IntOS: Persistent Embedded Operating System and Language Support for Multi-threaded Intermittent Computing

Yilun Wu, *Stony Brook University;* Byounguk Min, *Purdue University;* Mohannad Ismail and Wenjie Xiong, *Virginia Tech;* Changhee Jung, *Purdue University;* Dongyoon Lee, *Stony Brook University*

https://www.usenix.org/conference/osdi24/presentation/wu-yilun

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

INTOS: Persistent Embedded Operating System and Language Support for Multi-threaded Intermittent Computing

Yilun Wu[*], Byounguk Min[†], Mohannad Ismail[‡], Wenjie Xiong[‡], Changhee Jung[†], Dongyoon Lee[*]

[*]Stony Brook University, [†]Purdue University, [‡]Virginia Tech

Abstract

This paper introduces INTOS, an embedded operating system and language support for multi-threaded intermittent computing on a battery-less energy-harvesting platform. INTOS simplifies programming with a traditional "thread" and a "transaction" with automatic undo-logging of persistent objects in non-volatile memory. While INTOS allows the use of volatile memory for performance and energy efficiency, conventional transactions do not ensure crash consistency of volatile register and memory states. To address this challenge, INTOS proposes a novel *replay-and-bypass* approach, eliminating the need for users to checkpoint volatile states. Upon power restoration, INTOS recovers non-volatile states by undoing the updates of power-interrupted transactions. To reconstruct volatile states, INTOS restarts each thread bypassing committed transactions and system calls by returning recorded results without re-execution. INTOS seeks to build a persistent, full-fledged embedded OS, supporting priority-based preemptive multithreading while ensuring crash consistency even if power failure occurs during a system call or while some threads are blocked. Experiments on a commodity platform MSP430FR5994 show that when subjected to an extreme power failure frequency of 1 ms, INTOS demonstrated 1.24x lower latency and 1.29x less energy consumption than prior work leveraging idempotent processing. This trend turns out to be more pronounced on Apollo 4 Blue Plus.

1 Introduction

Instead of using a battery, energy-harvesting systems [24, 30, 40, 54, 57] capture necessary energy from ambient sources (*e.g.,* solar [27], radio frequency [35]) and leverage a small capacitor as energy storage. The ability to offer sustainable and long-term deployment without the need for battery replacements has unlocked a diverse range of emerging applications such as body implants [29], wearables [61], wildlife tracking [67], road monitoring [26], and satellites [5].

Since the capacitor undergoes cycles of depletion and recharge, program execution on an energy-harvesting system is inherently *intermittent*, involving repetitive power interruptions and resumptions. The nature of intermittent computing necessitates crash consistency to guarantee correct resumption throughout frequent power cycles.

An operating system (OS) offers essential services to application developers (users), including multi-threading, queues, semaphores, events, and timers, to assist in creating feature-rich applications. To illustrate, widely-used embedded OSes like FreeRTOS [3] have streamlined the development of diverse embedded applications. Unfortunately, this level of OS-/runtime support is absent in intermittent computing environments. For instance, ImmortalThreads [65] offers a tiny runtime supporting (pseudo) threads with cooperative scheduling; however, its capabilities are limited. It lacks a wait-list for blocking threads. Its event loop is based on polling, wasting microcontroller (MCU) cycles. Many task-based solutions such as Ink [64] and CatNap [52] do not support threads.

There arises a growing need for a more robust OS tailored specifically for intermittent computing. Advancements in hardware technologies, such as ultra-low power microcontrollers like TI's MSP430FR [6] and ARM's Cortex-M4 [2], as well as non-volatile memory (NVM) like FRAM [12] and MRAM [10], have empowered intermittent applications to perform more computations. Emerging intermittent applications are becoming increasingly complex, incorporating features like multi-threading, communication, synchronization, and responsiveness to events. We started witnessing machine and deep learning tasks [16, 28, 39, 48] on energy-harvesting platforms. Despite these advancements, users are compelled to manage this complexity without adequate OS support.

Unfortunately, the current crash consistency solutions are hard to adopt or result in inefficient designs when applied to the development of persistent embedded OS kernels. Some approaches [22, 31, 49, 52, 53, 56, 64] require users to decompose applications into a task graph, demanding each task to inherently possess failure-atomicity and idempotence. This poses considerable challenges for programmers [38, 65]. Breaking down a kernel system call, such as creating a thread or blocking on a full queue, into tasks is not trivial. Other compiler-based solutions [15, 18, 19, 37, 47, 50, 55, 62] automatically divide programs (*e.g.,* into idempotent regions) and incorporate checkpoints, requiring little to no user annotations. Thus, they may be used to build a persistent OS. Yet, many (except Chinchilla [50]) assume execution solely on NVM, overlooking potential advantages offered by volatile memory.

This paper introduces INTOS, a new persistent full-fledged embedded OS accompanied by language support for multi-threaded intermittent computing (§4). To ease intermittent application programming, INTOS offers a traditional "thread" along with a priority-based preemptive scheduler. INTOS also allows users to define a standard "transaction" with automatic undo-logging to ensure the crash consistency of persistent objects residing in NVM, akin to a widely adopted Intel's Persistent Memory Programming Kit (PMDK) [7]. INTOS places program stacks, encompassing local variables and function frames, in volatile memory to improve performance and energy efficiency. However, their crash consistency in the event of a power failure is not safeguarded by transactions. The absence of volatile states (*e.g.,* stacks) makes it impossible to simply resume from the beginning of a transaction.

To address this challenge, INTOS proposes a new *replay-and-bypass* approach (§5). Upon power restoration, INTOS recovers non-volatile states by undoing the updates of power-interrupted transactions. To reconstruct volatile states, it then restarts each thread from the beginning while bypassing committed transactions and system calls by returning recorded results without re-execution. This approach is grounded in the insight that reconstructing the volatile states with replay-and-bypass is more energy-efficient, compared to alternatives checkpointing volatile states to NVM—since NVM writes are the most energy-consuming in the instruction set architecture.

In particular, INTOS provides a programming model based on Rust, leveraging Rust's type system to enforce various programming rules (§6). These rules are designed to ensure crash consistency: *e.g.,* the prohibition of modifications to persistent objects outside of transactions. The adaptability of this programming model has been showcased through the successful implementation of the INTOS kernel, featuring multithreading, queues, semaphores, events, and timers.

We evaluate INTOS with three single-threaded and eight multi-threaded applications, including those ported from RIoTBench [58], an IoT/Edge stream processing benchmark for real city sensing and fitness sensing data, on MSP430FR5994 [6] and Apollo 4 Blue Plus [1]. We compare INTOS with Ratchet [62] where compiler-based idempotent processing is applied in both the INTOS kernel and application. On the MSP430FR platform without power failures, INTOS exhibited 1.65x lower latency and 1.85x less energy, compared to Ratchet. Even when subjected to an extreme power failure frequency of 1 ms, INTOS demonstrated 1.24x lower latency and 1.29x less energy overhead. This trend became more pronounced on the Apollo 4 platform.

This paper makes the following contributions:

- To the best of our knowledge, INTOS is the first persistent embedded OS that supports priority-based preemptive multithreading and other core features, tailored for intermittent computing with frequent power failures.
- INTOS combines transactional programming with a new replay-and-bypass recovery mechanism to ensure whole-system crash consistency, encompassing both volatile and non-volatile memory states at both user and kernel levels.
- INTOS introduces a Rust-based programming model ensuring crash consistency through the proposed transactions and replay-and-bypass recovery mechanisms.
- INTOS is to our best knowledge the first intermittent system that is evaluated with multithreaded applications.

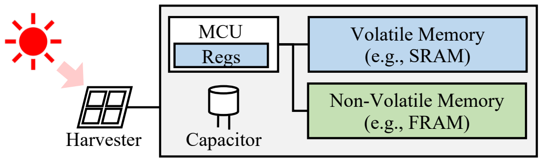


Figure 1: An architecture of energy harvesting platforms (*e.g.,* MSP430FR). Registers and SRAM (blue boxes) are volatile.

2 Background

This section briefly discusses intermittent computing, embedded OS, and transactions.

2.1 Intermittent Computing

Execution on an energy-harvesting platform is *intermittent*, *i.e.,* it abruptly halts upon the depletion of the capacitor and resumes after recharging, typically to the full capacitance. This implies that the program is often power-interrupted, and therefore intermittent computing requires to ensure crash consistency for correct recovery across frequent power cycles.

Figure 1 depicts an architecture of energy harvesting platforms available in TI MSP430 [6] or Ambiq Apollo 4 [1]. The energy harvester gathers ambient energy (*e.g.,* solar, RF) and stores it in a capacitor. Capacitor sizes typically vary from a few to several hundred microfarads (µF). For reference, WISP [57] uses 47 µF. The computing components include an ultra-low power microcontroller (MCU) along with both volatile memory (*e.g.,* SRAM) and non-volatile memory (*e.g.,* FRAM [12] or MRAM [10]). For instance, the MSP430FR5994 features a 16 MHz MCU with 8KB SRAM and 256KB FRAM. Registers and SRAM states (blue boxes) are lost upon a power outage. Previous solutions (§3) have suggested diverse approaches to maintaining the crash consistency of data stored in registers, volatile memory, and non-volatile memory across a power cycle.

Prior Works	Crash Consistency	Multithreads?	Queues?	Semaphores?	Events?	Timers?	Prog. Burden?	Volatile Mem?
Alpaca [49], Coala [53], MayFly [31]	manual task decomposition	no (tasks)	no	no	no	no	high	yes
Chain [22]	manual task decomposition	no (tasks)	limitedly	no	no	no	high	yes
Coati [56], Ink [64]	manual task decomposition	no (tasks)	no	no	limitedly	no	high	yes
CatNap [52]	manual task decomposition	no (tasks)	limitedly	no	limitedly	no	high	yes
Ratchet [62], WARio [37]	idempotent processing	no	no	no	no	no	none, very low	no
Chinchilla [50]	ckpt & undo-logging	no	no	no	no	no	none, very low	yes
HarvOS [15], RockClimb [19]	static energy analysis	no	no	no	no	no	none, very low	no
TICS [38]	ckpt & undo-logging	no	no	no	no	no	low	no
ImmortalThread [65]	ckpt & micro-continuation	yes (pseudo-stackful)	no	limitedly	limitedly	no	low	no
INTOS (ours)	replay & undo-logging	yes (stackful)	yes	yes	yes	yes	medium (transactions)	yes (replay)

Table 1: A comparison of the main features of INTOS with prior intermittent computing solutions.

