Forget Failure: Exploiting SRAM Data Remanence for Low-overhead Intermittent Computation

Harrison Williams
Virginia Tech
Blacksburg, Virginia
hrwill@vt.edu

Xun Jian
Virginia Tech
Blacksburg, Virginia
xunj@vt.edu

Matthew Hicks
Virginia Tech
Blacksburg, Virginia
mdhicks2@vt.edu

Abstract

Energy harvesting is a promising solution to power billions of ultra-low-power Internet-of-Things devices to enable ubiquitous computing. However, energy harvesters typically output tiny amounts of energy and, therefore, cannot continuously power devices; this leads to intermittent computing, where the energy harvester periodically charges a capacitor to sufficient voltage to power brief computation, until the capacitor's charge is drained, and the cycle repeats. To retain program state across frequent power failures, prior work proposes checkpointing program state to Non-Volatile Memory (NVM) before a power failure. Unfortunately, the most widely deployed, highest performance, and lowest cost devices employ Flash as their NVM, but the power, time, and endurance limitations of Flash writes are incompatible with the frequent NVM checkpoints of intermittent computation.

The multi-year data retention of Flash is overkill for retaining program state across intermittent computing's short power-off times (e.g., <1*s*). We observe that even after computation stops due to low voltage, charge remains in the system; this remaining charge keeps the voltage high enough to maintain data in SRAM—effectively making it a NVM—for 10's of minutes post-power loss. This paper explores how to leverage SRAM's data remanence to boost common-case performance and energy efficiency for Flash-based intermittent computation systems. We propose TotalRecall, a library-level, in-situ, checkpointing technique that retains program state in SRAM and identifies when SRAM acts as a NVM, falling back to conventional NVM checkpoints in the rare event of long off times. Our evaluation, on real hardware, using benchmarks from Texas Instruments, shows that TotalRecall incurs overheads as low as 0.8%—up to over 350*x* faster than checkpointing to Flash.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378478

CCS Concepts. • **Computer systems organization → Embedded software**; *Reliability*.

Keywords. Energy Harvesting, Intermittent Computation

ACM Reference Format:
Harrison Williams, Xun Jian, and Matthew Hicks. 2020. Forget Failure: Exploiting SRAM Data Remanence for Low-overhead Intermittent Computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378478

1 Introduction

As modern microcontrollers become smaller and more energy efficient, system designers are finding novel applications for these devices in a variety of areas. The advent of wearable devices [53], RFID smart tags [54], and smart dust [30] represents potential for designers to leverage new low-power chips towards more ubiquitous computing and a greatly expanded Internet-of-Things (IoT). The limiting factor is not microcontrollers themselves, but the batteries powering such devices—today's battery-powered devices are not limited in size by the density of devices on silicon, but by energy demands dictating a battery that often makes up the bulk of the space and cost of the product. Both the economic and engineering feasibility of shrinking IoT and other ubiquitous computing devices depend on foregoing batteries.

A necessary component of a batteryless future is energy harvesting circuits, which draw power from the environment from sources such as RFID readers [31] or ambient vibration [49]. Many energy harvesters cannot output enough energy to continuously power microcontrollers; they trickle charge into a capacitor, providing enough power for brief computation, until the capacitor empties sufficiently, and the cycle repeats.

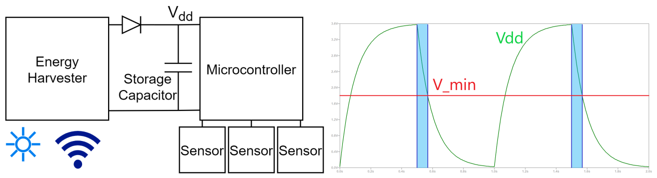
To enable reliable intermittent computation in the presence of frequent power failures, prior work proposes saving program state to Non-Volatile Memory (NVM) (e.g., Flash [47] and Ferrorelectric RAM (FRAM) [20]), preserving program state that is otherwise lost without power. They propose checkpointing volatile program state (e.g., registers and stack) to NVM, either before the next power failure [1, 2, 28, 47] or periodically at compiler-dictated points [5, 34, 37, 38, 52].

The most ubiquitous, highest performance, and lowest cost NVM is Flash. Unfortunately, while many aspects of Flash memory are great for energy harvesting systems, **Flash writes are particularly ill-suited for intermittent computation**. Flash writes are much slower and more energy intensive than writes to volatile memory [51]. Additionally, Flash's limited write endurance results in existing checkpointing schemes killing it in hours to a year of operation [23]. This is why recent work targets the more esoteric FRAM-based devices despite the many advantages of Flash-based devices [1, 2, 28, 38, 52].

We observe that Flash retains program state for years, but **intermittent computation only requires retention during short power-off times** (e.g., $<1s$). This paper answers the question, **"Is there a way to avoid paying the price for retention guarantees that intermittent computation does *not* benefit from?"** We answer this question affirmatively by exploiting the time-dependent volatility of Static Random-Access Memory (SRAM). The driving observation is that when a microcontroller hits its brown-out voltage (e.g., $1.8V$) and ceases computation, significant charge remains in the system; this remaining charge leaks away slowly, resulting in a gradual transition from the brown-out voltage to $0V$. In a process known as data remanence, SRAM retains its state as long as voltage is above its retention voltage (e.g., $0.4V$) [45]. Our experiments show that SRAM retains data for almost an hour after the microcontroller turns off (§3.2)—3000 times greater than common intermittent computation off times. Thus, **for the short off times common to intermittent computing, SRAM acts as a NVM—without Flash's write penalties**.

To demonstrate the ability of SRAM to serve as a low-overhead NVM for intermittent computation, we design and implement a library-level, lightweight, in-situ, one-time checkpointing approach called TotalRecall.[1] TotalRecall checkpoints are in-situ: SRAM data remains in place, while registers are checkpointed to SRAM. To handle worst-case off-times that expose SRAM's volatility, TotalRecall calculates a Cyclic Redundancy Check (CRC) over SRAM and adds it to the in-SRAM checkpoint. Upon recovery, TotalRecall verifies the CRC, falling back to an existing checkpointing approach in the event of a mismatch. We implement TotalRecall on both Flash- and FRAM-based microcontrollers common to intermittent computation. In experiments with benchmarks from Texas Instruments, **TotalRecall provides better-than-FRAM performance on Flash devices**—with over 99.999% worst-case reliability.[2] TotalRecall improves on the performance of state-of-the-art Flash-based one-time checkpointing between 230% and 37000%.

[1]TotalRecall stems from SRAM's ability to remember it's power-on state perfectly given short off times.
[2]Our experiments suggest that Flash-based checkpointing will cause Flash writes to fail (i.e., device failure) before a silent data corruption occurs with TotalRecall.



(a) Energy harvesting block diagram **(b)** Idealized supply voltage for energy harvester microcontroller. V_min is the minimum voltage for code execution.

Figure 1. Energy harvesting device block diagram and idealized power supply (V_{dd}). In a real system, the charge/discharge rate depends on the harvester's yield, the microcontroller's current draw, and the capacitor size. The blue highlighted regions indicate when the microcontroller computes.

This paper makes four technical contributions:

- We show that SRAM data remanence can safely store program state for stretching computation across short power cycles; previously, it had been shown to be a way to exfiltrate secret data [8, 9] and keep track of time [42, 45].
- We design TotalRecall, a library-level checkpointing technique that exploits SRAM data remanence. TotalRecall is NVM agnostic and supports existing software without modification (beyond linking against our library).
- We implement TotalRecall on both Flash- and FRAM-based MSP430 microcontrollers. Our evaluation using benchmarks from Texas Instruments shows correct operation even with up to 5-minute off times. TotalRecall boosts performance of Flash-based devices up to and beyond FRAM-based systems.
- We explore the performance/reliability trade space that TotalRecall exposes to designers for hardware and software CRC implementations as well as 16- and 32-bit CRC implementations. Our results show that hardware support for CRC maximizes TotalRecall's performance and reliability.

2 Background

Energy harvesting circuits output tiny amounts of energy relative to the demands of even ultra-low-power microcontrollers [10, 21]. A popular solution is to place a capacitor between the energy harvester and the microcontroller [41], as illustrated in Figure 1a. The energy harvester continuously pours charge into the capacitor, eventually increasing voltage enough to turn the microcontroller on. Once computation commences, the microcontroller drains charge faster than the harvester supplies it, causing the voltage to lower. Eventually, the capacitor's voltage drops below the microcontroller's operating voltage, forcing it to power down. This charge/discharge cycle repeats, as shown in Figure 1b. Correctly stretching long-running computation across frequent power cycles is known as intermittent computation. *This paper focuses on reducing the overhead of intermittent computation.*

2.1 Intermittent Computation

Supporting intermittent computation requires two things: (1) recover the program's pre-power-cycle state and (2) ensure the recovered program state is consistent. While meeting these two requirements is trivial with a wholly-non-volatile processor [36], all deployed energy harvesting platforms present a mix of volatility. Given some volatile memory (e.g., registers and the stack), checkpointing fulfills the first requirement by backing-up all volatile memory to NVM. Upon recovery, the volatile memory is restored from the checkpoint in NVM. Previous work shows that fulfilling the second requirement is more challenging [46]. In cases where execution continues post-checkpoint, a write to NVM risks creating an inconsistency upon recovery—because the state is not exactly as it was when the checkpoint was taken and where execution resumes. Inconsistent recovery opens the execution to producing semantically impossible (hence incorrect) results.

