead across all benchmarks (∼120 cycles), as the canary values ensuring SRAM integrity are set at power-on and do not need to be updated between tasks. The CAMEL$_{Canary}$ commit routine only needs to swap differential buffers and save architectural registers. Because CAMEL$_{CRC}$ needs to compute an integrity check over the safe CAMEL buffer between each task, commit overhead is higher and a function of buffer size: across all benchmarks, the average cycle count for a CAMEL$_{CRC}$ commit is ∼7600 cycles. Most of these cycles are spent calculating the CRC, which our implementation does in software; using hardware CRC support common to energy-harvesting-class devices would reduce CAMEL$_{CRC}$ commit runtime to close to that of CAMEL$_{Canary}$.

5.6 Binary Size

Figure 5.5 illustrates the increase in binary size of CAMEL and comparison systems, normalized to the binary size of the uninstrumented benchmarks. CAMEL$_{\text{canary}}$ produces smaller binaries compared to CAMEL$_{\text{crc}}$, because the sizes of the recovery routine and inter-task commits are significantly reduced when removing the CRC code. CAMEL$_{\text{canary}}$ reduces binary size when compared to all prior work, while CAMEL$_{\text{crc}}$'s binary overhead impact is comparable (1% larger than Alpaca [29]).

The commit routine which accompanies every CAMEL task can be implemented as an inline or a naked function; Figure 5.5 shows the overhead incurred by the inline version. Both approaches produce binaries that scale linearly with the number of tasks in a benchmark; the inline version produces larger but faster executables, while the naked function approach produces smaller but slower executables. A naked function call incurs a fraction of the run-time overhead of a regular function call. It increases the run-time overhead of the commit routine by a constant number of CPU cycles (∼5), resulting in a negligible change in the overall run-time.

5.7 Automatic Systems

Automatic checkpointing systems [30, 52] remove all burden from the programmer at the cost of increased checkpoint rate. Because it is not the focus of the paper we exclude detailed results, but our experiments show that CAMEL performs 4x better than Ratchet [52] on FRAM-based boards, with even better results on Flash-based devices due to Flash write/erase overhead. Given that Chinchilla's run-time overhead is comparable to Alpaca [30], we expect a similar 2x improvement.

6 Related Work

Just-in-time checkpointing approaches backup volatile state to non-volatile memory just-in-time, i.e., with just enough energy remaining to write the checkpoint [2, 3, 21, 41]. This requires the ability to measure the energy in the system's storage capacitor. The need for an energy monitor increases complexity and cost, as well as increasing overall energy usage—introducing another draw on the energy available for useful computation [52]. Just-in-time checkpointing systems minimize software overhead, but also introduce correctness and programmability concerns when interacting with peripheral devices or atomic operations. Samoyed [31] targets just-in-time systems with low-power energy monitors and introduces an interface for programmers to denote sections of code as atomic, combining the state-rollback mechanisms of task-based approaches with the low software overhead of just-in-time systems. Samoyed brings reliable atomic peripheral operations to just-in-time intermittent systems, but depends on rollback mechanisms designed for FRAM-based systems; one direction for future research is combining CAMEL's efficient task model with TotalRecall's just-in-time approach to enable similar functionality without high-performance NVM.

Continuous checkpointing eschews a single, large, just-in-time, checkpoint of the entire volatile program state for many small checkpoints of the volatile program state needed to resume execution. Continuous checkpointing trades performance for hardware-support-free intermittent computation; unfortunately, all such existing approaches—whether architecture- or software-driven—are incompatible with Flash devices due to Flash's endurance and power limitations [16].

Continuous checkpointing fits naturally with sequential hardware design which treats flip-flops as non-volatile state and the combinational logic between them as volatile state. Idetic [33] employs this model to support intermittent operation at the circuit level using existing high-level synthesis tools. Conventional processor pipelines are compatible with continuous checkpointing when treating pipeline registers as non-volatile state and operations between stages as volatile

state. Non-volatile processors [27, 28] leverage this observation by implementing pipeline registers with non-volatile flip-flops (e.g., FRAM). An alternative to non-volatile processors is Clank [11], which enforces dynamic memory idempotency.