2.2 Embedded Operating Systems

Embedded OSs [3,13,14,25,41,42] are a specialized software layer that provides essential services for the target embedded system. They empower users (application developers) to create applications with rich features using a conventional thread-based programming model, even within resource-constrained environments. For instance, FreeRTOS [3], widely recognized as one of the most adopted, supports (1) multi-threading with a priority-based preemptive scheduler; (2) synchronization (*e.g.,* semaphores) and communication (*e.g.,* queues) among threads; (3) dynamic memory allocation; and (4) software timers. An embedded OS is intimately linked with the application code and is typically included as part of the firmware image. Existing embedded OS kernels are not designed to be crash-consistent and do not support intermittent computing.

2.3 Transactions for Non-volatile Memory

Transactions stand out as a widely adopted programming model for NVM, as demonstrated by Intel's PMDK [7] for Optane memory [4]. Users can allocate a persistent object using a non-volatile memory allocator. A transaction employs undo logging (or redo logging) to ensure failure-atomicity (the "all-or-nothing" semantic) for operations executed during the transaction. Transactional programming has demonstrated success in the development of complex software such as persistent memchached [9] and redis [8].

3 Related Work

This section initially emphasizes the absence of essential OS features in prior solutions (Table 1) and then delves into the challenges associated with applying existing crash consistency solutions to design persistent OS services.

No OS exists for intermittent computing. As highlighted in the middle five columns of Table 1, current intermittent processing runtimes lack essential features present in modern embedded OSes. ImmortalThreads [65], for example, introduces (pseudo) multithreading with "non-blocking" spin-locks and event buffers. Spinning results in inefficient utilization of MCU cycles. To support "blocking" semaphores, queues, event groups, and software timers in intermittent computing, an OS/runtime should maintain a run-queue, wait-queues, and other relevant kernel metadata in a crash-consistent manner. ImmortalThreads (its runtime) does not offer them.

We believe ImmortalThreads can be extended to implement such missing kernel features using its micro-continuation approach. However, we expect ImmortalThreads would suffer from two fundamental problems. First, ImmortalThreads would incur high performance overhead. Unlike those hardware-based roll-forward solutions [17,21,36,51,66] that detect impending power failure and save registers to resume from the failure point, ImmortalThreads does not (cannot) sense the dying voltage. Thus, it ends up persisting a program counter in every store instruction to enable roll-forward recovery (micro-continuation). Second, micro-continuation only works for non-volatile memory and excludes volatile SRAM available in commodity energy harvesting systems, thereby losing a great opportunity to enable more energy-efficient intermittent computing. JustDo logging [34], from which the micro-continuation idea is inspired, also requires the entire memory hierarchy to be fully persistent. We discuss ImmortalThreads' potential high overhead later in §10.

On the other hand, Ink [64], Coati [56], and Catnap [52] offers partial support for task-based event-driven runtimes, yet they do not accommodate threads and demand a task-based programming model, which we explain next.

Manual task decomposition adds programming burden. For crash consistency, several prior solutions [22,31,49,52,53,56,64] require users to decompose an application into a graph of "tasks". Each task is compelled to inherently guarantee failure atomicity and idempotence in the face of a power failure, leading to considerable programming challenges and design complexities. Some runtime systems employ a cooperative scheduler to execute multiple tasks. However, the manual task decomposition shifts the responsibility of ensuring crash consistency onto users. This has been demonstrated to be a significant burden for programmers [38, 65]. For example, breaking down a kernel system call, such as creating a thread or blocking on a full queue, into tasks is far from trivial.

Within the task-based model, several new features have been introduced. For example, Alpaca [49] suggests task privatization, creating a volatile copy of shared non-volatile variables before entering each task. A task's local computation can run on volatile memory. Upon task completion, updates to shared variables are committed to NVM in a double-buffered manner. Chain [22] abstracts inter-task variable passing with the use of a persistent queue; Ink [64] and Coati [56] support event-driven programming; and CatNap [52] adaptively schedules tasks based on task priority, energy consumption, current energy level, and charging rate.

Automatic checkpointing often does not consider volatile memory. Several works [15, 19, 37, 47, 50, 55, 62] have introduced compiler support to automatically partition a program into multiple regions and insert checkpointing at the boundaries of these regions. Users require little to no annotations, so they can be used to build persistent OS services. Our evaluation (§10) includes a comparison against Ratchet [62] idempotent processing. However, many compiler-based solutions assume a program execution solely on non-volatile memory, foregoing the potential performance and energy efficiency benefits that volatile memory could provide. Experiments with MSP430FR5994 (§10.1) show that executing our 11 benchmarks entirely in non-volatile memory (FRAM) results in 1.11x latency and 1.16x energy overheads compared to running them entirely in volatile memory (SRAM). In this context, only live-in (volatile) registers necessitate checkpointing at a region boundary. A notable exception is Chinchilla [50] which maintains volatile and non-volatile stacks, yet it still involves frequent checkpoints of stacks to NVM.

Ratchet [62] divides a program into idempotent regions [23, 43, 45, 46] with no write-after-read dependencies within a region. Chinchilla [50] selectively skips certain checkpoints based on energy conditions. WARio [37] reduces the number of idempotent regions by reordering instructions and incorporating a loop optimization. Differently, HarvOS [15] partitions a program into regions where the energy required to complete that region is less than the energy buffer size. RockClimb [19] checks the energy level at the region boundary and only proceeds if there is sufficient energy. On the other hand, TICS [38] and ImmortalThreads [65] leverage a compiler for checkpoint instrumentation without region partitioning. TICS employs stack segmentation, where only a working stack (and registers) is checkpointed in NVM via a two-phase commit. ImmortalThreads introduces micro continuation, which checkpoints every memory update, ensuring the idempotence of the execution until the next checkpoint (store).

Other issues Prior works also address data timeliness [31, 38, 52, 64], event-driven programming model [38, 52, 56, 64], and others. Surbatovich et al. [59, 60] use Rust's type systems for data freshness checking and crash consistency. Hardwawre support [20, 32, 36, 44, 63, 68] also exists.

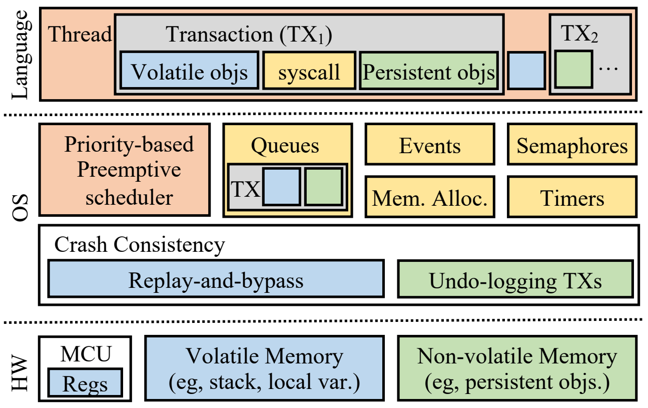


Figure 2: Overview of INTOS

4 Overview of INTOS

Figure 2 shows an overview of INTOS, embedded OS and language support for multi-threaded intermittent computing.

4.1 Multithreading and Transactions

Threads To ease the aforementioned programming burden and keep the same embedded application programming model, INTOS supports a traditional stackful "thread" as a programming unit and a schedulable entity, similar to commodity embedded OSes (*e.g.,* FreeRTOS[1]). Users can generate multiple threads through the system call `sys_create_thread` (Table 2). These threads run concurrently. Users can assign different priorities for threads. The INTOS scheduler employs a priority-based preemptive scheduling policy, a widely adopted approach for real-time capabilities. More discussion on INTOS's real-time capability will follow in §8.

Transactions To facilitate the utilization of both volatile and non-volatile memories while simplifying crash-consistent programming, INTOS offers a conventional "transaction" with automatic undo-logging to ensure crash-atomicity of persistent objects. Program stacks, encompassing local variables and function frames, reside in volatile memory. Users can either annotate a persistent variable (*e.g.,* globals) or employ the `sys_palloc` system call to create a persistent object in non-volatile memory. Both volatile and persistent objects can be used inside a transaction. Users do not need manual undo-logging. INTOS's transactions ensure that updates on persistent objects (not volatile ones) within a transaction are crash-atomic via automatic undo-logging. INTOS leverages Rust to identify the first writable dereference. As persistent objects share a base class or trait in Rust, the logging logic is integrated into the dereference operation within the class/trait. This approach mirrors PMDK's undo-logging support for its

[1]FreeRTOS employs the term "task", but it is technically a preemptive thread. For clarity and to distinguish it from the (cooperative) task in manual task decomposition works (§3), we will refer to it as a (preemptive) thread.

C++ programs. In §7.3, we discuss an undo logging optimization that logs old values only if there is a write-after-read dependency in a transaction.

Language The Rust type system in INTOS guarantees that persistent objects are not modified outside transactions (§6). Conversely, volatile states, such as local variables in program stacks, can be utilized both outside and inside transactions.

Challenges INTOS allows users to employ volatile variables for computing, yet INTOS transactions do not protect them. Consequently, the states of program stacks are susceptible to loss upon a power failure. A program cannot resume from the beginning of the failed transaction due to the absence of stack (and register) states. One solution could involve abstaining from the use of volatile memory, a proposition we oppose for energy efficiency reasons. Another approach might be to checkpoint volatile states to NVM at the onset of each transaction, but this would be expensive.