As §7 describes, there are two classes of checkpointing approaches, each ensuring consistent recovery in a different way. Because the more checkpoint heavy continuous checkpointing approaches are ill-suited for Flash devices due to its limitations (§2.3), we focus on one-time checkpointing approaches. One-time checkpointing approaches [1, 2, 28, 47] assume some voltage monitoring capability to detect when the energy storage capacitor has *just* enough energy to write a checkpoint before the microcontroller turns off. The microcontroller then stops computation to maintain recovery consistency. *This paper focuses on reducing the overhead of one-time checkpointing as used in intermittent computation.*

2.2 Microcontroller Memory Hierarchy

The microcontrollers deployed in energy harvesting devices [41, 48, 54] are scalar in-order processors. They employ a flattened memory hierarchy[3] consisting of two types of memory:

- **Static Random-Access Memory (SRAM)**: main memory: holds transient program data (e.g., heap and stack). Positives include fast reads and writes, low energy, and byte addressable. Negatives include small size (e.g., 512B) and volatility, i.e., it *eventually* loses state without power.

[3]FRAM devices support a hierarchical memory model where SRAM acts as a cache for FRAM [26]. This reduces volatile state to architected state. This organization decreases checkpointing overhead, but sacrifices common-case performance [29]. Thus, the default is a Flash-like, flattened, model.

	Flash [24]	**FRAM** [25]	**SRAM**
NVM size	256 KB	256 KB	
Cost	\$6.23	\$6.33	
SRAM	16 KB	8 KB	
Max freq.	25 MHz	16 MHz	
Dhrystone	8.59 ms	13.52 ms	
Released	2001	2013	
Read freq.	25 MHz	8 MHz	> 25 MHz
Write freq.	.036 MHz	8 MHz	> 25 MHz
Endurance	10^4	10^{15}	∞
Min V.	2.2 V	1.8 V	< .9 V [18]
Byte erase/program	No	Yes	Yes

Table 1. Comparison of Flash and FRAM microcontrollers from Texas Instruments, targeting the same NVM size and price point.

- **Non-Volatile Memory (NVM)**: permanent data store: holds stable program data (e.g., code and constants). The positive is its large size (e.g., 256KB). Negatives include being slower and higher energy than SRAM. Note that different NVMs exacerbate or diminish these trade offs.

2.3 NVM Choices

Given that checkpointing is a fundamental part of intermittent computation, NVM selection greatly impacts intermittent computation overhead. Commodity microcontrollers offer two choices of NVM: either Flash or FRAM. To compare and contrast Flash and FRAM with respect to energy harvesting, we select representative microcontrollers from a popular online electronics distributor. We select Texas Instruments MSP430 microcontrollers to facilitate cross-device performance comparison and because that is the microcontroller used by deployed energy harvesting platforms [41, 48, 54]. Since NVM size is the primary driver of cost and technology, we fix NVM size at 256KB. From the remaining microcontrollers, we select the wider voltage range (1.8–3.6V) as that enables longer on-times. From there, we limit the results to in-stock parts available in small quantities. For each NVM type, we select the cheapest device as the representative microcontroller. Table 1 highlights the tradeoffs between the representative microcontrollers, along with a comparison to SRAM.

Flash benefits: The green cells in Table 1 highlight where Flash outperforms FRAM. The most important benefit of Flash is its ubiquity: Flash-based MSP430s have a decade lead over FRAM-based MSP430s. Flash devices continue to outsell FRAM devices today as only one vendor sells FRAM-based devices, while dozens sell Flash-based variants. For example, the online distributor lists over 64,000 Flash-based microcontrollers, while only 777 options exist for FRAM variants. In addition to availability, Flash devices also have a performance advantage over FRAM-based devices. Specifically, Flash reads are 3x faster, provide 2x the SRAM, and a 62% higher clock frequency. These individual advantages add up to a 57% improvement in the Flash device's Dhrystone score over the FRAM device's. Thus, while one-time checkpointing approaches mesh well with FRAM, **ignoring high-performance solutions for Flash-based devices leaves most current and future deployed systems unserved**.

Flash drawbacks: Despite the availability and performance advantages of Flash, it has limitations that cause recent energy harvesting systems to move to FRAM-based devices [38, 41, 52, 54]. Flash's key disadvantage centers on writes. Programming Flash (i.e., writing) is a uni-directional, energy, and time-intensive process called hole punching. In hole punching, you increase the voltage of a Flash cell to force charge across a dielectric, changing the state of the cell from a 1 to a 0. This requires a higher starting voltage (e.g., $2.2V$) and sufficient time—and energy—to pump the voltage up to the required level (e.g., $12V$). Changing any cell's value back to a 1 requires an erase—which only occurs at segment granularity (e.g., 512B): updating a single word in Flash requires copying the entire segment to SRAM, erasing the segment in Flash, then writing the modified segment back to Flash. This makes Flash writes incredibly costly as Table 1 indicates that writing alone is 200x slower than FRAM.

Another write-related problem often overlooked in previous work is Flash's limited write endurance [23]. Each program/erase cycle ages the Flash cell, making it harder to program/erase in the future. This means that the frequent checkpoints required to support intermittent computation quickly kill Flash—taking the microcontroller with it. Thus, it is clear that without an alternative to checkpointing to Flash, the vast majority of microcontrollers will not support intermittent computation and the most advanced continuous checkpointing approaches are a non-starter. *The goal of this paper is to achieve FRAM-like checkpointing performance and lifetime on the more ubiquitous and performant Flash-based devices.*[4]

3 Motivation

To enable low-overhead and long-term intermittent computation on Flash-based devices, we exploit SRAM's data remanence to allow it to serve as a NVM. This approach stems from two observations: (1) the off times of intermittent computation are short[5] and (2) SRAM on microcontrollers retains data for a relatively long time. Given these observations, we see an opportunity to tradeoff unnecessarily long data retention guarantees for a increase in performance and lifespan for Flash-based intermittent computing devices.

[4] As emerging NVM technologies mature and approach Flash-based devices in terms of frequency, read latency, features, and availability, the whole-memory non-volatility guarantee provided by emerging NVM technologies is preferable to the added complexity of the mixed-volatility offered by Flash+TotalRecall.

[5] While many energy harvesting devices will go long periods without power (e.g., smart cards), we note that these long off times demarcate individual, unrelated, computations; we differentiate these workloads from those targeted by intermittent computation. Previous work makes a similar observation about temporal locality of results and intermittent computation [7].

Energy Source	**Source**	**Max Off Time (s)**	**SRAM Suitable** $10\mu F$, $47\mu F$
RF	[10, 36, 47]	10	< 80°C, ✓
Thermal	[36]	14	< 75°C, ✓
Piezoelectric	[36, 49]	2	✓, ✓
Solar*	[10, 36]	300	< 25°C, < 55°C

Table 2. Off times in various energy harvesting systems along with the maximum ambient temperature that affords 100% SRAM data retention. *Solar-powered systems experience longer off times at night, but this is (predictable) power loss—not intermittent computation. ✓ represents that the temperature is above the device's maximum operating temperature of 85°C.

3.1 Intermittent Off Times are Short

The first observation driving our approach is that the off times common to intermittent computation are short. To validate this observation we explore the off times of energy harvesting platforms across a range of energy sources. This task is complicated by the fact that previous work focuses on on-times due to its reliance on the long-term data retention guarantees of NVMs and because of the on-time's impact on checkpointing overhead. Fortunately, in providing on-time data, they also provide enough data to approximate off times. Table 2 presents a summary of the off times from different energy sources. From this summary, we make two observations: (1) intermittent off times tend to be of the same magnitude as on times—i.e., short—and (2) many long off times are predictable and incongruent with the goal of intermittent computation due to temporal locality. Thus the data retention guarantee of traditional NVMs is overkill for intermittent computation.

3.2 SRAM has Time-dependent Volatility

The second observation driving our approach is that after a system ceases computation due to power failure, the remanent charge keeps the device's voltage higher than SRAM's Data Retention Voltage (DRV) for some time. This is because SRAM DRV, which is ~$0.4V$ [16, 43], is much lower than the operational voltage of the microcontroller (e.g., $1.6V$ for the representative MSP430s). When supply voltage falls below the minimum operating voltage of the microcontroller, the power consumption drops drastically as transistors stop switching; power drain is now dominated by leakage current in the MCU and surrounding circuitry. While this leakage eventually reduces the supply voltage below the SRAM DRV, both extrinsic (i.e., decoupling capacitors and PCB trace capacitance) and intrinsic (e.g., transistor and trace capacitance) sources of capacitance prevent the supply voltage from dropping instantaneously. The energy storage capacitor described in §2 dominates the charge remanence effect; previous work on SRAM remanence supports this observation [45].

