Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems

Emily Ruppel
Carnegie Mellon University
Pittsburgh, U.S.A.
eruppel@andrew.cmu.edu

Brandon Lucia
Carnegie Mellon University
Pittsburgh, U.S.A.
blucia@andrew.cmu.edu

Abstract
Batteryless energy-harvesting devices are computing platforms that operate in environments where batteries are not viable for energy storage. Energy-harvesting devices operate intermittently, only as energy is available. Prior work developed software execution models robust to intermittent power failures but no existing intermittent execution model allows interrupts to update global persistent state without allowing incorrect behavior or requiring complex programming. We present Coati, a system that supports event-driven concurrency via interrupts in an intermittent software execution model. Coati exposes a task-based interface for synchronous computations and an event interface for asynchronous interrupts. Coati supports synchronizing tasks and events using transactions, which allow for multi-task atomic regions that extend across multiple power failures. This work explores two different models for serializing events and tasks that both safely provide intuitive semantics for event-driven intermittent programs. We implement a prototype of Coati as C language extensions and a runtime library. Using energy-harvesting hardware, we evaluate Coati on benchmarks adapted from prior work. We show that Coati prevents failures when interrupts are introduced, while the baseline fails in just seconds. Moreover, Coati operates with a reasonable run time overhead that is often comparable to an idealized baseline.

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

Keywords intermittent computing, event-driven concurrency, transactions

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

PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06...
https://doi.org/10.1145/3314221.3314583

1 Introduction
Batteryless energy-harvesting devices are tiny, sensing, computation and communication platforms [15, 29, 72, 88] that can operate in environments that prevent the use of traditional batteries due to extreme temperatures [25, 57, 87], difficulty of maintenance [16, 38, 45, 68], or restrictions on weight and size [45, 68, 87]. Such a device harvests a weak input power supply, buffering energy in a capacitor until it accumulates a useful amount. The device then uses the buffered energy to operate in a short burst, before powering off to await more energy. A device’s energy buffer size is highly application- and platform-dependent and a power failure may occur at any time, up to hundreds of times per second with a small buffer [12, 52, 69].

The resulting intermittent software execution model on such a device is increasingly well studied [13, 15, 31, 52–55, 69, 82], but intermittent systems have some key, unmet needs. Systems often rely on atomic tasks [5, 13, 31, 52, 55] or checkpoints [33, 54, 58, 82], to ensure that data in volatile and non-volatile memory remain consistent despite frequent, unpredictable power failures. Most prior work on intermittent computing focused solely on memory consistency [13, 33,
solving input/output (I/O), ensuring that computations were timely in their consumption of data collected from sensors [15, 31, 86]. However, no prior work on intermittent computing provides clear semantics for programs that use event-driven concurrency, handling asynchronous I/O events in interrupts that share state with transactional computations that execute in a main control loop. The idiomatic use of interrupts to collect, process, and store sensor results is very common in embedded systems. The absence of this event-driven I/O support in intermittent systems is an impediment to developing batteryless, energy-harvesting applications.

Combining interrupts and transactional computations in an intermittent system creates a number of unique problems that we address in this work using new system support. First, an interrupt may experience a power failure while updating persistent, shared state, leaving the state inconsistent on reboot. As Section 3.1 shows, the inconsistent shared state is likely to remain inconsistent because it is unintuitive to checkpoint and restart an event-driven interrupt’s execution after a power failure. Second, task-based intermittent execution models assume that tasks will repeatedly attempt to execute idempotently, allowing them to selectively buffer data and commit it when a task ends[13, 33, 52, 55, 82]. An unmoderated interrupt may cause a task’s re-execution to be non-idempotent, violating the underlying assumption of task-based intermittent execution systems that allows only selectively buffering state. Consequently, these prior approaches may lose updates or produce inconsistent state in an intermittent execution. An appealing alternative is to disable all interrupts during task execution, with behavior like TinyOS atomics [21, 48]. However, unlike the small amount of code typically protected by TinyOS atomics (e.g., synchronization), intermittent execution requires all code to be in a task; disabling interrupts during any task blocks interrupts for most of a program’s execution.