4.2 Replay-and-Bypass

To address the above challenge, INTOS proposes a novel *replay-and-bypass* approach (§5) to guarantee whole-system crash consistency across a power cycle. INTOS eliminates the need for users to checkpoint or create customized crash consistency solutions for volatile register and memory states. Upon power restoration, INTOS first recovers non-volatile states by undoing uncommitted transactions. Then, the thread is restarted from the beginning, safely resuming with empty registers and stack states. Throughout the execution, committed transactions and system calls are replayed and bypassed by returning the recorded results without re-execution – resulting in a more energy-efficient recovery process. Volatile states are reconstructed, enabling the program to resume beyond the point of power failure.

4.3 Persistent Embedded OS

System Calls INTOS provides comprehensive multithreading features (Table 2), comparable to those found in FreeRTOS. For instance, threads can communicate and/or synchronize with each other using the `sys_queue_*` and `sys_semaphore_*` system calls. A thread might block, for instance, if a queue is either empty or full. Multiple threads may access a shared persistent object by obtaining its reference (inside a transaction). Later in §6, we delve into how INTOS's programming model ensures the obligatory use of a mutex for synchronization via Rust's strong type system.

Kernel Crash Consistency Similar to user threads, INTOS kernel codes, including system calls, utilize volatile and non-volatile memories. The INTOS kernel employs the same undo-logging transactions to ensure crash consistency of persistent kernel objects that undergo updates during system calls. Table 2 lists the number of transactions and examples

Features	System calls	TXs	Persistent kernel objects
Threads	sys_create_thread	2	ready_list, thread_cnt, heap
	sys_thread_delay	1	delay_list
Queues	sys_queue_create	1	heap
	sys_queue_send_back	2	queue and its waitlist
	sys_queue_receive	2	queue and its waitlist
Events	sys_event_group_create	1	heap
	sys_event_group_wait	3	event_grp and its waitlist
	sys_event_group_set	3	event_grp and its waitlist
Semaphores	sys_create_semaphore	1	heap
	sys_semaphore_take	2	semaphore and its waitlist
	sys_semaphore_give	2	semaphore and its waitlist
Dyn. memory	sys_palloc	1	heap
	sys_pfree	1	heap
Timers	sys_timer_create	1	heap
	sys_start_timer	2	timer_cmd_q and its waitlit
	sys_reset_timer	2	timer_cmd_q and its waitlist

Table 2: INTOS supports full-fledged embedded OS features, akin to FreeRTOS [3]. Some system calls are not listed.

of persistent kernel data safeguarded by kernel-level transactions. Later in §7.2, we also discuss that the kernel uses optimized transactions (without undo-logging) for frequently used linked lists operations (*e.g.,* `ready-list`, `wait-list`). Using the same replay-and-bypass, INTOS provides a crash consistency guarantee even if a power failure occurs in the midst of a system call and some threads are blocked.

Energy efficient execution Designing an OS and language support for intermittent computing requires more than merely ensuring crash consistency. Both (fail-free) execution and recovery should be energy-efficient. INTOS provides energy-efficient execution by: (1) Utilizing both volatile and non-volatile memories; (2) Avoiding the checkpointing of volatile states; (3) Optimizing undo-logging for non-volatile states (§7.3); (4) Offering blocking/waiting system calls, such as semaphores and events, in contrast to existing approaches like ImmortalThreads [65], which requires spinning and wastes MCU cycles; (5) In the absence of events, with a blocking mechanism, allowing the MCU to enter a deep sleep mode where only a subset of interrupts are monitored.

Energy efficient recovery INTOS offers energy-efficient recovery by: (1) Utilizing replay-and-bypass recovery to avoid redundant execution (§5); (2) Undoing only the non-volatile state relevant to the high-priority thread that will resume during recovery (§5.2); (3) Introducing loop optimization (§7.1).

4.4 INTOS Program Example

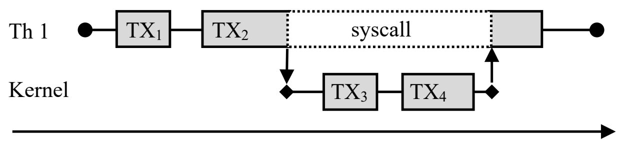
The example presented in Listing 1 illustrates the `recognize` program with two transactions. In this example, a queue is created to enable message passing between two threads, like a Linux pipe. A thread (`recognize`) is reading data from the sensor and sending the data to another thread (not shown) for processing using the queue. `PBox` is a smart pointer for a

```
1  fn recognize(model: PBox<Model>) {
2    let (q,stats) = transaction::run(|j, t| {
3      // syscall to create a queue
4      let q=sys_queue_create::<Result>(Q_SZ, t).unwrap();
5      // syscall to create a persistent object
6      let stats = PBox::new(Stats::new(), t);
7      ...
8      return (q,stats)
9    });
10   transaction::run(|j, t| {
11     // obtain read only ref, no logging
12     let mdl_ref = model.as_ref(j);
13     // syscall to perform I/O
14     let reading = sys_read(SENSOR_0);
15     // data processing in volatile buffer
16     let mut window = [AccelReading::new(); 3];
17     init_window(&readings);
18     transform(&mut window, j);
19     let feature = featurize(&window);
20     let class = classify(&feature, mdl_ref);
21     // obtain mutable ref, auto. undo logging
22     let mut stats_ref = stats.as_mut(j);
23     stats_ref.cnt[class] += 1;
24     // syscall to send result
25     sys_queue_send_back(q, class, WAIT_TIME, t);
26   });
27   ...
28 }
```

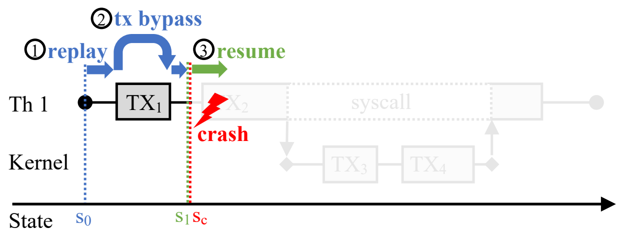
Listing 1: An example INTOS program with transactions.

persistent object. Users can enclose a program region with the transaction construct, `transaction::run(|j,t|{ ... })`, where `j` represents the journal object and `t` is the system call token. The journal object enforces restrictions, ensuring that persistent smart pointers like `PBox` cannot be dereferenced outside a transaction, while the system call token restricts system calls to occur exclusively within a transaction. Further details on this will be provided in §6.

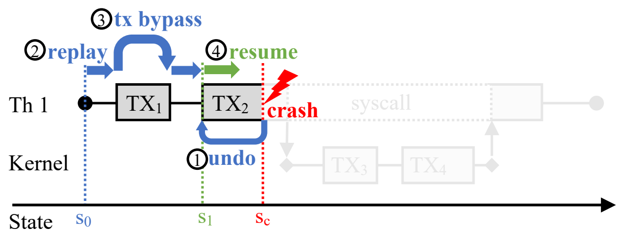
The first transaction (Lines 2-9) involves creating a queue with a size of `Q_SZ`. This queue contains objects of type `Result`, and a persistent object (`stats`) that holds counts (`cnt`) for each class/result. The transaction returns them after some processing. The second transaction (Lines 10-26) reads an ML model using a read-only reference. This eliminates the need for undo-logging. Following I/O, it conducts data processing (Lines 16-20) such as filtering, normalization, and classification, notably on a volatile buffer. This strategy enhances performance and energy efficiency compared to conducting all intermediate computations on a non-volatile buffer. Subsequently, the transaction obtains a mutable reference to a persistent object (`stat`), created, and passed from the first transaction, and updates it. As this is the first write after getting a mutable reference, INTOS automatically applies undo logging. Finally, the transaction makes a system call `sys_queue_send_back` to place the result into the queue, maintained by INTOS. Another thread (not shown) can then receive the result from the queue for subsequent processing.



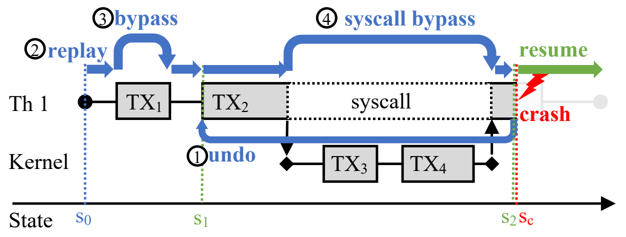
(a) Case 1: An execution of Listing 1 without a power failure.



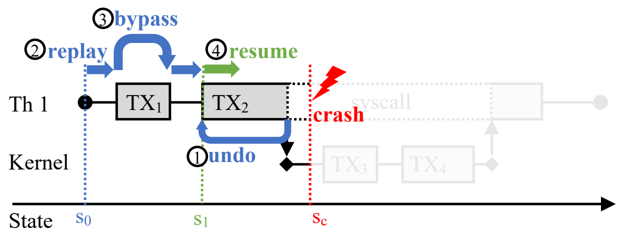
(b) Case 2: A power failure outside a transaction



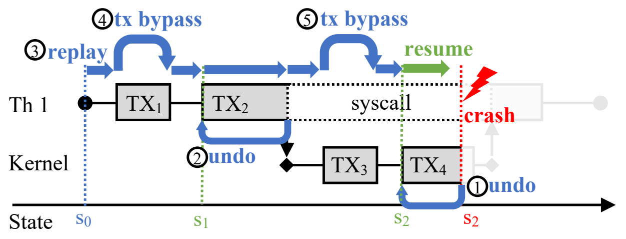
(c) Case 3: A power failure inside a transaction before a syscall



(d) Case 4: A power failure inside a transaction after a syscall



(e) Case 5: A power failure during a syscall before a transaction



(f) Case 6: A power failure during a syscall inside a transaction

Figure 3: Replay-with-bypass recovery examples

5 Replay-and-Bypass Recovery

The following two sections demonstrate INTOS's replay-and-bypass approach. along with examples.

5.1 Single Thread Crash Consistency

Let's illustrate INTOS's *replay-and-bypass* recovery mechanism using the `recognize` example in Listing 1, which involves two user-level transactions, $\texttt{TX}_1$ and $\texttt{TX}_2$. Figure 3a depicts an execution of `recognize` without a power failure. For simplicity, the system calls in the first transaction $\texttt{TX}_1$ are omitted, and only the system call `sys_queue_send_back` (Line 23) made by $\texttt{TX}_2$ is highlighted. Assume that the system call includes two kernel-level transactions, $\texttt{TX}_3$ and $\texttt{TX}_4$.