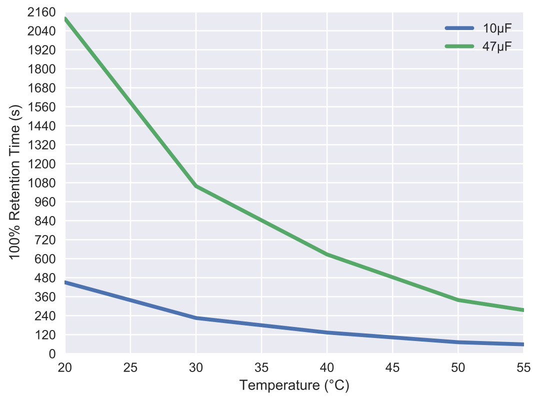


Figure 2. The time before the first SRAM cell in a microcontroller loses state varies with ambient temperature and capacitor size.

To empirically determine how long the remanent charge in microcontrollers allows SRAM to retain state after a power failure, we perform real-system experiments using a Flash-based MSP430. For this experiment, we first initialize all SRAM cells to a known value, then disconnect power for a set time, and finally, read back the SRAM data, checking for bit flips. Because SRAM fails bi-directionally—some cells tend to fail into a 0 state, while others fail into a 1 state—we run two trials at each off time, writing all 1's then all 0's [14]. We perform a binary search to determine the maximal off time where full SRAM state retention occurs.

Figure 2 shows the retention time of our MSP430 microcontroller across a range of temperatures and the two energy storage capacitor sizes common to deployed energy harvesting devices [41, 48]. We vary temperature (using a Test Equity 123H thermal chamber) and capacitance (via power rail decoupling capacitors) as those two variables dominate charge leakage and by extension SRAM data retention time. To contextualize these results, we add the maximum supported temperature for a given energy source and capacitor size combination in Table 2. In this paper, **we design and implement a system that reliably uses SRAM as a low overhead, long lifetime, NVM for the short off times common to intermittent computing**, falling back to existing checkpointing to support longer, power off, scenarios.

4 TotalRecall Design

We design TotalRecall, an efficient checkpointing technique for intermittent computing on energy harvesting devices. TotalRecall eliminates costly NVM checkpoints by reliably retaining program state in SRAM. The pervasiveness of SRAM makes TotalRecall deployable regardless of NVM technology. The library-based design of TotalRecall makes it deployable to current and future commercial off-the-shelf microcontrollers, without software or hardware modification. When deployed on Flash microcontrollers, TotalRecall enables system designers to take advantage of the benefits of Flash (Table 1), without its checkpointing-induced high write/erase overheads and limited lifetime. Much of TotalRecall follows from previous one-time checkpointing systems [1, 2, 28, 47], except that SRAM data remains in place. Keeping SRAM data in place creates the central challenge that we address in this section: how does TotalRecall know when SRAM acted as a NVM while the power was off?

4.1 Challenge: Detecting Volatility

Our experiments with SRAM remanence (§3.2) show that SRAM acts as a NVM given normal off times, but becomes volatile memory given sufficiently long off times. While experiments show that SRAM gradually transitions from completely non-volatile to completely volatile as the time off increases, we assume that any bit loss constitutes complete volatility. Thus, our goal is to design a mechanism to detect if any bit has changed during power loss. Below we explore the SRAM volatility detection design space.

Do nothing: The highest performance, but lowest reliability, solution is to assume SRAM retains all data. This incurs near-zero overhead (only registers are copied to SRAM) for off times that stay within SRAM's non-volatile range; but leads to program failure for unexpectedly long off times.

Canary: To add some cheap error detection, we could add a canary value to our in-SRAM checkpoint. By checking if the canary value changed at power on, we would know if SRAM lost state. Unfortunately, the SRAM cells that fail first and the direction of failure is dictated by manufacturing-time process variation—hence each device is different [14, 15]. Thus, a canary only detects SRAM state loss—not retention.

Enrollment: A viable solution that has a low run time overhead and is reliable is to pre-characterize devices to determine where to place canary values in each device's SRAM. Characterization follows the same binary search procedure of our remanence experiments (§3.2), but at a finer granularity (to tease-out the specific SRAM cell that fails first). Characterization works because SRAM cells lose state in a predictable order that is robust against temperature and voltage changes [12]. We do not choose this solution because it limits the generality of TotalRecall; otherwise, this is the optimal solution.

Redundancy: Another approach that takes advantage of the unique failure pattern of SRAM is dual modular redundancy: duplicate all words in SRAM and after powering on, check for disagreement. While this provides stronger reliability guarantees than doing nothing or using canaries without enrollment, redundancy does not provide significant guarantees in the face of frequent power failures. Additionally, this approach has high memory overhead.

		$10\mu F$		$47\mu F$	
	Temp. (°C)	16-bit	32-bit	16-bit	32-bit
Office	20	3.7	239K	17.2	1,126K
Death Valley	55	0.8	54K	3.9	256K

Table 3. Worst-case time, in years, until a silent data corruption.

ECC:. Error Correcting Codes (ECCs) are commonly used to protect DRAM memory. ECC is a poor fit for the bursty errors encountered by SRAM volatility. For example, the MSP430's word size is 16-bits; adding a parity bit to each data word incurs 6.25% overhead, but can only detect a single bit flip. Upgrading to 5-bits, guarantees two-bit error detection, but incurs a 31.15% overhead. In the average case, our design requires 4-bits per word of detection.

Hash function: The final solution that we rule out is employing a hash function. Using a hash function trades a huge performance loss for near-perfect reliability. To use a hash function, you would feed it the entire contents of SRAM and store the resulting digest in SRAM, as part of a checkpoint. Upon power on, you recalculate the digest and verify against the stored digest. The likelihood that the two digests match, but the SRAM lost state is less than 1 in 100 trillion. This reliability is overkill for the duty cycle of intermittent computation and expensive in terms of run-time overhead.

4.2 Our Solution: Cyclic Redundancy Checks (CRC)

Our goal is to maximize system performance without (practically) compromising data integrity. To this end, we find a solution that handles error bursts like hash functions, is designed with random errors in mind like ECC, but balances performance and integrity. Ideally, we want the cheapest solution that provides *just enough* data integrity. The solution that meets these requirements is a Cyclic Redundancy Check (CRC); CRCs are simple, fast, and provide tunable data integrity that allows system designers to trade performance for integrity. Another benefit of CRC is that, due to its prevalence (e.g., the WISP platform makes heavy use of CRC [48]), many microcontrollers provide hardware support for it.

Cyclic Redundancy Checks (CRCs) are error-detecting codes common in many communication and data storage applications to provide a high degree of confidence that a data packet is error-free. To verify a data packet, the CRC algorithm divides the data packet by a pre-determined generator polynomial using repeated shift and XOR operations. The sender stores/transmits the data packet along with the remainder of this division; when the data must be verified, the message is divided by the generator polynomial again. If the calculated remainder matches the stored remainder, the receiver can assume with a high degree of confidence that the message is correct. If the remainders do not match, the data is assumed to be corrupted. The shift and XOR operations used to calculate CRCs are simple to implement in both software and hardware.

Algorithm 1 In-situ SRAM checkpoint routine.

```
...Copy all CPU registers...
SRAM_Ptr = SRAM_BOTTOM
// CRC everything in SRAM but CRC value
while SRAM_Ptr ≠ SRAM_TOP do
    // Next input to CRC engine
    CRC_Input ← *SRAM_Ptr
    SRAM_Ptr = SRAM_Ptr + 2
end while
// Save CRC result to top of SRAM
*SRAM_TOP ← CRC_Result
...Power off...
```

Besides being efficient, CRCs have high error detection capability. A CRC is guaranteed to detect G bits of errors, where G depends (roughly) on the bit-width of the CRC; a 16-bit CRC guarantees detection of up to three flipped bits anywhere within the data, while a 32-bit CRC guarantees detection of up to five flipped bits [32]. For errors corrupting more bits than these, CRCs provide probabilistic error detection of $1/2^m$, where m is the width of the CRC. Such a multi-bit error is undetected only if the checksum of the corrupted and un-corrupted data match exactly. The probability of undetected corruption is further reduced because CRC guarantees detection of an odd number of bit flips and there is a 50% chance that SRAM fails in the direction of the value stored in the cell. Assuming each power failure is just long enough to corrupt data (see Figure 2), we calculate CRC's expected time to first undetected corruption for several energy storage capacitor sizes and CRC bit-widths. Table 3 presents the results of this calculation.

4.3 TotalRecall Overview

Like previous one-time checkpointing approaches [1, 2, 28, 47], TotalRecall creates a checkpoint immediately before power loss. TotalRecall performs all checkpointing actions in software by implementing them in an interrupt service routine associated with an interrupt-enabled voltage supervisor that monitors the system's energy storage capacitor. The checkpoint contains all architected state (e.g., general-purpose registers) and a CRC checksum computed from all other SRAM data. Note that the registers themselves are implemented using SRAM, thus they may retain data after power loss; however, TotalRecall still copies them as part of the checkpoint for generality, as some systems tie registers to hardware resets to ensure a known startup state. Unlike previous one-time checkpointing approaches, TotalRecall's checkpoints leave program data in-place in SRAM—eschewing costly Flash writes. After completing the checkpoint, the microcontroller powers-off to avoid inconsistent recovery [1, 46].