CAMEL improves existing programmer-guided intermittent computation systems: combining Chain's [6] expressive programming interface with idempotence analysis as used by Alpaca [29]. From this, CAMEL introduces the idea of of swappable mixed-volatility worlds backed by a differential analysis that allows data reuse across tasks. This improves performance regardless of NVM type by reducing redundant data copying. To enable programmer-guided approaches on Flash devices, CAMEL bifurcates program data into a inter-task non-volatile world and a intra-task volatile world.

Ratchet [52] and Chinchilla [30] replace programmer reasoning with compiler analysis to produce an automatic approach to supporting intermittent computation—at the cost of removing the abstraction of forward progress atomicity from the programmer. Ratchet decomposes programs into restartable units using idempotence analysis while Chinchilla [30] builds on Ratchet with a smart timer and basic-block-level energy estimation to elide checkpoints at run time.

7 Conclusion

This paper exposes and addresses the lifetime and performance limitations of programmer-guided approaches to intermittent computation on both Flash devices. The improvements center on the abstraction of two worlds that co-exist during program execution: a non-volatile world that contains the data tasks use to communicate and that is used for post-power-cycle recovery and a volatile world that contains data used by a task. The result is the first long-life, programmer-guided intermittent computation system for Flash devices and highest-performance for FRAM devices.

8 Acknowledgements

We thank the anonymous reviewers for their feedback and suggestions that enhanced the quality of this work. The project depicted is sponsored by the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited. This material is based upon work supported by the National Science Foundation under Grant No. 2240744.

References

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

[2] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, March 2016.

[3] Domenico Balsamo, Alex Weddell, Geoff Merrett, Bashir Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems. In *IEEE Embedded Systems Letters*, 2014.

[4] BBC News. Samsung confirms battery faults as cause of Note 7 fires, January 2017. https://www.bbc.com/news/business-38714461.

[5] Daniel Beer. Mspdebug, 2022. https://github.com/dlbeer/mspdebug.

[6] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 514–530, October 2016.

[7] H. Desai and B. Lucia. A power-aware heterogeneous architecture scaling model for energy-harvesting computers. *IEEE Computer Architecture Letters*, 19(1):68–71, 2020.

[8] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014.

[10] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson,

and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computer Systems*, 15(4):77:1–77:28, August 2016.

[11] Matthew Hicks. Clank: Architectural support for intermittent computation. In *International Symposium on Computer Architecture*, ISCA, pages 228–240, 2017.

[12] D. E. Holcomb, W. P. Burleson, and K. Fu. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, September 2009.

[13] Daniel E. Holcomb, Amir Rahmati, Mastooreh Salajegheh, Wayne P. Burleson, and Kevin Fu. Drv-fingerprinting: Using data retention voltage of sram cells for chip identification. In *Proceedings of the 8th International Conference on Radio Frequency Identification: Security and Privacy Issues*, RFIDSec'12, pages 165–179, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] G. Huang, L. Qian, S. Saibua, D. Zhou, and X. Zeng. An efficient optimization based method to evaluate the drv of sram cells. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(6):1511–1520, June 2013.

[15] Texas Instruments. MSP430G2955, MSP430G2855, MSP430G2755 Mixed-Signal Microcontroller, March 2013. https://www.ti.com/lit/ds/symlink/msp430g2955.pdf.

[16] Texas Instruments. MSP430 Flash Memory Characteristics (Rev. B), 2018. http://www.ti.com/lit/an/slaa334b/slaa334b.pdf.

[17] Texas Instruments. MSP430F5438A—MSP430F543xA, MSP430F541xA Mixed-Signal Microcontrollers, September 2018. http://www.ti.com/lit/ds/symlink/msp430f5438a.pdf.

[18] Texas Instruments. MSP430FR5964—MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, August 2018. http://www.ti.com/lit/ds/symlink/msp430fr5964.pdf.

[19] Texas Instruments. MSP430FR698x(1), MSP430FR598x(1) Mixed-Signal Microcontrollers, 2018. http://www.ti.com/lit/ds/symlink/msp430fr6989.pdf.

[20] Texas Instruments. MSP430G2553 LaunchPad Development Kit (MSP-EXP430G2ET), 2018. http://www.ti.com/lit/ug/slau772/slau772.pdf.