In Figure 3b, we consider a scenario where $\texttt{TX}_1$ has been committed, and then a power failure occurs before $\texttt{TX}_2$ starts (outside transactions). Upon power recovery, INTOS initiates a *replay* of the thread from the beginning (step ①), restarting with empty registers and stack state $\texttt{s}_0$. INTOS's type system (§6) ensures that no non-volatile states are updated outside the transaction. Volatile states are *reconstructed* during replay. Since the non-volatile states at $\texttt{s}_c$ (before the power failure) already incorporate the effects of the committed transaction $\texttt{TX}_1$, re-executing $\texttt{TX}_1$ would be incorrect and non-idempotent. Therefore, INTOS *bypasses* the transaction $\texttt{TX}_1$ (step ②), simply returning the logged return value without re-execution. No system calls are made during bypass, and no kernel-level recovery is required. INTOS ensures that the program reaches the same state $\texttt{s}_1$ as $\texttt{s}_c$, from which it can safely resume.

Now, let's consider a power failure inside a transaction. In Figure 3c, a power failure occurs inside a user-level transaction before a syscall. INTOS's undo-logging transaction ensures the failure-atomicity of non-volatile states changed within the transaction. Upon power recovery, INTOS applies undo-logging (step ①) to roll back the (user-level) non-volatile states from $\texttt{s}_c$ to $\texttt{s}_1$, the state before the transaction begins. Next, INTOS starts a replay from the beginning state $\texttt{s}_0$ (step ②). The committed transaction $\texttt{TX}_1$ is bypassed (step ③), and INTOS reconstructs all volatile states along the way, making the state $\texttt{s}_1$ (after replay) equivalent to $\texttt{s}_c$ (before the failure).

Figure 3d illustrates the actions to be taken if a power failure occurs after a syscall completes (inside a user-level transaction). As usual, INTOS applies undo-logging (step ①) and initiates a replay (step ②). The committed transaction $\texttt{TX}_1$ is bypassed (step ③). Notably, in this scenario, while replaying transaction $\texttt{TX}_2$, INTOS also bypasses the completed system call (step ④). Consequently, INTOS avoids the need to alter kernel states — any changes to kernel-side non-volatile states made during the original system call (before a power outage) can remain unchanged. The INTOS kernel caches the return value of a system call upon its completion (before a power failure). Then it simply returns the cached value during replay. From the user thread's perspective, a system call can be considered as a nested black-box transaction.

Now, let's delve into scenarios where a power failure occurs during a system call. As mentioned earlier, INTOS utilizes transactions ($\texttt{TX}_3$ and $\texttt{TX}_4$) to safeguard kernel-side non-volatile data. If a crash occurs before (or outside) a kernel-side transaction, as depicted in Figure 3e, the situation is straightforward and aligns with the case presented in Figure 3c. There is no need to undo anything in the kernel. INTOS simply undoes the user-level transaction that invoked the system call (①) and initiates the replay-and-bypass recovery mechanism.

On the other hand, if a crash occurs inside a kernel-side transaction, as illustrated in Figure 3f, INTOS must first undo transaction $\texttt{TX}_4$ (step ①) to roll back the kernel-side state to $\texttt{s}_2$, followed by undoing transaction $\texttt{TX}_2$ (step ②) to roll back the user-side state to $\texttt{s}_1$. INTOS then employs a replay from initial $\texttt{s}_0$ (step ③), bypassing the committed transactions on the user side, $\texttt{TX}_1$ (step ④), and on the kernel side, $\texttt{TX}_3$ (step ⑤). Note that INTOS rolls back the kernel-side transaction first (before any aborted user-level transaction). This has correctness implications in multi-thread scenarios, which will be discussed in the subsequent section.

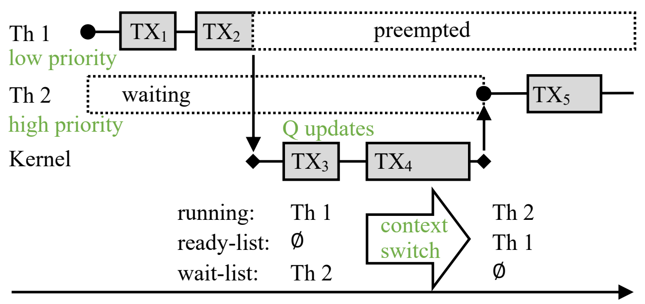
5.2 Multi-Threads Crash Consistency

Next, we discuss INTOS's approach to ensuring crash consistency for multiple threads. Specifically, INTOS employs priority-based recovery and resumption. Upon power restoration, INTOS always recovers and replays the thread with the highest priority among those ready.

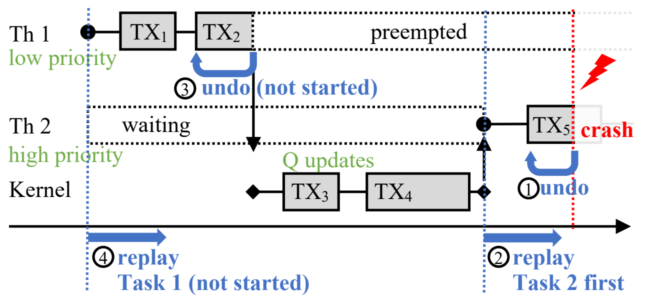
Figure 4a illustrates a two-thread execution without a power failure. Initially, high-priority Thread 2 waits, for example, on a queue. A low-priority Thread 1 uses `sys_queue_send_back` to enqueue data, allowing Thread 1 enabled (its waiting condition is satisfied). During the system call, the kernel-side transaction $\texttt{TX}_3$ updates the kernel queue object in NVM. As Thread 1 is awakened and has a higher priority, the INTOS scheduler preempts Thread 1 and context-switches to Thread 2 by modifying thread-related persistent linked lists, such as `ready-list` and `wait-list`, in transaction $\texttt{TX}_4$. It is a common pattern for a system call to update a system call-specific kernel data structure (*e.g.,* queue) in one transaction and to modify schedule-related linked lists in another transaction. After the context switch, Thread 2 runs, and Thread 1 remains on the `ready-list`, awaiting its turn.

Let's first consider a simple power failure case. If power is lost during the system call (*e.g.,* during $\texttt{TX}_3$ or $\texttt{TX}_4$ or between them), it constitutes a single-thread scenario. The recovery protocol remains the same as the case presented in Figure 3f.

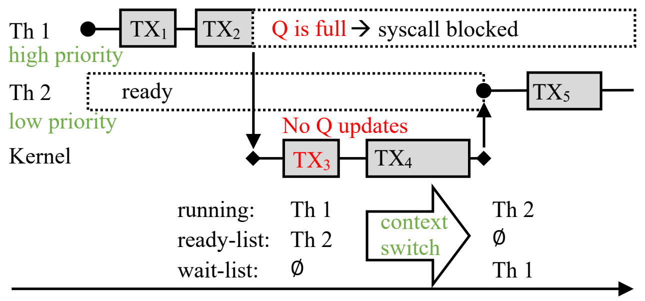
Suppose a power outage occurs while running Thread 2 (after the context switch) as depicted in Figure 4b. This makes a multi-thread scenario: Threads 1 and 2 are runnable. Upon power restoration, INTOS recovers and runs Thread 2 — the thread that was running and experienced a power failure. The priority-based scheduler always schedules the thread with



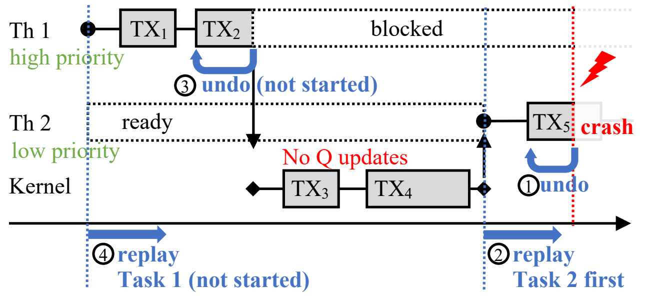
(a) A two-thread execution without a power failure. Initially, a high-priority Thread 2 is waiting. A low-priority Thread 1 makes it ready.



(b) A power failure occurs after Thread 2 is scheduled.



(c) A two-thread execution without a power failure. Initially, a high-priority Thread 1 runs first and makes a blocking system call, yielding a turn to a low-priority Thread 2.



(d) A power failure occurs after Thread 2 is scheduled.

Figure 4: Multi-threads recovery examples

the highest priority among ready threads. Thus, recovering the failed thread implies that when power becomes available, INTOS runs the ready thread with the highest priority. In this example, it is Thread 2. INTOS roll-backs transaction $\mathtt{TX}_5$ (step ①) and replays Thread 2 (step ②). On the other hand, INTOS does not eagerly undo transaction $\mathtt{TX}_2$ (step ③). Hard-won energy should not be wasted. A system may not possess enough energy to run Thread 1 (after Thread 2). High-priority ready Thread 2 takes precedence over undoing the transaction $\mathtt{TX}_2$ of Thread 1. Sometime later, when Thread 1 is scheduled, INTOS then rolls back transaction $\mathtt{TX}_2$ on demand (step ③) and replays Thread 1 (step ④).

When managing multiple threads, a blocking system call warrants detailed discussion. Figure 4c depicts another two-thread execution without a power failure, distinct from Figure 4a. In this scenario, Thread 1 possesses high priority, and even though Thread 2 is ready, it is not scheduled. Suppose the queue is already full. Assuming the queue is already full, when Thread 1 employs the `sys_queue_send_back` system call to enqueue data, it discovers the queue lacks space and becomes blocked. Subsequently, the scheduler moves Thread 1 to the `wait_list` in transaction $\mathtt{TX}_4$. In this scenario, it is crucial to note that $\mathtt{TX}_3$ is indeed a null transaction, making no updates to the queue. An essential invariant established by the INTOS kernel is that a blocking system call, if it actually blocks, does not alter the state of system call-specific persistent objects (*e.g.,* queue). The system call is not considered complete, and no result is cached for bypassing. The impact of a blocking call is confined to schedule-related linked lists in $\mathtt{TX}_4$. Given that a blocking system call has no substantive effect on kernel states, it is safe to proceed with the same recovery and replay of the ready thread with high priority — Thread 2 in this example. Any processing for the blocked threads, such as Thread 1, can be deferred, as illustrated in Figure 4d. When Thread 1 is later scheduled for recovery and replay, it will re-invoke the system call as if it had not been issued previously. In the INTOS implementation, those system calls that may block always first check for a blocking condition to uphold this invariant.