Algorithm 2 In-situ SRAM checkpoint recovery routine.

```
1:  SRAM_Ptr = SRAM_BOTTOM
2:  while SRAM_Ptr ≠ SRAM_TOP do
3:      // Next input to CRC engine
4:      CRC_Input ← *SRAM_Ptr
5:      SRAM_Ptr = SRAM_Ptr + 2
6:  end while
7:
8:  if *SRAM_TOP ≠ CRC_Result then
9:      // SRAM corruption detected
10:     ...restart or load NVM checkpoint...
11: end if
12:
13: ...Restore all CPU registers...
14: Restore stack pointer
15: ...execute initialization callback, if necessary...
16: Restore status register
17: Restore program counter
```

Upon power-on, TotalRecall's recovery routine gets invoked. It first recomputes the CRC over SRAM and compares it to the recorded checksum. In the common case when the old and new checksums match, TotalRecall restores the checkpointed register values from SRAM and then resumes program execution. In the uncommon case that the checksums do not match, TotalRecall restarts the user program from the beginning or from a conventional checkpoint in NVM. While we envision more sophisticated ways of handling long off times, that is not the focus of this paper.

4.4 Checkpoint Layout and Creation

TotalRecall's checkpointing procedure stores the checkpoint in a static location reserved at the top of SRAM. This simplifies CRC processing, makes the checkpoint and restore code more efficient, and enables TotalRecall to seamlessly integrate with existing programs at link time. This reserved address range is small (e.g., 40 bytes) because TotalRecall's checkpoint only contains register values, which are few in number, and a checksum. To prevent the checkpoint content from being inadvertently overwritten by the user program, we modify the memory map used by the linker script to start the stack just after our checkpoint location. By doing this, TotalRecall is able to integrate with existing programs as late as the linker. Algorithm 1 provides the details of TotalRecall's checkpointing routine.

4.5 Restoring from Checkpoints

On power up, TotalRecall's recovery routine executes before any platform initialization or user code executes. Algorithm 2 details TotalRecall's recovery routine. TotalRecall first re-calculates the checksum over SRAM (excluding the CRC value) and compares it to the checksum stored at the top of the in-SRAM checkpoint. When the checksums match, TotalRecall repopulates the register file, restores the stack pointer, and executes any peripheral initialization function that the user program registered (by writing its function pointer to a reserved space in the checkpoint area), finally passing control to where the program left off (by restoring the status register and the program counter).

Checksum mismatches result from two situations: (1) no checkpoint exists (i.e., this is the first power on event) or (2) the data in SRAM was lost during an a unexpectedly[6] long power cycle. In the first case, TotalRecall starts execution from the beginning of the program. In the second case, TotalRecall looks for a checkpoint in Flash, restoring it if it exists, otherwise restarting the program.

Platform	MSP430G2553	MSP430FR6989
NVM Type	Flash	FRAM
NVM Size	16 KB	128 KB
SRAM Size	512 B	2048 B
Max Clock Frequency	16 MHz	16 MHz
CRC Engine	No	Yes

Table 4. Microcontrollers that we implement TotalRecall on.

5 TotalRecall Implementation

To validate the applicability of TotalRecall to energy harvesting devices and intermittent computation, we implement and evaluate it on two Texas Instruments microcontrollers: MSP430G2553 and MSP430FR6989. Table 4 summarizes the relevant aspects of each device. From a high level, these devices represent existing energy harvesting devices [41, 48, 54] and cover both Flash and FRAM NVM options. From a low level, we select these specific devices because they are available as part of development boards.

Power control board: The development boards have the same debug interface, which enables us to use the same power control board across both boards. We approximate energy harvesting conditions using a custom daughter board compatible with both MSP430 development boards, shown in Figure 3. Testing intermittent computation systems is difficult because energy harvesting environments are, by nature, unpredictable and difficult to replicate [10]. The power control board allows us to test TotalRecall and baseline systems in a consistent, repeatable environment. A microcontroller on the daughter board controls a Digital-to-Analog Converter (DAC) capable of powering the MSP430-under-test and transistors to isolate the MSP430-under-test when power is disconnected to better imitate energy harvesting circuit behavior.

Baseline implementation: To serve as a baseline for our evaluation, we implement the state-of-the-art one-time checkpointing approach (i.e., Hibernus [1]) on both microcontrollers. Being one-time approaches, the systems take a checkpoint when triggered by the interrupt indicating imminent

[6]For expected long off times (e.g., sunset with a photovoltaic), write a checkpoint to Flash instead.

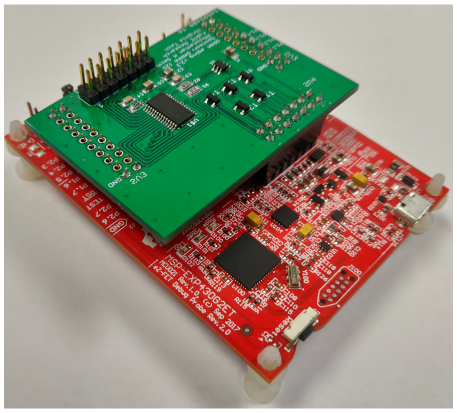


Figure 3. The experimental setup with power-control daughter board (top) and the MSP430G2553 Launchpad (bottom).

power loss. A checkpoint consists of writing all registers and SRAM data to a static, reserved, location in NVM. Compare this to TotalRecall's checkpoints, which leave SRAM data in place, and only copies registers.

5.1 CRC Implementation

The main question that we answer in the evaluation (§6) is whether taking a CRC is more efficient than copying SRAM to NVM. TotalRecall incurs near-zero data transfer overhead because it does not access slower NVM and only needs to copy register data (40 bytes) to SRAM. Instead, the primary source of overhead is calculating the CRC checksum of SRAM data. We evaluate four different implementations of the SRAM integrity check: two table-based software routines applicable to any device, and two routines taking advantage of hardware CRC support common to modern microcontrollers.

Software CRC16/32: The software CRC approach is independent of hardware support, thus applicable to any system. Our software CRC implementation is optimized for speed based on a memoization of CRC divisions; platforms with limited code space could sacrifice checkpoint and recovery speed for reduced code size by removing the CRC table and directly calculating the CRC remainder. Algorithm 3 shows the software CRC16 algorithm, adapted from Texas Instruments example code [22]. Not shown is a software implementation of CRC32. Software CRC32 is similar to CRC16, but with added operations to deal with 32-bit numbers on a 16-bit microcontroller. Experiments show that CRC32 takes roughly 1.6*x* the time of CRC16—but increases reliability (Table 3).

Hardware CRC16/32: Many low-power microcontrollers which transmit or receive data include hardware dedicated to calculating CRC checksums. The programmer writes data to the input register of the CRC engine, sequentially, until

Algorithm 3 Software CRC16 routine.

```
1:  MASK ← 0xFF00
2:  CRC ← 0x0
3:  P_TABLE ← &CRC_TABLE
4:  P_SRAM ← SRAM_BOTTOM
5:  while P_SRAM ≠ SRAM_TOP-2 do
6:      INDEX ← CRC[7:0]
7:      INDEX ← *P_SRAM xor INDEX
8:      P_SRAM ← P_SRAM + 1
9:      INDEX ← INDEX + INDEX
10:     INDEX ← INDEX + P_TABLE
11:     CRC ← CRC and MASK
12:     CRC ← *INDEX xor CRC
13: end while
14: return CRC
```

all data is processed; the updated checksum is available the next CPU cycle. We implement TotalRecall using both the CRC16 and CRC32 engines available on the MSP430FR6989 to measure the overhead improvement possible when CRC hardware is available. Our results show that hardware acceleration increases checkpoint/recovery speed by 342% when compared to the software implementation of the CRC16 and 561% compared to the software CRC32.

5.2 Additional Challenges

We encountered a number of challenges implementing our design, primarily related to circumventing embedded system startup routines that assume no valuable data is in SRAM immediately after power-on. Startup routines such as `crt0` (responsible for setting up the C runtime environment) make function calls and allocate variables at the top of the stack because they run before anything else on the system. Because the system writes checkpoints to a static location at the top of the stack, TotalRecall must take care to avoid overwriting `crt0` and other runtime data during the checkpointing process.

Rather than re-run the runtime initialization code on every startup (which increases checkpoint recovery overhead) or re-instrument startup code to accomodate SRAM checkpoints (which increases time to deployment because startup routines are platform-specific), we modify the linker script to make the space used for storing checkpoint data unavailable to the compiler. This reduces the total available RAM space by the size of the checkpoint even if no checkpoints are taken; we consider this an acceptable penalty because register file checkpoints are small.

6 Evaluation

To validate TotalRecall's effectiveness and compare against the state-of-the-art one-time checkpointing approach, we evaluate TotalRecall and Flash-based one-time checkpointing using a set of benchmarks written for the MSP430 by Texas Instruments [17]. These benchmarks, shown in Table 5, include common embedded system applications and

Benchmark	Size (B)	SRAM Usage (B)	Approximation?
FIR Filter	3572	254	Yes
Dhrystone	1285	474	No
Whetstone	16136	468	Yes
Quicksort	864	362	No
Factorial	1532	44	No
Matrix Mult.	1536	122	Yes

Table 5. Memory usage information for several benchmarks on the MSP430FR6989.