This paper presents Coati, which adds concurrency control for event-driven I/O to an existing task-based intermittent programming and execution model that does not support interrupts. The key contribution of Coati is to define an execution model that safely serializes atomic, transactional computations with concurrent, event-driven interrupts during an intermittent execution. Borrowing from prior work on event-handling in embedded operating systems (OS) [21, 48], Coati defines events as shown on the right of Figure 1, which are regions of code that atomically process I/O and occur asynchronously. Borrowing from prior work on intermittent systems [13, 31, 55], as well as embedded OS [21, 48], Coati defines tasks, which are regions of code that are atomic with respect to power failures and atomic with respect to events. Coati borrows from prior work on transactional memory [8, 24, 27, 28, 59, 73] defining transactions, which allow sequences of multiple tasks to execute atomically with respect to events. Coati’s support for events and transactions is the main contribution of this work. Coati provides the critical ability to ensure correct synchronization across regions of code that are too large to complete in a single power cycle. Figure 1 shows a Coati program with three tasks contained in a transaction manipulating related variables x, y, and z, while an asynchronous event updates x and y. Coati ensures atomicity of all tasks in the figure, even if any task individually is forced to restart by a power failure.

This work explores the design space of transaction, task, and event implementations by examining two models that make different trade-offs between complexity and latency. Coati employs a split-phase model that handles time-critical I/O immediately in a brief interrupt handler, but defers processing the interrupt’s result until after the interrupted task or transaction completes, ensuring the task or transaction remains atomic. We also examine an alternative buffered model that fully buffers all memory updates made in a transaction and immediately processes events, but on a memory conflict between an event and transaction the event’s memory effects are discarded. In contrast, Coati’s split-phase model is efficient, requiring neither full memory buffering nor conflict detection for transactions and events.

We prototyped Coati as a set of extensions to the C language and a runtime library that ensures safe, intermittent operation while supporting tasks, events and transactions. We evaluated Coati on a set of benchmarks taken from prior work [55], running on a real intermittent energy-harvesting system [15]. The data reveal that Coati prevents incorrect behavior of event-driven concurrent code in an intermittent execution. In contrast, we demonstrate that an existing, state-of-the-art task-based intermittent system produces an incorrect, inconsistent result in nearly all cases. This work makes the following contributions:

- An exploration of the challenges of implementing shared memory concurrency control in an intermittent execution on an energy harvesting, computing system.
- An execution model for concurrency that defines the interaction between transactional computational tasks and asynchronous events in an intermittent execution.
- An API and runtime that support intuitive semantics for task, transaction and event interaction, without the need to reason about complex pre-emptive behavior.
- An evaluation with real applications on real hardware showing that Coati supports event-driven concurrency in an intermittent execution with reasonable overheads, where prior system support fails.

---

1 Concurrent Operation of Asynchronous Tasks with Intermittence
2 Background and Motivation

This paper is the first to simultaneously address the challenges of concurrency control for event-driven I/O and atomicity for computations in an intermittent system. Event-driven I/O faces the challenge of managing asynchronous interactions between a program’s main computational work loop and operations in interrupts. Intermittent execution faces the challenge of spanning a program's execution across unpredictable power failures, while ensuring that memory and execution context remain consistent. Coati is motivated by the combination of these two challenges: handling asynchronous I/O in interrupts during a consistent, progressive intermittent execution. Together, these challenges lead to fundamental correctness problems that are not well-addressed by existing hardware or software systems for intermittent computing [6, 7, 13, 33, 54, 55, 82, 86].

2.1 Intermittent Computing

Energy-harvesting, intermittent systems are embedded computing, sensing, and communications platforms that operate using energy extracted from their environment. While batteries are appropriate for many applications [1, 19, 67], intermittent computing is useful when a system cannot use a battery because of a restriction on a device’s size, weight, operating temperature, lifetime, serviceability, or operating current. Batteryless operation enables use deeply embedded in civil infrastructure [38], in the body [16, 45, 68], and in heavily constrained chip-scale satellites [87].

Several experimental intermittent computing platforms exist [15, 29, 72, 75, 88]. These devices are batteryless and operate by harvesting weak (e.g., µW) input energy sources (e.g., RF, vibration) and buffering the energy in a capacitor. After buffering a quantum of energy, the device operates for a brief interval (e.g., 50ms), using sensors, processing, storing data in volatile and non-volatile [80, 81] memory, and communicating [15, 32, 51, 60]. Operating power is typically much higher than harvestable power and the device rapidly depletes its energy supply then powers off to recharge.