6 INTOS Programming Model

INTOS's programming model upholds five rules designed to guarantee crash consistency.

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged. Modifications on persistent objects outside transactions are untracked. Therefore, any update to persistent objects should be confined within transactions. INTOS also prohibits the reading of persistent objects outside transactions to prevent potential divergence in program control flow during replay. When restarting, non-volatile memory states are not rolled back to the thread's outset. For example, in the scenario illustrated in Figure 3c, replay begins with non-volatile memory states still reflecting the state $\mathtt{s}_1$ after $\mathtt{TX}_1$. Consequently, control flow outside transactions should not rely on persistent objects. To precisely identify a subset of persistent object reads that may influence control flows, one can perform static analysis and selectively prevent them. INTOS, for simplicity, conservatively enforces the restriction of no reads (and writes) outside transactions. This approach does not overly constrain programmability since it is natural to assume that persistent objects are primarily used within transactions. Furthermore, INTOS permits a transaction to acquire references to persistent objects that were created or modified by another transaction and subsequently update them arbitrarily within the executing transaction, as demonstrated in Listing 1 (Lines 6, 22-23).

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value. Rule 2 further enforces Rule 1. Allowing the return of references would potentially enable users to directly modify persistent objects without proper logging. Mutable references should be acquired and dereferenced exclusively within a transaction, as exemplified in Listing 1 (Lines 22-23).

Rule 3: Persistent objects should not contain references to volatile objects. Volatile objects are susceptible to data loss during power failures. Storing their references in persistent objects is thus unsafe.

Rule 4: System calls (excluding Locks) should only be made within transactions. There is, in theory, no fundamental restriction against using a system call outside a transaction for crash consistency. Yet, INTOS mandates adherence to this rule to constrain the length of system call replay and bound memory resources. After a transaction concludes, there is no necessity to replay any system call within that transaction. As a result, the upper limit for system calls to be replayed is determined by the number of system calls in the last uncommitted transaction. INTOS can safely free the system call replay metadata for committed transactions.

Rule 5: Locks should not be used inside transactions. A critical section, defined by locks, should be larger than a transaction. Suppose two concurrent transactions, $\texttt{TX}_1$ and $\texttt{TX}_2$, utilize a lock when accessing a shared object `X` within transactions. $\texttt{TX}_1$ acquires the lock, updates `X`, releases the lock, but remains uncommitted. The concurrent $\texttt{TX}_2$ acquires the lock, reads `X`, performs some computation, releases the lock, and eventually commits. If a power failure occurs at this point while $\texttt{TX}_1$ remains uncommitted, a data consistency issue arises. This occurs because our transaction lacks "isolation" among concurrent transactions. Rule 5 is enforced to avoid this problem. Ultra-low power intermittent computing systems hardly use multi-cores. Introducing a more intricate yet efficient solution, such as tracking data dependencies between transactions and aborting one if a conflict is detected, doesn't appear necessary in this context.

Enforcement INTOS employs Rust's robust type system to uphold the aforementioned rules, akin to [33] that statically prevents common persistent memory programming errors within the realm of server-side (non-energy-harvesting) persistent memory programming. Rules 1-3 resemble those in [33], with INTOS extending Rule 1 to disallow reading persistent objects outside transactions to avoid potential control flow divergence during replay. Rules 4-5 are distinctive to INTOS. The implementation utilizes Rust's traits.

7 Optimization

INTOS employs three performance optimizations.

7.1 Loop Optimization

Threads in embedded systems often involve loops, such as event loops handling sensor readings or loops with numerous iterations, as seen in matrix multiplication for neural network machine learning threads. Consider a thread with a loop where the loop body comprises T transactions, and a power crash occurs on the N-th iteration. While INTOS's replay-and-bypass approach can bypass $(N-1)*T$ transactions (in addition to any committed transaction in the last iteration), the overall replay window's length could potentially be excessively long, leading to substantial energy consumption during replay.

To address this common scenario, INTOS introduces the new `nv_for_loop!` macro, extending the loop construct in Rust to utilize a non-volatile variable as the iteration counter. With the non-volatile iteration counter, INTOS can infer completed iterations (committed transactions therein) during replay, enabling a safe and efficient fast-forward to the last iteration without executing the bypass logics.

INTOS's Rust language enforces the absence of loop-carried dependent volatile states to safely employ `nv_for_loop!` optimization as it skips iterations during recovery. Users are still able to employ volatile variables within a loop body, provided there is no loop-carried dependency.

7.2 Linked List Optimization

The INTOS kernel extensively utilizes doubly-linked lists to manage threads and scheduling states. Nearly every system call involves the manipulation of these linked lists. Notably, we have identified optimization opportunities, recognizing that linked list updates within the kernel occur within a critical section, eliminating the need to account for arbitrary interleaving. Additionally, there are no intermediate volatile hardware buffers (such as store buffer or cache) between registers and non-volatile memory. Consequently, any store instruction promptly persists as it retires from the pipeline. With these factors combined, we can scrutinize the crash non-volatile state, reason through intermediate linked list update steps, and precisely identify the power failure point.

INTOS presents *crash state analysis*-based roll-forward recovery optimization for linked list transactions. There is no undo-logging. Instead, INTOS records an operation log including the type (*e.g.,* insert) and the node (data) — only one per operation. During recovery, INTOS analyzes the crash

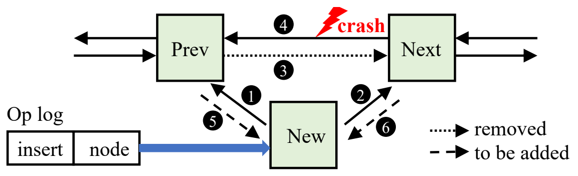


Figure 5: Crash state analysis to roll forward a list insertion

state remaining in NVM to infer the steps that have been completed. Then, it rolls forward the rest of the operation.

Consider an insertion transaction, illustrated in Figure 5. The process of list insertion involves six ordered steps. Initially, we link the node to its previous and next node (step ❶ - ❷). Then, the backward/forward link between the next node and the previous node is removed (step ❸ - ❹). Finally, we insert the new forward/backward link between the previous/next node and the new node (step ❺ - ❻). Suppose a power crash occurs before step ❹. During recovery, the operational log is first retrieved to determine the operation type and the node involved. It is discovered that the link from the new node to the previous/next node already exists, indicating that steps ❶ - ❷ are completed. It is also found that the forward link from the previous to the next node is removed, but the backward link remains intact. This observation suggests that the crash occurred before step ❹ was completed. INTOS can roll forward the operation by executing steps ❹ - ❻.

7.3 Undo-Logging Optimization

The default INTOS transactions automatically perform undo-logging on every first write (after acquiring a mutable reference), as in Listing 1 (Lines 22-23). INTOS introduces another smart pointer type, `Ptr<T>`, providing an option to leverage Rust's type system for the static detection of write-after-read (WAR) dependencies. A transaction utilizing `Ptr<T>` logs an old value only if there is a WAR dependency in the transaction, resulting in fewer logs. Within a transaction, users should dereference a persistent object pointer to obtain a reference. `Ptr<T>` does not provide users with a raw reference and imposes restrictions on the access interface, such as `r.read()` and `r.write()`. Consequently, utilizing `Ptr<T>` involves some additional coding efforts.

8 Discussion

Transactions for Partial vs. Whole System Persistence A crucial distinction between PMDK [7] (`libpmemobj`) and INTOS transactions lies in their persistence guarantees. `libpmemobj` supports "partial" system persistence, only ensuring the recoverability of non-volatile objects within transactions. Thus resuming program execution often requires user-defined custom crash-recovery logic to achieve consistent whole system states including volatile ones. In contrast, INTOS offers "whole" system persistence through the proposed replay-and-bypass mechanism, guaranteeing the recovery of both persistent and volatile states.

Transaction Length To ensure forward progress, INTOS mandates that a transaction must be able to complete with a fully charged capacitor. INTOS handles only one ready, highest-priority thread at a time and employs replay-and-bypass mechanisms to skip committed transactions and system calls, ensuring progress as long as one transaction successfully passes each power cycle. INTOS asks users to ensure this property via profiling. Bounding the size of a program region is a common requirement for many intermittent computing systems (*e.g.,* an idempotent region, a failure atomic section, and a transaction in INTOS) to ensure stagnation-free execution. Consequently, previous solutions including Choi et al. [19] have proposed various dynamic (profiling) and static program analysis techniques considering the worst-case behaviors. INTOS's kernel transactions are intentionally designed to be brief, considering this constraint. Our evaluation (§10) reports the maximum number of cycles per transaction in tested applications is short enough.

Energy-aware Scheduler If hardware provides a capability to monitor the remaining energy in the capacitor, one can design an energy-aware scheduler in INTOS: *e.g.,* not scheduling a thread if it is soon to stop.

Real-time Capabilities INTOS provides real-time capabilities comparable to FreeRTOS as long as the power is on. Yet, INTOS does not provide (hard) real-time guarantees due to the non-deterministic energy nature inherent in intermittent computing, rendering such assurances impossible.

Rust Rust is chosen for static correctness guarantees. Users can use C or other languages, provided they adhere to the programming rules (§6). It is feasible to statically link C programs with the Rust INTOS kernel since the contract/interface between the kernel and a user program is well-defined. Using C would require complex static program analysis to verify adherence to the programming rules. Additional static analysis should be employed for automatic undo-logging.