CPU performance benchmarks. We also include `quicksort` and `factorial` benchmarks to test against stack-intensive programs. Table 5 includes the memory usage of the benchmarks, because one potential optimization for TotalRecall is to only compute the CRC over SRAM in use. This optimization is especially useful for programs that do not use dynamic memory. Table 5 also details which benchmarks are amenable to approximate computing techniques as we foresee that TotalRecall's use of SRAM as best-effort approximate storage has potential synergy with approximate computing systems, although we leave this line of research for future work.

We compile all benchmarks using the open-source MSP430 GCC toolchain developed by Texas Instruments [27] with -Os (optimization for size) enabled. We use the results of this evaluation to answer the following questions:

1. Does TotalRecall correctly stretch program execution across short yet frequent power cycles?
2. Can TotalRecall detect and handle when unexpectedly long power cycles expose SRAM's volatility?
3. How does TotalRecall's run-time overhead compare to existing one-time checkpointing approaches?

6.1 Correctness

Figure 4 shows a test of the system working in a typical energy harvesting environment. The supply voltage to the microcontroller varies as TotalRecall extends the `quicksort` benchmark across five power cycles on the MSP430G2553. Figures 4 and 5 illustrate three stages of the power failure the microcontroller experiences; these traces are taken on hardware using the Picoscope 2208B Mixed Signal Oscilloscope (MSO) [50]. Before the device disconnects from the power source, the supply voltage is steady at $3.3V$. Immediately after disconnecting power, the microcontroller continues execution, but rapidly drains the energy storage capacitor until V_{dd} reaches the brown-out voltage ($1.5V$). With the microcontroller not executing, V_{dd} drops slowly due to charge leakage from the microcontroller and surrounding development board. The rate of V_{dd} drop can be unpredictable depending on the behavior of power-hungry peripherals such as radios that share the power supply with the MCU. Hardware-oriented solutions such as energy federation [11] can mitigate this problem by intelligently choosing when to

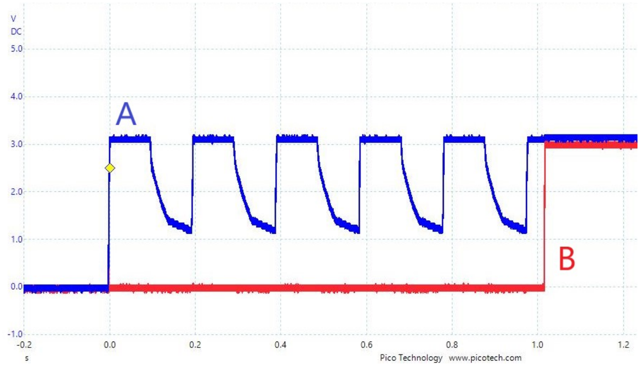


Figure 4. Extending the `quicksort` benchmark across five power cycles. Channel A shows supply voltage and channel B shows the status of a GPIO pin that is driven high when the program completes with correct output.

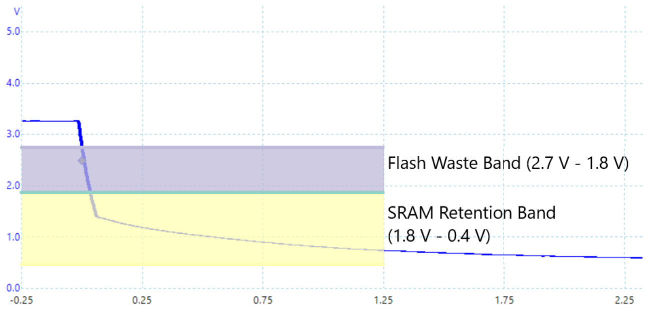


Figure 5. MSP430G2553 V_{dd} decay after power is disconnected. Flash checkpointing systems must begin writing a checkpoint well above the minimum Flash write voltage, wasting energy. TotalRecall allows computation to continue until just before V_{dd} reaches the brown-out voltage and uses the remaining energy in the system to retain SRAM state.

connect peripheral supply capacitors to the MCU power rail, reducing their impact on SRAM remanence time. This gradual drop in supply voltage allows data to remain in SRAM as it never goes below SRAM's data retention voltage. Thus, in the case of Figure 4, `quicksort` successfully completes execution after five power cycles. We perform similar experiments with all benchmarks, validating TotalRecall's ability to stretch execution across frequent power cycles. We conduct all experiments at 20°C to reduce the impact of operating temperature variation on SRAM retention time.

What about when longer power cycles cause data loss? To show that TotalRecall can detect and handle cases where SRAM becomes volatile, we create data loss by driving V_{dd} below SRAM's data retention voltage. This causes some SRAM cells to lose their state. We perform variations of this experiment down to $0V$ and by disconnecting from power for a long time, instead of directly driving V_{dd}. In all cases, the recovery routine detects the data corruption and restarts the program from the beginning rather than continuing with faulty data.

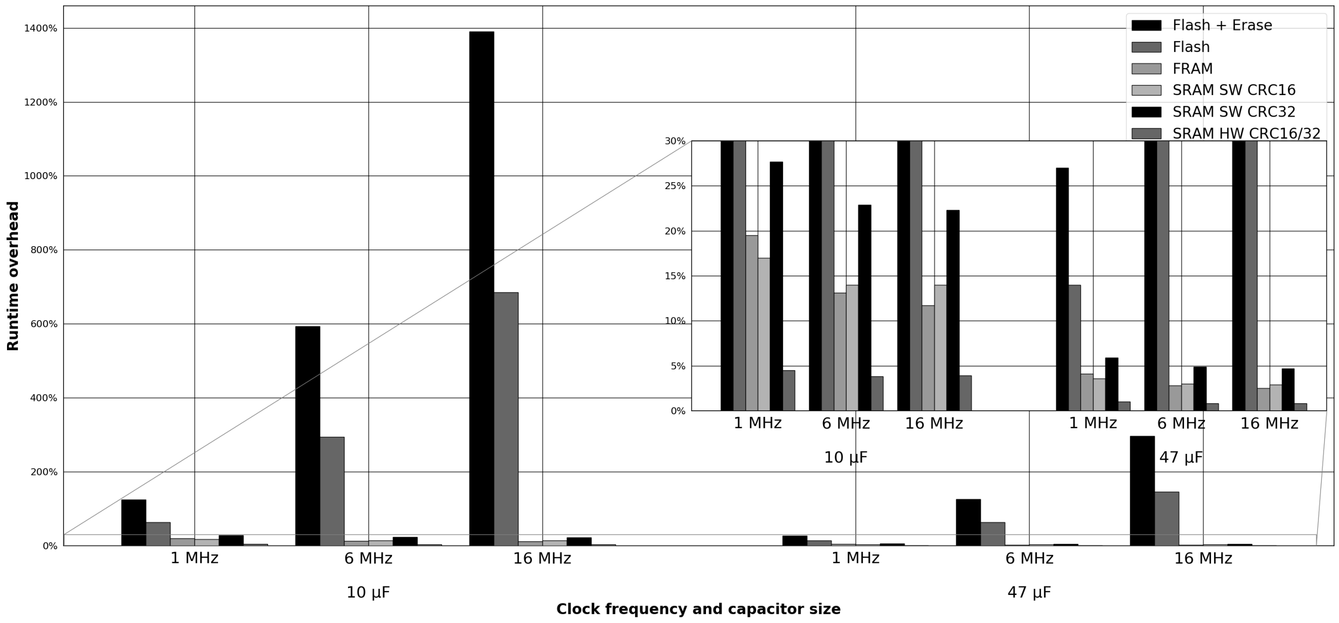


Figure 6. Comparison of one-time checkpointing system overheads on platforms with different energy storage capacitor sizes and clock frequencies. Flash+Erase represents overhead when the Flash page must be erased before writing the checkpoint; SRAM-based checkpoint systems are divided by CRC implementation (hardware or software) and bit-width.

6.2 TotalRecall's Overhead

Execution-time Overhead: TotalRecall correctly extends program execution across power cycles, but how does its performance compare to state-of-the-art Flash-based checkpointing solutions? Does it approach FRAM's overhead? To compare TotalRecall's overhead to Flash- and FRAM-based checkpointing systems, we time each checkpoint recording/recovery routine on the experimental hardware setup described in §5. Figure 6 shows these results as percentages of the total on-time spent in the checkpointing and recovery routines at each capacitor, frequency, and device combination. An overhead result >=100% indicates that the checkpointing system does not support useful computation in the given configuration, because the time needed to record and recover is greater than the available time in a single power cycle. In a system with a 47μF capacitor operating at $1MHz$, the hardware CRC implementation of TotalRecall extends program execution with an average overhead of 1.0% compared to the 27% overhead of the `Flash+Erase`[7] system under the same conditions—a 96% reduction in overhead. TotalRecall also scales better with clock speed, supporting $16MHz$ execution with an overhead of 0.8%, while the fixed-time Flash write+erase incurs nearly 300% overhead. In summary, not only does TotalRecall outperform Flash-based checkpointing, **it enables intermittent computation on a wider array of system configurations**.