Software on an energy-harvesting device executes intermittently, making progress during operating bursts. At each power failure the execution loses volatile program state (e.g., stack, registers) and retains non-volatile program state (e.g., some globals). Power failures compromise forward progress and can leave program state inconsistent, for example, if power fails during an update to a multi-word non-volatile data structure. Figure 2a, left, shows an excerpt of a plain C program that performs activity recognition, using sensor data. The code loops over a rolling window of data, computing statistics about the data, assembling a feature vector, and classifying the data (the figures omit classification code for clarity). The code includes an interrupt handler (marked with __interrupt) that, when a sensor has data ready, adds them to the window, increases the count of samples collected, and returns. The interrupt asynchronously produces the data that the program classifies. Figure 2b, left, shows the program intermittently executing. As power fails, the intermittent execution does not make forward progress, and repeatedly restarts from main().

Prior work used programming and runtime support to operate reliably, avoiding inconsistent memory states despite frequent, arbitrarily-timed failures [13, 31, 33, 35, 52, 54, 55, 58, 82]. Some approaches [33, 52, 54, 69, 82] collect periodic checkpoints of volatile state to preserve progress. After a power failure, execution resumes from a checkpointed context. Resuming from a naively placed checkpoint can make execution context and non-volatile memory inconsistent, requiring conservatively placed checkpoints at some data dependencies identified by a compiler [82] or programmer [52].

A task-based intermittent programming system asks the programmer to break a program into regions of code that the programming model guarantees will execute atomically and idempotently. A programmer ensures that a task will finish within the device’s energy budget by conservatively testing the code before deployment [15] or using energy debugging tools [12, 14]. A task-based model’s runtime system implementation ensures task atomicity by ensuring that repeated re-executions of a task are idempotent. An idempotent task always produces the same result when re-executed, never overwriting its inputs. The idempotence of tasks allows the runtime system to track the executing task’s identifier and immediately jump to the start of that task after a reboot; a task is a top level function and does not require restoring execution context (e.g. stack and register values). Different systems ensure idempotence differently, using checkpoint-like boundaries [31], static data duplication [13], a compiler-inserted logging [5], and task-privatization [55].

Figure 2a shows the example program re-written to use tasks, in an Alpaca-like [55] language. The code’s main functions map to tasks and arrows indicate control-flow. The interrupt is its own task and has no control-flow arcs because it is asynchronous. Figure 2b shows the task-based program executing intermittently. A task runtime buffers updated values (the figure excludes this "privatization" code [55]). A task preserves progress when it completes, commits its updates, and transitions to another task. After power failure, execution restarts at the most recent task, instead of main(); in the figure, FeaturizeWin resumes after the failure.

This work focuses on adding transactional concurrency control to task-based systems in particular due to performance and programmability considerations. First, tasks have lower runtime overhead than static checkpointing [56]. Second, tasks are a simple lexical scope defining an intermittent failure-safe code region. Third, tasks avoid dynamic checkpoint behavior [7, 35] that is difficult to reason about statically, especially with event-driven I/O.
2.2 Concurrency in Embedded Devices

Embedded systems in cyber-physical applications must asynchronously interact with unpredictable stimuli from their environment often using peripherals to perform I/O. Embedded systems typically handle such asynchronous operations using *interrupts*. An interrupt is a signal triggered by an asynchronous event that is moderated by hardware and eventually delivered to a software *interrupt service routine* (ISR). An ISR can perform application-specific event-handling operations, including interacting with peripherals (i.e., the one that triggered the interrupt), performing arbitrary computation and manipulating variables. An asynchronous ISR preemptes the program’s main thread of control, and may concurrently (although not in parallel) access program state. After an ISR completes, control returns to the point in the program at which the preemption occurred.

Event-driven concurrency of interrupt handlers and program code requires embedded software to synchronize accesses to shared data. Code may synchronize data using mutex locks, reader-writer locks, or semaphores to establish critical regions that atomically read or update data. TinyOS [21, 48] allows specifying *atomic* operations that, in effect, disable interrupts for their duration. One use of atomic is to synchronize direct access to shared data by an interrupt and a program. While atomic program operations execute, interrupts are disabled, instead of immediately being handled. TinyOS-style atomics also allow building synchronization primitives, which may be useful when an application cannot disable interrupts for a long time (i.e., to remain responsive). A key problem addressed by Coati is that task-based intermittent programming systems do not support interrupts and existing concurrency control mechanisms do not gracefully handle interrupt operation.