9 Implementation

We implement INTOS using the Rust programming language, leveraging its strong static type system to uphold INTOS's programming model (§6) with performance comparable to C. The initial implementation of the INTOS kernel mirrors FreeRTOS, having been ported to Rust and extended with transaction and crash consistency support. User threads are also crafted in Rust. Presently, INTOS extends support to two architectures: ARM Cortex-M4 and MSP430. The overall INTOS implementation, excluding testing and benchmark code, encompasses approximately 9900 lines of Rust code. We elaborate on some details below:

Multithreading The INTOS kernel allocates essential data structures, such as the thread control blocks, inter-thread communication objects (*e.g.,* queue, semaphore), and scheduling lists (*e.g.,* ready-list, wait-list) in non-volatile memory. Table 2 (last column) lists some examples.

Replay Tables To support *replay-and-bypass* recovery, INTOS maintains three per-thread replay tables that cache the return values of user-level transactions, kernel-level transactions, and system calls. For each table, the `tail` pointer indicates the last completed transaction or syscall, and the `current` pointer points to the presently executing one. The transaction `tail` pointer contains the commit flag, transaction id, and the pointer to the replay table.

Log Sizes The logged results of system calls generated within a transaction are garbage-collected upon the completion of each transaction, thereby bounding the maximum length of system call logs. Upon the completion of a task, all transaction logs associated with that task can be cleaned. We assume that a task entails a finite number of transactions, which is typically valid given that embedded application tasks often serve as short event handlers. An exception arises with tasks executing transactions within a loop, potentially leading to unbounded logs. This scenario is addressed by the `nv_for_loop!` optimization (§7.1). Transaction result logs from completed (old) loop iterations can be safely discarded, thus capping the transaction log size per loop iteration.

10 Evaluation

We evaluate the performance of INTOS on two platforms: MSP430FR5994 [6] and Apollo 4 Blue Plus [1]. MSP430FR5994 features 256KB of non-volatile FRAM and 8KB of volatile SRAM. We configured its MCU to operate at 16MHz. The Apollo 4 Blue Plus is equipped with an ARM Cortex-M4 processor. It has 384KB of TCM (faster SRAM), 2MB of SRAM, and 2MB of non-volatile MRAM.

The benchmark suite comprises 11 applications. The first group encompasses three single-thread applications (BC, AR, MLP), utilized in previous studies [22,38,49,50,53,56,64,65]. The second group comprises four multithreaded applications (KV, SEN, EM, MQ) designed to evaluate the performance of INTOS's OS features, including locks (semaphores), timers, events, and queues, respectively. The final macro-benchmark group comprises four multithreaded applications (ETL, PRED, STATS, TRAIN), adapted from RIoT-Bench [58]. Table 3 provides the application name, description, the number of threads, transactions, and system calls. The last three columns will be discussed later.

We compare the following four configurations:

- **SRAM (not crash consistent, baseline)**: A vanilla application and INTOS kernel without crash consistency support (*i.e.,* no undo-logging, no replay-and-bypass) operate on volatile SRAM. All the data is in SRAM, while the code is stored on FRAM due to space limitations. In the event of a power outage, both register and SRAM memory states are lost. This configuration serves as the baseline.
- **FRAM (not crash consistent)**: A vanilla application and INTOS kernel without crash consistency support (*i.e.,* no undo-logging, no replay-and-bypass) runs on non-volatile FRAM. All data resides in FRAM. Volatile registers remain susceptible to loss. This setup underscores the limitations of not utilizing SRAM and establishes the lower bound for existing compiler-based checkpointing solutions assuming no volatile memory (§3).
- **INTOS (crash consistent)**: This configuration represents our approach using both SRAM and FRAM. It uses INTOS's transaction undo-logging and replay-and-bypass recovery to ensure whole-system crash consistency.
- **Ratchet (crash consistent)**: Ratchet [62] represents a state-of-the-art compiler-based idempotent processing solution that uses non-volatile FRAM only. We used Ratchet compiler to transform a vanilla application and INTOS kernel to idempotent regions — with neither undo-logging nor replay-and-bypass.

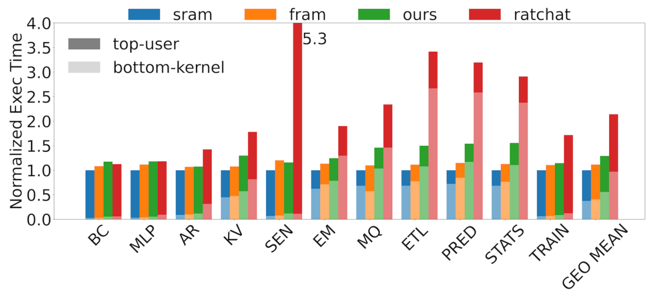


Figure 6: Latency overhead without power failure on MSP430

It is worth noting that we were unable to compare INTOS with ImmortalThreads [65] due to the incomplete nature of the publicly available code. It offers only the essential logic for micro-continuations and lacks OS/runtime features required by the tested benchmarks (*e.g.,* blocking queues). It was originally evaluated with four simple single-threaded Bitcount (BC), Cuckoo Filter (CF), Activity Recognition (AR), and DNN, which do not involve any application-OS interactions. Thus, conducting a fair comparison becomes impractical without ImmortalThreads' supplementary runtime support. Nonetheless, as discussed in §3, we expect its micro-continuation would suffer from high runtime overhead. For example, ImmortalThreads reports (See [65] Table 4 and Figure 7) that AR incurs 237% overhead with no failure and 300% with 5ms-period power failure. In contrast, we later show that in INTOS, AR experiences 8% and 15% overheads, respectively (See Figure 6 and Figure 8).

App	Description	Threads	TXs	Syscalls	Max Cycles/TX	LoC	Add&Mod
BC	Count the number of 1s in an integer using multiple algorithms	1	8	1	10676	181	32
MLP	Multi-layer perception with two fully connected layers	1	4	2	3488	155	30
AR	Train an activity recognition model and analyze the activities	1	3	3	12060	301	33
KV	Two threads perform concurrent operations on KV Store with locks	2	9	23	6276	325	102
SEN	Periodic Sensing using software timers	2	3	4	6420	107	15
EM	One thread monitors events and notifies other threads with event groups	3	6	12	2592	113	29
MQ	One thread distribute messages to other threads using queues	4	6	13	2532	166	51
ETL	Extract, Transform and Load dataflow in RIoTBench [58] (*e.g.,* range filter, bloom filter, interpolation, join, annotation, kv store)	5	10	23	3580	709	148
PRED	Predictive analysis dataflow in RIoTBench [58] (*e.g.,* average, kalman filter, distinct count, sliding linear reg., kv store)	5	9	26	3808	440	58
STATS	Statistic summerization dataflow in RIoTBench [58] (*e.g.,* decision tree, multivar linear reg., average, error estimation, kv store)	5	11	27	4884	413	46
TRAIN	Model training dataflow in RIoTBench [58] (*e.g.,* multivar linear reg. training, decision tree training)	4	20	28	9472	511	132

Table 3: Description for Benchmarks and Statistics

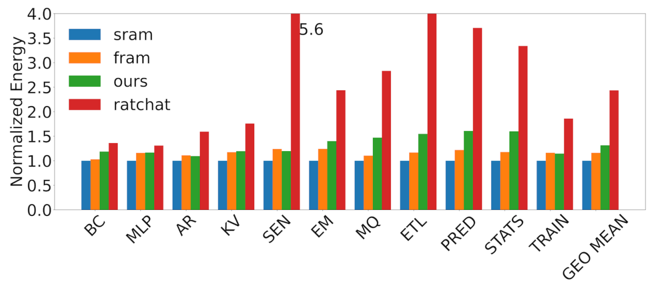


Figure 7: Energy overhead without power failure on MSP430

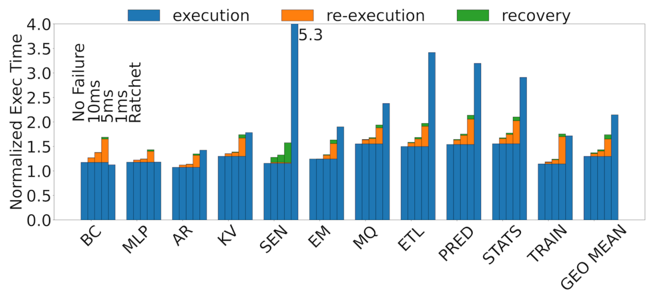


Figure 8: Latency overhead with power failure on MSP430

10.1 Without Power Failures on MSP430

We first measure the performance and energy overhead without power failure on MSP430FR5994.

Figure 6 illustrates the latency overhead of four configurations, normalized to the SRAM baseline. Each bar provides a breakdown between user-level (top, solid color) and kernel-level (bottom, light color) execution times. On a geometric mean, FRAM (the second bar) shows 1.11x latency overhead compared to SRAM. This highlights the performance loss when SRAM is not utilized, as in existing compiler-based checkpointing solutions. This serves as the lower bound for the latency overhead imposed by such tools. Specifically, Ratchet (the last bar) incurs a latency overhead ranging from 1.12x to 5.30x, with a geometric mean of 2.14x. Ratchet's performance is highly dependent on the precision of static analysis and application characteristics.

Contrastingly, INTOS (the third bar) demonstrates substantially lower latency overhead, ranging from 1.07x to 1.55x, with a geometric mean of 1.29x. This showcases the advantages of placing the stack and performing computations on local variables in SRAM while storing persistent objects in FRAM. Notably, for AR, SEN, and TRAIN, INTOS demonstrates comparable or superior performance to FRAM, even considering INTOS's transaction logging overhead.

Regarding the breakdown between user and kernel levels, simple single-thread BC, MLP, and AR predominantly operate in the user level, while multi-threaded RIoTBench's ETL, PRED, STATS, and TRAIN frequently utilize system calls for queues, mutexes, etc. The SEN application conducts periodic sensing using software timers. It displays a small kernel (syscall) time, as the kernel's timer handler indeed runs as a thread and is thus counted as user time.