[7]Flash-based systems must spend time and energy to erase data. They can do it every power cycle if there is sufficient energy (`Flash+Erase`) or they devote an entire power cycle to an erase operation when there is insufficient energy to both erase and write in a single power cycle. Thus, real deployments of Flash-based checkpointing systems are not able to achieve the overheads shown in `Flash` due to the need to eventually erase.

Checkpoint Voltage Guard Bands: In addition to faster checkpoint and recovery routines, TotalRecall reduces overhead by increasing the total charge available for useful calculations. One-time checkpointing systems use the concept of a voltage "guard band", which designates the minimum supply voltage at which the system must stop useful calculations and begin writing the checkpoint to ensure that the checkpoint is completely written before power is lost. Figure 5 shows a trace of the MSP430G2553 supply voltage during a power failure. Flash checkpointing systems must finish writing a checkpoint before V_{dd} reaches $2.2V$, the minimum voltage required to guarantee successful Flash writes on MSP430-family devices [19]. Because Flash writes are slow, the Flash checkpoint routine must begin well before V_{dd} reaches $2.2V$. For example, on similar hardware, the Flash-based approach Mementos [47] sets the beginning of the checkpoint guard band between $2.8V$ and $2.6V$. Given that the MSP430G2553 guarantees code execution between $3.6V$ and $1.8V$, Flash checkpointing systems waste 45% to 55% of the otherwise usable charge held by the capacitor.

TotalRecall maximizes energy use because it is not limited by the $2.2V$ Flash write minimum supply voltage; it exploits the microcontroller's full voltage range. Based on our test platform of the MSP430G2553 with the stock 11μF capacitor running at 1MHz, we use Equation 1 to determine that TotalRecall can delay taking a checkpoint until V_{dd} reaches 1.837V. This leaves 98% of the total useful capacitor charge available to the system for useful computation.

$$V_{guard} = \left(\overbrace{\Delta t}^{\text{Checkpoint time}} * \underbrace{I}_{\text{Active current}} / \overbrace{C}^{\text{Total capacitance}} \right) + \underbrace{V_{low}}_{\text{Min. voltage}} \qquad (1)$$

TotalRecall on FRAM Platforms: FRAM and other emerging NVM technologies such as Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) are potential alternatives to Flash on intermittent computing platforms due to their lower write currents, higher write endurance, and lower write latency. The state of the art for one time checkpointing systems uses FRAM-based microcontrollers that enable NVM-based checkpoints with approximately 15-20% time and energy overhead [1, 2]. We implement TotalRecall on the FRAM-based MSP430FR6989 to compare its performance to FRAM-based checkpoints. Our evaluation in Figure 6 shows that TotalRecall, with a software-based CRC implementation, enables checkpointing performance competitive with state-of-the-art FRAM checkpointing. On a system with a 10μF capacitor operating at 1MHz, TotalRecall using a software CRC32 implementation introduces 27.7% overhead compared to FRAM's 19.5% overhead. However, TotalRecall's capacity for hardware acceleration allows it to outperform FRAM-based checkpoints on devices that include hardware CRC support: on the same system, but using hardware CRC support, TotalRecall incurs 4.5% overhead—a 77% reduction. These results highlight TotalRecall's potential for high speed, NVM-agnostic checkpoints.

6.3 TotalRecall Practical Considerations

To gauge TotalRecall's practical impact on intermittently executed programs, we determine the number of power cycles required to complete a benchmark program running on the MSP430G2553. We model the total active time available as the time required to drain a capacitor from the maximum operating voltage (3.6V) to the minimum voltage (2.2V for Flash and 1.8V for SRAM) while drawing the typical active mode current at the specified clock frequency. Figure 7 depicts the number of power cycles required to complete the benchmark, normalized to the power cycles required if there were no checkpointing overhead (i.e., 100% of the active time was dedicated to running the user program). Operating at 1MHz with a 10μF capacitor, TotalRecall reduces the number of power cycles (and thus the total energy) required to complete the benchmark by over 7x compared to Flash-based checkpointing, while enabling intermittent computation at higher clock speeds and with smaller energy storage capacitors.

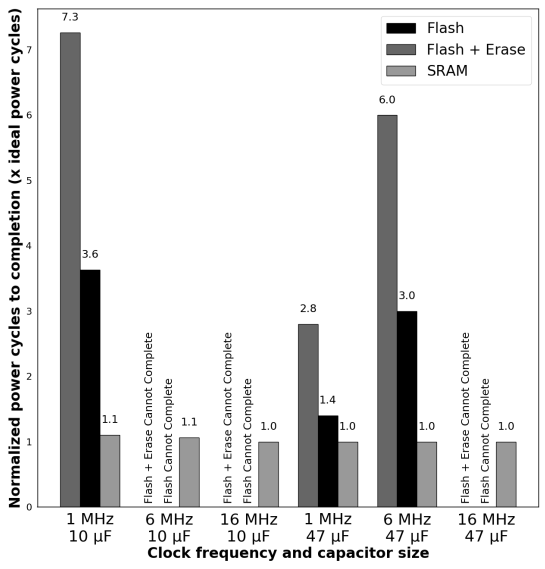


Figure 7. Power cycles required to complete the `FIR` benchmark normalized to continuous execution. `Flash+Erase` represents erasing every other power cycle.

7 Related Work

TotalRecall represents a shift in how intermittent computation systems maintain state across power cycles. All previous work, because it assumes instant loss of volatile state, involves storing, in the form of a checkpoint, volatile state (e.g., registers and SRAM) to non-volatile memory (e.g., Flash and FRAM). Two classes of checkpointing approaches exist: one-time checkpointing approaches that store all volatile state just before power loss and continuous checkpointing approaches that are power-failure-agnostic and make many, smaller checkpoints that ensure consistent recovery. In this section, we discuss the progression of advancement in each approach class, including a look at work that extends intermittent computation beyond the microcontroller.

7.1 One-time Checkpointing

Mementos [47] is the first system to tackle the problem of stretching computation across frequent power cycles. Mementos relies on periodic measurements of the system's energy storage capacitor, coupled with an energy model, to estimate how much energy remains. The goal is to commit a checkpoint just before power failure. While Mementos works well when programs treat non-volatile memory as read-only, follow-on work exposes state inconsistency when programs modify non-volatile state during post-checkpoint execution [46]. The problem is that Mementos allows for uncheckpointed work to occur. If uncheckpointed work updates both volatile and non-volatile memory, only the non-volatile memory updates persist, creating an inconsistent state upon recovery.

QuickRecall [28] addresses Mementos's correctness issue by storing all program data in non-volatile memory; effective, but expensive due to non-volatile memory's speed limitations. Hibernus [1, 2] solves the problem in a different way: through the introduction of guard bands and hibernation. A guard band is a voltage threshold that represents the amount of energy required to store the largest possible checkpoint to non-volatile memory in the worst-case device and environmental conditions. Execution occurs only when voltage is above the guardband threshold. Beyond correctness, Hibernus also improves upon Mementos in terms of performance. To avoid the overhead of polling the voltage of the energy storage capacitor's voltage and the risk of energy estimation, Hibernus leverages the Analog-to-Digital Converter's (ADC) interrupt mechanism. Hibernus configures the ADC to fire an interrupt when voltage dips below a the guard band threshold.

As Hibernus represents the most correct and highest performance one-time checkpointing system, TotalRecall uses it as the baseline for comparison. As our evaluation shows (§6.2), for the FRAM-based systems targeted by Hibernus and QuickRecall, guardbands are reasonable due to the low time and energy cost of FRAM-based checkpoints. But, the time and energy costs of Flash-based checkpoints make adapting Hibernus's approach to the Flash-based systems targeted by Mementos prohibitively expensive. TotalRecall, by keeping checkpoints in-place in SRAM, enables it to provide improved performance—with smaller guardbands—for both Flash- and FRAM-based intermittent computation systems.

7.2 Continuous Checkpointing

The more recent continuous checkpointing systems eschew taking a single, large, checkpoint, for many, smaller, checkpoints. The driving observation that underlies such approaches is that the short on-times of intermittent computation limits the amount of state changed during a power cycle. Thus, the checkpoint needed to track such a change is also small. The challenge is avoiding the incorrectness of Mementos that occurs due to post-checkpoint execution [46]. To avoid state inconsistency during recovery, continuous checkpointing systems must track program execution at a fine-grain, inserting checkpoints where consistency demands. Researchers approach this problem from two directions: compiler tracking and hardware tracking. Note that no matter the execution tracking approach, continuous checkpointing is incompatible with Flash-based devices due to the time and energy overheads of Flash writes/erases and Flash's limited write endurance.[8]

7.2.1 Compiler-directed. DINO [34], is a software-only, programmer-driven checkpointing scheme for intermittent computation. With DINO, programmers write programs using annotations that define a series of independent tasks, backed by volatile data versioning (i.e., checkpointing) to achieve recovery consistency. By doing this, DINO provides the notion of task-based atomicity. Chain [5] extends Dino with a more well-defined data passing interface between tasks that reduces overhead through checkpoint size reduction at the expense of requiring more complex programmer reasoning about possible data interfaces. Alpaca [37] extends DINO and Chain through the dynamic privatization of statically-identified inter-task data. Alpaca detects shared data using idempotence analysis and copies only the identified data into a private per-task buffer. This further reduces log/checkpoint size to just the buffered data—ignoring all volatile state.