2.3 Benefits of Interrupts in Intermittent Systems

Event-driven interrupts are crucially important for intermittent systems applications. Recent work has demonstrated the value of local DNN inference on intermittent devices to enable complex, event-driven applications [22]. Without interrupts, event-driven applications must alternate between processing event data and polling for new events. Computationally intensive event processing causes long unresponsive periods because computation monopolizes the processor. The device will not observe a new event until it processes an older one. Intermittent execution increases the length of the unresponsive periods because the application must frequently wait to recharge after depleting its buffered energy.

Figure 3 shows data from a high level software simulation of an event-driven image processing application that captures bursts of events (e.g. a pack of coatis passing by a wildlife camera). The simulation compares the fraction of events captured over an hour using interrupts (the int-* lines) versus polling (the poll-* lines) for a continuously-powered (*-CP) and energy-harvesting (*-EH) system. A burst of 5 events (e.g. coatis in close proximity) occurs with an expected interarrival time of 3 seconds. An event lasts 1.2 seconds, twice the device’s recharge time (i.e., recharging does not cause missed events). Our simulation models powered-on, recharge, and data collection times using measurements of...
the Capybara platform with its onboard MCU at 8MHz [15]. The simulated workload models the intermittent MNIST implementation from prior work [22, 44]. Each event requires 6 seconds of computation while the device is powered on, and the device may enqueue up to 16 events (a 16KB event queue). We assume a low power camera with a separate energy buffer and harvesting source [30]. The simulation shows that even with continuous power, polling captures less than 60% of the events because the system misses events that occur while processing prior events. The effect is exacerbated under harvested energy because the time to recharge extends the time to process the first event in the burst, and prevents the system from capturing any additional events. Introducing interrupts allows the system to capture all events with continuous power. Interrupts on harvested energy (int-EH) converges to capturing 100% of the events once the burst interarrival time is long enough to prevent the event queue from saturating. The data emphasize the need for interrupts in intermittent systems.

Figure 3. Interrupts support bursts of events. Using interrupts, the simulated energy-harvesting device (int-EH) outperforms the continuously powered, polling baseline (poll-CP) if there is time to recharge between bursts.

### 3 The Challenge of Intermittent Event-Driven Concurrency

Naively combined, existing solutions for intermittence and event-driven concurrency can cause incorrect and unintuitive behavior. Event-driven, concurrent execution is incorrect in the presence of intermittence: simply using TinyOS-style atomic concurrency control in an intermittent system behaves incorrectly when power fails. Additionally, both static checkpointing and task-based intermittent execution can be incorrect or inefficient in the presence of interrupts.

#### 3.1 Interrupts + Intermittent Operation

Event-driven code behaves incorrectly if power fails during an interrupt. The key problem is that even using atomic operations, if power fails during an ISR, the ISR may have only partially updated a multi-byte data structure. On reboot, intermittent execution resumes in the most recently executing task or from the most recent checkpoint. If the program accesses the data partially updated by the ISR, the program behaves incorrectly. An (intuitive) alternative approach is to restart execution after a power failure in the context of the ISR. The problem with restarting in an ISR after a power failure is that important device state may be unavailable on reboot (because a peripheral reset). Moreover, the device may have been inoperational for an arbitrary duration, violating timeliness constraints on the ISR [31].

Figure 4 illustrates this “Interrupt Interrupted” problem with example task-based code. The figure adds a task to the program that checks the consistency of the size of Win[]. The interrupt adds new entries to Win[], but power fails before it updates totCnt. When control returns to the task, the assertion fails because the data are inconsistent.

The ISR can produce partial updates even with the privatization analysis of a task-based intermittent execution framework [55, 82], treating the ISR as a task. Privatization analysis identifies data to privatize to a task. A task buffers updates to privatized data and commits updates only when the task completes. Privatization analyses in intermittent execution frameworks assume that a task repeatedly executes until successfully completing. To reduce buffering and commit cost, the analysis only privatizes data that are read, then written by the task (i.e., finding WAR dependences). Such “WAR”-based privatization prevents data written by a failed attempt to execute the task from being visible to a read in a re-execution of that task. Accesses to data not involved in a WAR dependence directly update memory. Privatization analysis works correctly for sequential intermittent programs because they always re-attempt a failed task until it completes. Existing privatization analyses [33, 55, 82] are incorrect with interrupts: if power fails during an ISR that only writes to a multi-byte data structure, a partial update is unbuffered and made visible to the continuation of the interrupted task after power resumes.