Figure 7 illustrates the energy overhead of four configurations, normalized to the SRAM-only baseline. We measured the energy consumption for MSP430FR5994 using TI's EnergyTrace tool [11]. The observed trend aligns generally with the latency overhead discussed earlier. The FRAM setting incurs 1.16x (geometric mean) more energy consumption compared to the SRAM setting. The energy consumption gap between INTOS and Ratchet widens, with INTOS consuming 1.31x more energy on a geometric mean relative to the baseline, while Ratchet consumes 2.43x more energy. Notably, across various applications, including MLP, AR, KV, SEN, and TRAIN, INTOS exhibits comparable or superior performance to the FRAM-only setup, even when factoring in INTOS's transaction overhead.

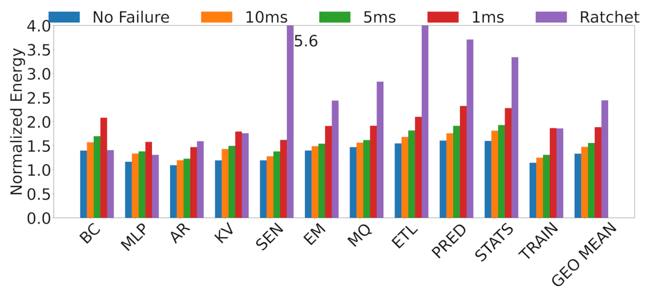


Figure 9: Energy overhead with power failure on MSP430

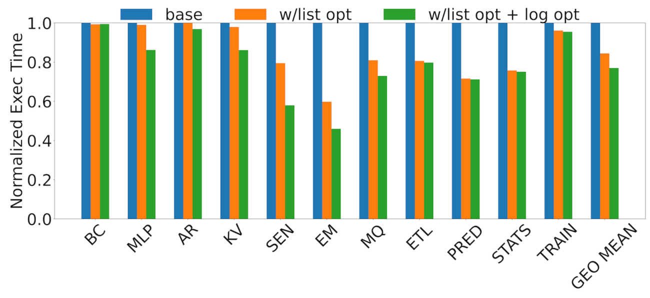


Figure 10: Latency with/without optimizations on MSP430

10.2 With Power Failures on MSP430

In this section, we investigate the latency and energy overhead of INTOS and Ratchet under frequent power failures. We inject controlled power failure at regular intervals of 10ms, 5ms, and 1ms using a soft reset, following the methodology employed in previous works [37,38,49,62,64,65]. Intermittent computing devices continue operation until the energy stored in the capacitor is depleted, subsequently restarting after the capacitor is fully recharged. In cases where the capacitor can recharge during the run, it results in extended run time for the power cycle. Employing a regular power failure interval represents the worst-case scenario, where the capacitor cannot be effectively charged during execution.

To determine and justify the power failure interval, we ran our benchmarks and used EnergyTrace to measure the average power consumption of MSP430FR5994 and the number of MCU cycles spent for each time interval. We considered the maximum power consumption (among applications) and calculated the corresponding capacitor size under a 3V voltage. To sustain continuous operation for 1ms, where MSP430 MCU can run for about 16,000 cycles, the capacitor size required is approximately 4 µF, which is ten times smaller than a typical capacitor size (*e.g.,* WISP [57] utilizes 47 µF). Thus, the 1ms interval represents an extreme case.

INTOS requires each transaction to be completed with a fully charged capacitor to guarantee forward progress. The third last column in Table 3 displays the maximum number of cycles per user transaction in tested applications, indicating that an application can be implemented with a (relatively) short transaction. Should a longer transaction be desired, INTOS might necessitate a larger capacitor.

Figure 8 depicts the latency overhead under power failures on MSP430. Moving from left to right, the bars represent the latency overhead of INTOS in a no-failure scenario, with failure intervals of 10ms, 5ms, 1ms, and Ratchet – all of which are normalized to the SRAM setting (baseline). Note that as an idempotent processing solution, Ratchet exhibits negligible latency difference between with and without power failures. Across different applications, we observed 30-900 power outages with the 1ms failure interval, and 3-60 power failures with the 10ms interval. As anticipated, the latency overhead of INTOS increases with the frequency of power failures. On a geometric mean, INTOS exhibits latency overheads of 1.37x, 1.43x, and 1.73x for 10ms, 5ms, and 1ms intervals, respectively. INTOS's recovery mechanism involves restoring volatile states (while bypassing numerous committed transactions and system calls), making its latency sensitive to the failure frequency. However, for the 10ms, 5ms, and even in the extreme 1ms failure intervals, INTOS demonstrates significantly better performance than Ratchet, particularly when considering realistic complex applications like RIoTBench's ETL, PRED, and STATS, while excluding trivial single-thread applications like BC and MLP.

The figure also provides a breakdown of the latency overhead between re-execution (orange bar) and recovery (green bar). Re-execution overhead involves rerunning an interrupted program region, representing wasted computation, while bypassing committed transactions and system calls. Recovery overhead is incurred by applying undo logging to roll back a failed transaction and executing other basic recovery checking codes. The results indicate that INTOS's latency overhead is predominantly attributed to re-execution overhead. SEN is unique in that it uses software timers, so in most cases, it has no task to run but simply checks for recovery.

Figure 9 illustrates the energy overhead under the same power failure experiments on MSP430. The energy overhead follows a similar trend as the latency overhead. INTOS consistently demonstrates a lower energy profile than Ratchet across all failure intervals, especially when considering realistic applications ETL, PRED, and STATS. On a geometric mean, INTOS exhibits energy overheads of 1.47x, 1.55x, and 1.88x for 10ms, 5ms, and 1ms intervals, respectively. In comparison, Ratchet incurs an energy overhead of 2.43x.

10.3 Optimization Effectiveness on MSP430

This section investigated the impact of linked list optimization (§7.2) and undo logging optimization (§7.3). Each optimization was individually enabled, and the execution time was measured. The results, normalized to INTOS with no optimizations (the first bar), are presented in Figure 10. The second bar illustrates the outcomes with only the list optimization enabled, while the last bar represents the results with both

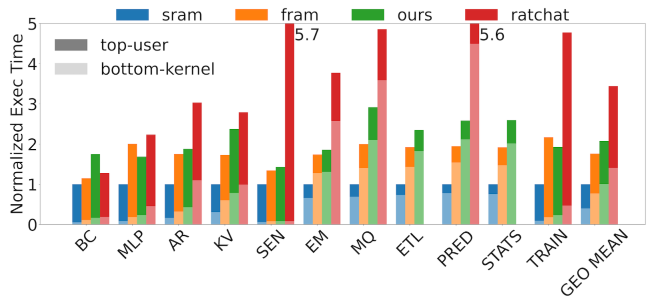


Figure 11: Latency without power failures on Apollo 4

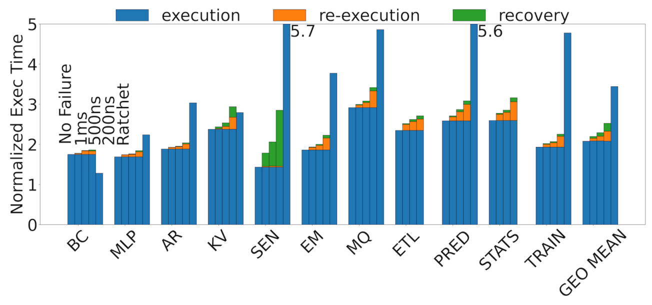


Figure 12: Latency with power failures on Apollo 4

optimizations in use. For applications with frequent system call usage, the list optimization significantly enhances performance, with improvements exceeding 40%. However, for simpler single-thread BC, MLP, and AR, which only utilize memory allocation syscalls, there was marginal improvement. The effectiveness of the undo-logging optimization is highly dependent on application characteristics. MLP, KV, SEN, EM, and MQ have a small fraction of stores with write-after-read dependencies. Thus the undo logging optimization demonstrates substantial improvements.

10.4 Experiments with Apollo 4

Now, we transition our experiment to the Apollo 4 Blue Plus, equipped with an ARM Cortex-M4 MCU, 384KB of TCM (faster SRAM), 2MB of SRAM, and 2MB of non-volatile MRAM. However, it is important to acknowledge that the MRAM in Apollo is presently only byte-readable and not byte-writable. To overcome this constraint, we simulate the execution environment by utilizing (fast) TCM as volatile memory and designating the (slow) SRAM as non-volatile memory. In our experiment, the SRAM is approximately 2-3 times slower than TCM for sequential access, which is bigger than the FRAM-SRAM gap in MSP430. The (simulated) Apollo 4 experiment has two purposes. First, it demonstrates that INTOS can support different MCU architectures: MSP430 and ARM Cortex-M4. Second, it illustrates a scenario in which the latency disparity between volatile and non-volatile memories is more pronounced. The board does not have an on-board debugger probe that allows us to measure the energy, so this experiment focuses on latency comparison.

Figure 11 shows the latency overhead of Apollo 4 Blus Plus without a power failure, normalized to the TCM-only baseline. The ETL and STATS bars are missing for Ratchet because the programs instrumented by Ratchet crashes. With the higher gap between volatile and non-volatile memories (simulated by TCM and SRAM), the result shows higher latency overheads than the MSP430 experiments (Figure 6). INTOS and Ratchet incur 2.07x and 3.44x latency overhead, respectively, where Ratchet is more penalized by slow non-volatile memory.

Figure 12 shows the latency overhead when considering power failure intervals of 1ms, 500ns, and 200 ns. The intervals are set to be much smaller than those of MSP430 as ARM Cortex-M4 in Apollo 4 runs at a much higher clock frequency. 200 ns allows around 19,000 cycle executions. The trend again remains the same. Even in the extreme case of 200 ns failure interval, INTOS incurs 2.52x latency overhead (compared to SRAM). INTOS is 1.37x less than Ratchet.

10.5 INTOS Programming Overhead

The INTOS programming model asks users to allocate persistent objects in NVM and define transactions to ensure crash consistency of updates on persistent objects. Quantifying programming overhead is challenging, but as a proxy, Table 3 presents the lines of source code (LOC) for each application and the added/modified LOC for persistent object allocation and transaction codes. Examining four realistic RIoTBench applications, the table reveals that the extent of modification varies from 11% (STATS: 46/413) to 26% (TRAIN: 132/511) of the source code. Although these percentages may seem large, it is important to note that these changes pertain to persistent object allocation and transaction codes, aspects that we believe are well-understood and manageable.