Task-based intermittent systems afford a low overhead, but required programmers to reason correctly and statically about the effects of power loss. An automatic alternative is Ratchet [52]. Ratchet is a compiler that decomposes unmodified code into a series of checkpoint-connected idempotent computations. Tracking idempotency allows Ratchet to add overhead only on the subset of non-volatile memory writes that are critical for recovery consistency. Chinchilla [38] improves upon Ratchet with guidance from a smart timer and basic-block-level energy estimation. Like Ratchet, Chinchilla's compiler inserts the checkpoints required to make correct forward progress with very short on-times. Like Mementos, Chinchilla includes a timer-based runtime component that deactivates checkpoints when there is enough energy to make it to the next checkpoint. In many cases, this eliminates 99% of Ratchet's checkpoints, while maintaining correct execution.

7.2.2 Hardware-directed. Idetic [40] is the first hardware-based system for intermittent computation. Idetic takes applications, creates a hardware circuit from them using existing high-level synthesis tools, and inserts non-volatile checkpoints in the resulting circuit's control-and-data-flow graph. While Idetic works for simple applications and single-function hardware, it is not general purpose. Non-volatile processors [35, 36] generalize the Idetic approach by incorporating non-volatility into existing processor pipelines (e.g., via non-volatile flip-flops). Recent work mixes-in approximation to improve performance [7] for applications that benefit from partial results. Lastly, in an approach closer to compiler-directed approaches than to other hardware-directed approaches, Clank [13] inserts in-hardware idempotence monitors in the memory hierarchy to better identify idempotence violations (compared to Ratchet) and to buffer idempotence-breaking writes to maximally delay checkpoints. Elastin [3] extends on the idea of Clank, but at page granularity, which is used by more complex systems.

[8]Even with systematically elided checkpoints [38], continuous checkpointing induces Flash write failure in less than a week [23].

7.3 Checkpointing Beyond MCUs

While the focus of most intermittent computation research targets decreasing checkpointing overhead, other important problems exist. In the first line of work the focus is on peripherals. Most intermittent computation deployments involve sensors, thus depend on peripherals. One hidden problem of peripherals is that initialization often takes longer than a single power cycle [6]. Samoyed addresses this problem with a one-time checkpointing system that runs user-annotated peripheral functions by disabling checkpoints and undo-logging [39]. Samoyed guarantees progress by energy profiling, dynamic peripheral workload scaling, and a user-provided software fallback routine. The second line of work focuses on extending intermittent computation to x86-class processors. Work in this direction includes architectural support for out-of-order superscalar processors [36] and compiler-based techniques that adapt to x86-class systems [4, 33, 44].

8 Conclusion

TotalRecall exploits the time-dependent volatility of Static Random-Access Memory (SRAM) to eliminate the need for expensive checkpoints to Flash-based non-volatile memory for the off times common to intermittent computation. To address the central challenge of identifying if SRAM acted as a non-volatile memory during the off time, TotalRecall uses a Cyclic Redundancy Check (CRC) before and after a power cycle to validate SRAM data integrity. Our evaluation on real hardware with unmodified benchmarks, shows that, compared to the state-of-the-art one-time checkpointing approach applied to Flash-based systems, TotalRecall increases performance by up to 370x, while dramatically increasing system lifetime. On microcontrollers that have hardware support for CRC, performance surpasses FRAM-based checkpointing.

These results show that is is possible to have better than FRAM performance on the more widely deployed, more available, and higher performance Flash-based systems. Beyond validating TotalRecall's approach to supporting intermittent computation, these results also open the door to a wave of future approaches that leverage the synergy between the relatively short off-times common to intermittent computing and SRAM's time-dependent volatility.

Acknowledgements

We thank our shepherd Brandon Lucia for his guidance and the anonymous reviewers for their helpful suggestions. 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.

References

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

[2] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. 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 (March 2016), 1968–1980.

[3] J. Choi, H. Joe, Y. Kim, and C. Jung. 2019. Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 331–344.

[4] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler Directed Speculative Intermittent Computation. In *International Symposium on Microarchitecture (MICRO).* 399–412.

[5] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 514–530.

[6] John H. Davies. 2008. *MSP430 Microcontroller Basics* (1 ed.). Elsevier Ltd.

[7] K. Ganesan, J. San Miguel, and N. Enright Jerger. 2019. The What's Next Intermittent Computing Architecture. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).* 211–223.

[8] Peter Gutmann. 2001. Data Remanence in Semiconductor Devices. In *USENIX Security Symposium (USENIX Security).*

[9] C. Helfmeier, C. Boit, D. Nedospasov, and J. P. Seifert. 2013. Cloning Physically Unclonable Functions. In *International Symposium on Hardware-Oriented Security and Trust (HOST).* 1–6.

[10] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems.*

[11] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *ACM Conference on Embedded Networked Sensor Systems (SenSys).* 5–16.

[12] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. 2016. Persistent Clocks for Batteryless Sensing Devices. *ACM Transactions on Embedded Computer Systems* 15, 4, Article 77 (Aug. 2016), 77:1–77:28 pages.

[13] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *International Symposium on Computer Architecture (ISCA).* 228–240.

[14] D. E. Holcomb, W. P. Burleson, and K. Fu. 2009. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Comput.* 58, 9 (Sept. 2009), 1198–1210.

[15] Daniel E. Holcomb, Amir Rahmati, Mastooreh Salajegheh, Wayne P. Burleson, and Kevin Fu. 2012. 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).* 165–179.

[16] G. Huang, L. Qian, S. Saibua, D. Zhou, and X. Zeng. 2013. An Efficient Optimization Based Method to Evaluate the DRV of SRAM Cells. *IEEE Transactions on Circuits and Systems I: Regular Papers* 60, 6 (June 2013), 1511–1520.

[17] Texas Instruments. 2006. MSP430 Competitive Benchmarking. (July 2006). https://people.eecs.berkeley.edu/~boser/courses/40/labs/docs/microcontroller%20benchmarks.pdf.

[18] Texas Instruments. 2010. MSP430L092—MSP430L092, MSP430C09x Mixed-Signal Microcontrollers. (September 2010). http://www.ti.com/

lit/ds/symlink/msp430l092.pdf.
[19] Texas Instruments. 2013. MSP430x2xx Family User's Guide (Rev. J). (2013). http://www.ti.com/lit/ug/slau144j/slau144j.pdf.
[20] Texas Instruments. 2014. FRAM FAQs. (January 2014). http://www.ti.com/lit/ml/slat151/slat151.pdf.
[21] Texas Instruments. 2015. MSP432P4xx SimpleLink Microcontrollers Technical Reference Manual. (March 2015). http://www.ti.com/lit/ug/slau356i/slau356i.pdf.
[22] Texas Instruments. 2018. CRC Implementation with MSP430™MCUs. (2018). http://www.ti.com/lit/an/slaa221a/slaa221a.pdf.
[23] Texas Instruments. 2018. MSP430 Flash Memory Characteristics (Rev. B). (2018). http://www.ti.com/lit/an/slaa334b/slaa334b.pdf.
[24] Texas Instruments. 2018. MSP430F5438A—MSP430F543xA, MSP430F541xA Mixed-Signal Microcontrollers. (September 2018). http://www.ti.com/lit/ds/symlink/msp430f5438a.pdf.
[25] Texas Instruments. 2018. MSP430FR5964—MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. (August 2018). http://www.ti.com/lit/ds/symlink/msp430fr5964.pdf.
[26] Texas Instruments. 2018. MSP430FR698x(1), MSP430FR598x(1) Mixed-Signal Microcontrollers. (2018). http://www.ti.com/lit/ds/symlink/msp430fr6989.pdf.
[27] Texas Instruments. 2019. MSP430 GCC. (June 2019). http://www.ti.com/lit/ug/slau646e/slau646e.pdf.
[28] 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 *International Conference on VLSI Design and International Conference on Embedded Systems.*
[29] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2016. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID).*
[30] Joseph Kahn, Randy Katz, and Kristofer Pister. 1999. Next Century Challenges: Mobile Networking for "Smart Dust". In *Conference on Mobile Computing and Networking (MobiCom).*
[31] Udo Karthaus and Martin Fischer. 2003. Fully Integrated Passive UHF RFID Transponder IC With 16.7-μ W Minimum RF Input Power. In *IEEE Journal of Solid-State Circuits, Volume 38.*
[32] P. Koopman and T. Chakravarty. 2004. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks, 2004.* 145–154.
[33] Q. Liu, C. Jung, D. Lee, and D. Tiwari. 2016. Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery *(SC).* 228–239.
[34] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Conference on Programming Language Design and Implementation (PLDI).* 575–585.
[35] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Sampson, and Vijaykrishnan Narayanan. 2016. Nonvolatile Processor Architectures: Efficient, Reliable Progress with Unstable Power. In *IEE Micro Volume 36, Issue 3.*
[36] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).* 526–537.
[37] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 96:1–96:30.
[38] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *USENIX Conference on Operating Systems Design and Implementation (OSDI).* 129–144.
[39] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-time Checkpoints. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 1101–1116.
[40] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. 2013. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *International Conference on Pervasive Computing and Communications (PerCom).* 216–224.
[41] University of Washington. 2014. WISP 5 GitHub. (April 2014). http://www.github.com/wisp/wisp5.
[42] Yossef Oren, Ahmad-Reza Sadeghi, and Christian Wachsmann. 2013. On the Effectiveness of the Remanence Decay Side-channel to Clone Memory-based PUFs. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES).* 107–125.
[43] Huifang Qin, Yu Cao, Dejan Markovic, Andrei Vladimirescu, and Jan Rabaey. 2004. SRAM Leakage Suppression by Minimizing Standby Supply Voltage. In *International Symposium on Quality Electronic Design (ISQED).* 55–60.
[44] Qingrui Liu and Changhee Jung. 2016. Lightweight Hardware Support for Transparent Consistency-Aware Checkpointing in Intermittent Energy-Harvesting systems. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA).*
[45] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne Burleson, and Kevin Fu. 2012. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *USENIX Security Symposium.*
[46] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Workshop on Memory Systems Performance and Correctness.*
[47] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS).*
[48] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (Nov 2008), 2608–2615.
[49] Henry Sodano, Gyuhae Park, and Daniel Inman. 2004. Estimation of Electric Charge Output for Piezoelectric Energy Harvesting. In *Strain, Volume 40.*
[50] Pico Technology. 2016. PicoScope 2000 Series. (2016). https://www.picotech.com/download/datasheets/picoscope-2000-series-data-sheet-en.pdf.
[51] Priya Thanigai. 2011. FRAMs as alternatives to flash memory in embedded designs. (July 2011). https://www.embedded.com/design/mcus-processors-and-socs/4390688/2/FRAMs-as-alternatives-to-flash-memory-in-embedded-designs.
[52] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 17–32.
[53] Yu-Chi Wu, Pei-Fan Chen, Zhi-Huang Hu, Chao-Hsu Chang, Gwo-Chuan Lee, and Wen-Ching Yu. 2009. A Mobile Health Monitoring System Using RFID Ring-Type Pulse Sensor. In *Conference on Dependable, Autonomic, and Secure Computing (DASC).*
[54] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011. Moo: A Batteryless Computational RFID and Sensing Platform. In *Technical Report UMCS-2011-020.*