[21] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *International Conference on VLSI Design and International Conference on Embedded Systems*, 2014.

[22] Joseph Kahn, Randy Katz, and Kristofer Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[23] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks, 2004*, pages 145–154, June 2004.

[24] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Conference on Programming Language Design and Implementation*, PLDI, pages 475–486, 2012.

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

[26] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Conference on Programming Language Design and Implementation*, PLDI, pages 575–585, 2015.

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

[28] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Sampson, and Vijaykrishnan Narayanan. Nonvolatile Processor Architectures: Efficient, Reliable Progress with Unstable Power. In *IEE Micro Volume 36, Issue 3*, 2016.

[29] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 96:1–96:30, October 2017.

[30] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 129–144, November 2018.

[31] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI, pages 1101–1116, 2019.

[32] Microchip. MCP4725 12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6, 2009. https://ww1.microchip.com/downloads/en/devicedoc/22039d.pdf.

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

[34] University of Washington. WISP 5 GitHub, April 2014. http://www.github.com/wisp/wisp5.

[35] Sandro Pinto and Nuno Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6), January 2019.

[36] Powercast. P2110B 915 MHz RF Powerharvester Receiver, December 2016. https://www.powercastco.com/wp-content/uploads/2016/12/P2110B-Datasheet-Rev-3.pdf.

[37] Powercast. TX91501B – 915 MHz Powercaster Transmitter, October 2019. https://www.powercastco.com/wp-content/uploads/2019/10/User-Manual-TX-915-01B-Rev-A-1.pdf.

[38] Powercast. 915 mhz dipole antenna datasheet, November 2020. https://www.powercastco.com/wp-content/uploads/2020/11/DA-915-01-Antenna-Datasheet_new_web.pdf.

[39] Huifang Qin, Yu Cao, Dejan Markovic, Andrei Vladimirescu, and Jan Rabaey. Sram leakage suppression by minimizing standby supply voltage. In *Proceedings of the 5th International Symposium on Quality Electronic Design*, ISQED '04, pages 55–60, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Benjamin Ransford and Brandon Lucia. Nonvolatile Memory is a Broken Time Machine. In *Workshop on Memory Systems Performance and Correctness*, 2014.

[41] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[42] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

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

[44] Henry Sodano, Gyuhae Park, and Daniel Inman. Estimation of Electric Charge Output for Piezoelectric Energy Harvesting. In *Strain, Volume 40*, 2004.

[45] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. Nonvolatile processors: Why is it trending? In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, pages 966–971, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association.

[46] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[47] Vishay. TEMT6000X01 Ambient Light Sensor, 2011. https://www.vishay.com/docs/81579/temt6000.pdf.

[48] Mark Weiser. Ubiquitous computing. *Computer*, 10:71–72, 1993.

[49] Harrison Williams, Xun Jian, and Matthew Hicks. Forget failure: Exploiting SRAM data remanence for low-overhead intermittent computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 69–84, March 2020.

[50] Harrison Williams, Alexander Lind, Kishankumar Parikh, and Matthew Hicks. Silicon Dating. *arXiv*, abs/2009.04002, 2020. _eprint: 2009.04002.

[51] Harrison Williams, Michael Moukarzel, and Matthew Hicks. Failure sentinels: Ubiquitous just-in-time intermittent computation via low-cost hardware support for voltage monitoring. In *International Symposium on Computer Architecture*, ISCA, pages 665–678, June 2021.

[52] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 17–32, November 2016.

[53] X. Wu, I. Lee, Q. Dong, K. Yang, D. Kim, J. Wang, Y. Peng, Y. Zhang, M. Saliganc, M. Yasuda, K. Kumeno, F. Ohno, S. Miyoshi, M. Kawaminami, D. Sylvester, and D. Blaauw. A 0.04MM3 16NW wireless and batteryless sensor system with integrated cortex-m0+ processor

and optical communication for cellular temperature measurement. In *IEEE Symposium on VLSI Circuits*, pages 191–192, June 2018.

[54] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. Moo: A Batteryless Computational RFID and Sensing Platform. In *Technical Report UMCS-2011-020*, 2011.