11 Conclusion

INTOS is a persistent embedded OS and language support for multi-threaded intermittent computing. INTOS uses transactions to ensure the crash consistency of non-volatile objects. Instead of checkpointing volatile states, INTOS proposes a replay-and-bypass recovery mechanism, reconstructing volatile states without re-executing committed transactions and system calls. Evaluation with MSP430FR and Apollo 4 shows that INTOS exhibits lower latency and energy costs compared to compiler-based idempotent processing.

Acknowledgements

We appreciate the valuable feedback from anonymous reviewers and the shepherd. This work is in part supported by the NSF grants CNS-2135157, CCF-2153747, CCF-2153748, CCF-2153749, CNS-2314681, and CNS-2214980.

References

[1] Apollo4 Blue Plus. https://ambiq.com/apollo4-blue-plus/.

[2] ARM Cortex-M4. https://developer.arm.com/Processors/Cortex-M4.

[3] FreeRTOS. https://www.freertos.org/index.html.

[4] Intel Optane Memory. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[5] KickSat. https://kicksat.github.io/.

[6] MSP430FR5994. https://www.ti.com/product/MSP430FR5994.

[7] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[8] Persistent Redis v3.2. https://github.com/pmem/redis/tree/3.2-nvml.

[9] Pmem-Memcached. https://github.com/lenovo/memcached-pmem.

[10] The MRAM on the Apollo 4 Processor. https://www.techinsights.com/blog/memory/disruptive-technology-tsmc-22ull-emram.

[11] TI's EnergyTrace software for MSP430™ MCUs. https://www.ti.com/tool/ENERGYTRACE.

[12] TI's FRAM. https://www.ti.com/lit/wp/slat151/slat151.pdf.

[13] Zephyr. https://www.zephyrproject.org/.

[14] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Gunes, and Thomas Schmidt. *RIOT: One OS to rule them all in the IoT*. PhD thesis, INRIA, 2012.

[15] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. IPSN '17, page 209–219, 2017.

[16] Luca Caronti, Khakim Akhunov, Matteo Nardello, Kasım Sinan Yıldırım, and Davide Brunelli. Fine-grained hardware acceleration for efficient batteryless intermittent inference on the edge. *ACM Trans. Embed. Comput. Syst.*, 22(5), sep 2023.

[17] Wei-Ming Chen, Tai-Sheng Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. Value-based task scheduling for non-volatile processor-based embedded devices. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 247–256, 2016.

[18] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–344. IEEE, 2019.

[19] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–54, 2022.

[20] Jongouk Choi, Qingrui Liu, and Changhee Jung. Cospec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 399–412, 2019.

[21] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. Write-light cache for energy harvesting systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

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

[23] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 475–486, 2012.

[24] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), sep 2020.

[25] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

[26] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: Using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, page 29–39, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021.

[28] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 199–213, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Philipp Gutruf, Vaishnavi Krishnamurthi, Abraham Vázquez-Guardado, Zhaoqian Xie, Anthony Banks, Chun-Ju Su, Yeshou Xu, Chad R Haney, Emily A Waters, Irawati Kandela, et al. Fully implantable optoelectronic systems for battery-free, multimodal operation in neuroscience research. *Nature Electronics*, 1(12):652–660, 2018.

[30] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017.

[31] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017.

[32] Matthew Hicks. Clank: Architectural support for intermittent computation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 228–240, 2017.

[33] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 429–442, 2021.

[34] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, page 193–204, New York, NY, USA, 2019. Association for Computing Machinery.

[36] 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 *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 330–335, 2014.

[37] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. Wario: Efficient code generation for intermittent computing. PLDI 2022, page 777–791, 2022.

[38] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. ASPLOS '20, page 85–99, 2020.

[39] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. Intermittent learning: On-device machine learning on intermittently powered system. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(4), sep 2020.

[40] Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *2012 IEEE International Solid-State Circuits Conference*, pages 402–404, 2012.

[41] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[42] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, 2017.

[43] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[44] Qingrui Liu and Changhee Jung. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2016.

[45] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Notices*, 50(5):1–10, 2015.

[46] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 228–239. IEEE, 2016.

[47] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 575–585, 2015.

[48] Yubo Luo and Shahriar Nirjon. Smarton: Just-in-time active event detection on energy harvesting systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 35–44, 2021.

[49] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[50] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 129–144, 2018.

[51] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1101–1116, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1005–1021, 2020.

[53] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sen. Netw.*, 16(1), feb 2020.

[54] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, ENSsys '19, page 8–14, 2019.

[55] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. ASPLOS XVI, page 159–170, 2011.

[56] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. PLDI 2019, page 1085–1100, 2019.

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

[58] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21), October 2017.

[59] Milijana Surbatovich, Limin Jia, and Brandon Lucia. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 851–866, 2021.

[60] Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. A type system for safe intermittent computing. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[61] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. Capband: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 54–67, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 17–32, 2016.

[63] Harrison Williams, Xun Jian, and Matthew Hicks. Forget failure: Exploiting sram data remanence for low-overhead intermittent computation. ASPLOS '20, page 69–84, 2020.

[64] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 41–53, 2018.

[65] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. Immortal threads: Multithreaded event-driven intermittent computing on Ultra-Low-Power microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 339–355. USENIX Association, July 2022.

[66] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. Replaycache: Enabling volatile cachesfor energy harvesting systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 170–182, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in zebranet. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, page 227–238, New York, NY, USA, 2004. Association for Computing Machinery.

[68] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. Sweepcache: Intermittence-aware cache on the cheap. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1059–1074, 2023.

A Artifact Appendix

Abstract

IntOS is an intermittent multithreaded embedded RTOS based on FreeRTOS for research. It features transactions with replay and bypasses to enable cheap crash consistency. The kernel and user applications are all written in the Rust programming language. The Rust type system is used to enforce safe programming rules defined by the framework to guarantee crash consistency and Persistent Memory safety. Currently, we support three platforms: QEMU, Apollo 4 Blue Plus, MSP430FR5994.

Scope

This artifact contains code to build the crash-safe INTOS kernel and benchmark/user app to run on intermittent computing platforms(e.g. MSP430FR5994). Users can use the artifact to reproduce the results in the paper. For evaluation of functionality, please just follow the instructions for QEMU. This artifact is for research purposes only.

Contents

The artifact contains Rust written kernel, user library code, and the benchmarks used in the paper. The general kernel code is under the `src/` directory. The user library is in `src/user`. Benchmarking code is under `src/benchmarks`. Platform/Architecture-related code is in `src/arch` and `src/board` directory. A simple demo app is hosted under `src/app`.

Hosting

The artifact is hosted in https://github.com/yiluwusbu/IntOS. The branch is master and the commit version is a916c16

Requirements

The OS we use is Ubuntu 22.04. For evaluating the functionality and debugging, QEMU(for ARM) is sufficient. For evaluating the performance, you need to get the MSP430FR5994 or Apollo 4 Blue Plus development board.

Run with Docker

We provide a docker image for users to run the system with QEMU. If you use docker, please skip the dependency/-toolchain installation sections. To build the docker image, run:

```
docker build -t rtosdev .
```

Then, run the docker:

```
docker run -v $(pwd):/repo -it rtosdev bash
```

Install System Dependency

```
sudo apt install curl wget p7zip-full
libncurses5 libncursesw5 build-essential
qemu-system-arm
```

Install Rust Toolchain

```
curl --proto '=https' -tlsv1.2 -sSf
https://sh.rustup.rs | sh -s -- -y
```

Set the compiler version:

```
rustup toolchain add nightly-2022-04-01
```

Install MSP430 Toolchain

Download and install the msp430-gcc toolchain from TI's website. For detailed commands, see README.md in the github repo.

Install ARM Toolchain

Install JLink flasher/debugger and the ARM gcc toolchains:

1. Download the Segger JLink tools (v7.92) on your platform from their website
2. Download ARM (arm-none-eabi) toolchain (version 12.3.Rel1) from the official ARM website

Compile INTOS

You can compile the OS and benchmarks/example applications using the provided Python script:

```
./compile.py --board [qemu|apollo4bp|msp430fr5994]
--bench [app name] [--run (for qemu)]
```

Example:

```
./compile.py --board qemu --bench bc --run
```

Configuration Parameters

To list all the available benchmarks and custom compilation flags, you can run:

```
./compile.py -h
```

Table 3 describes the benchmarks we use in this work. To enable timer daemon, you can pass `--timer_daemon`

Power Failure Injection

To inject soft power failure to the system at a given frequency, you can use the following command:

```
./compile.py --board [board name] --bench [app
name] --fail --pf_freq [frequency: e.g. 1ms]
[--run (for qemu)]
```

Example:

```
./compile.py --board qemu --bench bc --fail
--pf_freq 1ms --run
```

Run Demo App

We give a simple example of two tasks communicating using a Queue (i.e. IPC). The full code can be found in the `app/demo.rs` file.
To run the demo:

```
./compile.py --board qemu --app demo --run
```

Flash and Run App on MSP430FR5994

You can install the TI's Uniflash or CCSTUDIO IDE to flash the application binary (located under target/msp430-none-elf/release/) onto the board.

The application/OS will print debug/perf related information through the UART interface to the host machine. The default Baud Rate is 115200. To view the printed message, you can use any Serial Monitor tools to view the printed message. For example, on Linux/Win, you can install the Serial Monitor Plugin. Termite is another handy tool you can use.

Flash and Run App on Apollo4 Blue Plus

Use GDB and JLink to load and run the application.

1. In one terminal, run `JLinkGDBServer -if SWD -device AMA4B2KP-KXR`
2. In another terminal run `arm-none-eabi-gdb -x apollo.gdb <path/to/binary>`

After the binary is loaded onto the board, enter 'c' to run. The application will print message to the gdb interface and port 2333 (TCP/IP) .