A Artifact Appendix

A.1 Abstract

This artifact appendix describes how to use the toolchain and reproduce functionality results for TotalRecall. The artifact contains source code for TotalRecall, baseline systems, and several benchmarks and is publicly available as a GitHub repository and on the open-access repository Zenodo. This artifact does not produce performance data for the comparison between TotalRecall and baseline systems; it is intended to be used to demonstrate basic functionality on real hardware.

A.2 Artifact check-list (meta-information)

- **Algorithm:** SRAM-based one-time checkpointing system.
- **Program:** Texas Instruments and custom-developed benchmarks. These benchmarks are included.
- **Compilation:** msp430-gcc version >= 8.3.0. Not included.
- **Binary:** Pre-compiled binaries for the MSP430G2553 and MSP430FR6989 are included.
- **Run-time environment:** Tested on Ubuntu 18.04 with mspdebug version >= 0.25 installed. Root access is required to install dependencies.
- **Hardware:** Texas Instruments MSP-EXP430G2ET or MSP-EXP430FR6989 Launchpads.
- **How much disk space required (approximately)?:** < 5 MB.
- **How much time is needed to prepare workflow (approximately)?:** Twenty minutes.
- **How much time is needed to complete experiments (approximately)?:** Ten minutes.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License

A.3 Description

A.3.1 How delivered.
TotalRecall is publicly available through its git repository at https://github.com/FoRTE-Research/TotalRecall-artifact and on Zenodo (https://doi.org/10.5281/zenodo.3562888). The repository contains both TotalRecall and benchmark source code, as well as prebuilt binaries to facilitate testing.

A.3.2 Hardware dependencies.
TotalRecall was developed and tested on the MSP-EXP430G2ET and MSP-EXP430FR6989 Launchpads; either works to reproduce the results here.

A.3.3 Software dependencies.

- msp430-gcc is the Texas Instruments open source toolchain for MSP430 microcontrollers. It is necessary to generate the benchmark binaries.
- mspdebug is a free, open source debugger for MSP430 MCUs. It is used to flash benchmark binaries to hardware.

A.4 Installation

TotalRecall: Clone the repository from `https://github.com/FoRTE-Research/TotalRecall-artifact` and follow the instructions provided in README.md.

msp430-gcc: We do not provide msp430-gcc as part of the artifact evaluation repository; install it by following the instructions at http://www.ti.com/tool/MSP430-GCC-OPENSOURCE.

mspdebug: A detailed description of the mspdebug installation process is available in README.md of the artifact repository.

A.5 Experiment workflow

1. Compile the benchmarks for your board by running `make DEVICE={msp430g2553, msp430fr6989} SYS={sram, flash, fram}` in the msp430 directory.
2. Flash a benchmark to the device by opening mspdebug and programming the corresponding binary, for example by entering `prog msp430/msp430g2553/bin/quicksort.bin` in the mspdebug console.
3. Close mspdebug to allow the MCU to execute freely.
4. On the MSP430G2553, the RGB LED is blue during benchmark execution and green when the benchmark is complete with correct output. On the MSP430FR6989, the red and green LEDs are both on during benchmark execution. The red LED turns off when the benchmark is complete. The quicksort benchmark takes approximately 10 seconds to complete.
5. Press button S1 to take a checkpoint and halt execution. On the MSP430G2553, the LED connected to port P1.0 will flash green; it is illuminated during the checkpoint routine.
6. Reset the device either by pressing the reset button or unplugging the Launchpad. Execution will continue from where it left off—if the benchmark was complete when the checkpoint occurred, the RGB LED should immediately turn green. Otherwise, the device should finish the benchmark starting from the point when the checkpoint was taken (e.g., if the checkpoint is taken 7 seconds into the quicksort benchmark it should reach completion in 3 seconds following a reset).
7. Verify that TotalRecall detects when SRAM state is lost by taking a checkpoint, unplugging the Launchpad, and then either waiting an extended time period (>= 5 minutes) or holding down the reset button, both of which drain the processor supply voltage. Restoring power to the device will start the benchmark execution from the beginning.

A.6 Evaluation and expected result

On either platform, resetting the device after taking a checkpoint results in execution restarting from the checkpoint rather than from the beginning. This is best measured by recording the runtime of a benchmark and then checkpointing/resetting partway through and determining the total runtime, or by waiting until the benchmark completes and then taking a checkpoint and resetting (the device should immediately enter the "benchmark complete" state).

On the MSP-EXP430G2ET, the device can also be reset by momentarily unplugging the Launchpad from its power supply. The 11.1μF of onboard capacitance is sufficient to retain SRAM across the time it takes a user to unplug and plug back in the launchpad. This procedure is not practical on the MSP-EXP430FR6989 due to the more power-intensive MCU, feature-rich launchpad, and lower capacitance (10.4μF); resetting the device without unplugging it demonstrates the software checkpoint/recovery, but not SRAM remanence.

The results in Figure 4 are collected using the custom power-control board shown in Figure 3. The control board separates the device under test (DUT) and the debug/power interface, which are normally connected/disconnected using jumpers on the MSP430 Launchpads. To prevent current draw through the debug interface, the DUT is isolated by MOSFET switches toggled by a secondary microcontroller on the control board. The DUT side of the $3.3V$ rail is powered by a DAC on the control board, and the $5V$ rail is left disconnected. When removing power, the DAC is switched to a high-impedance mode to better approximate real energy harvesting hardware.

Note: When using the NVM (Flash or FRAM) checkpoints, users should take care to manually remove the checkpoint when re-programming a workload so the system does not inadvertently "recover" into the old checkpoint. On the MSP430G2553, the pre-program Flash erase takes care of this. On the MSP430FR6989, running `fill 0x4400 2 0` in mspdebug will remove the flag value indicating a valid checkpoint.

A.7 Customization/Re-use

TotalRecall can be added to any system by recompiling and linking against the checkpointing libraries and new linker scripts. Because the demonstration version depends on buttons and LEDs attached to GPIO pins, porting to new devices also requires minor pin assignment changes in both the TotalRecall and benchmark code. More detailed instructions on how to re-use TotalRecall is in the README.md of the artifact.

We also provide different implementations of TotalRecall as well as baseline systems for comparison. On the MSP430G2553, we provide two checkpoint systems:

- SRAM in-place checkpoint verified by software-generated CRC 16 (default).
- Flash checkpoint.

On the MSP430FR6989, we provide two checkpoint systems:

- SRAM in-place checkpoint verified by hardware-generated CRC 32 (default).
- FRAM checkpoint.

Instructions on how to change the checkpoint system are included in the artifact's README.md.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20200102.html
- http://cTuning.org/ae/reviewing-20200102.html
- https://www.acm.org/publications/policies/artifact-review-badging