#### 3.2 Synchronization + Privatization

Synchronization is also complicated by intermittent execution. TinyOS-style atomics are useful for building synchronization primitives, such as flags and locks, enabling synchronization operations to perform read-modify-update operations that an ISR will not interrupt. Often a program
cannot disable interrupts during a long region of code, barring use of atomic and requiring use of such a synchronization primitive. An intermittent task may leave a critical region by updating a synchronization variable (unsettling a flag or releasing a lock). If the intermittent task system does not privatize the synchronization variable (i.e., because it is not read then written by the task), the update directly modifies main memory. If power fails after a task leaves a critical region, execution resumes from the start of the task which may be inside the critical region. The problem is that the update made to the synchronization variable to leave the critical region remains in memory. After the task restarts in the critical region, an ISR may successfully enter its critical region, leaving both the task and the ISR in the critical region, which is incorrect. This problem also occurs in a checkpointing system if a programmer uses a lock to span multiple static checkpoints. If power fails after the code has released the lock and before it has reached the next checkpoint, on reboot the scheduled code and the ISR will both be able to enter the critical region.

Figure 5 shows the "False Flag" problem, which illustrates how flag synchronization is complicated by intermittent operation. The WindowStats and FeaturizeWin tasks compute related properties of Win[]. Both should see the same values in Win[], requiring them to be in a critical region protecting Win[]. The flag flag controls access to the critical region, preventing the interrupt from entering while WindowStats and FeaturizeWin are executing. It is reasonable to use a flag across multiple tasks, because one very long task would exhaust the device’s energy supply, impeding progress.

The execution sets flag and proceeds through WindowStats. The first attempt to run FeaturizeWin fails just after clearing flag. The task restarts without restoring flag=1 because flag is write-only; privatization does not restore write-only data on reboot [52, 55, 82]. After restart, the task is in the critical region. The interrupt immediately fires, checks flag, sees it clear, and also enters the critical region. The interrupt’s updates to Win[] violate the critical region’s atomicity, leading to an inconsistency between values computed by WindowStats and FeaturizeWin.

Privatization causes a symmetric "False Flag" problem when the interrupt needs to send a signal to the task. A common programming pattern in embedded systems is to wait in the main thread until ISR code updates a shared variable that signals the main thread to continue. For instance, the main thread might wait until a signal variable is set that indicates new data has arrived, then the main thread will process the data and clear the signal. However, in an intermittent execution, if the read and the clear of the signal variable happen in the same task, privatization redirects all accesses of the variable to a private copy. Since the task only accesses its private copy, writes to the variable in the ISR are not visible until the task completes or the device powers down and restarts the task.

4 Intermittent Interrupts with Coati

Coati is a programming API and runtime that allows a programmer to control event-driven concurrency in a task-based intermittent execution system. Figure 6 shows an overview of Coati’s use. Like prior intermittent execution systems (Coati builds on Alpaca [55]), Coati asks the programmer to write their program as a collection of tasks. A task is a function with no callers that can include arbitrary code and explicitly transfers control to another task. Tasks communicate by accessing global variables stored in persistent memory. Coati tasks are atomic with respect to power failures because Coati buffers a task’s updates to memory and commits them on task completion, discarding them on a failure. A task interrupted by a power failure idempotently re-executes until progressing to the next task.

Figure 5. The “False Flag” problem. When power fails after FeaturizeWin clears flag, the task and the interrupt are both in the critical region, violating atomicity.

Figure 6. Overview of Coati. The programmer codes using Coati primitives and links the Coati-enabled program to the Coati runtime, handling both intermittence and interrupts so the app executes correctly when deployed.
Coati allows the programmer to specify events. An event is a special task that a programmer can use to process an asynchronous interrupt. An event in Coati is similar to an event in TinyOS [21, 48] in that an event may be concurrent with a task and an event may share state with a task. The primary difference between tasks and events is that a task does not explicitly transfer control to an event. Instead, an event is associated with an asynchronous interrupt and invoked automatically by Coati on that interrupt’s asynchronous occurrence. Coati also allows the programmer to define a transaction, which is a sequence of tasks that execute together atomically with respect to events, while remaining individually atomic (and idempotently restartable) with respect to intermittent power failures. Like TinyOS [48], Coati assumes that events are short and relatively infrequent so that the application can make forward progress. Further, Coati assumes that the programmer correctly balances transaction length with responsiveness requirements for processing events.

To correctly include events in an intermittent execution, Coati must meet several requirements. First, Coati must preserve task atomicity and idempotence despite asynchronous events by serializing tasks’ and events’ updates to task-shared variables. Second, Coati must support atomic regions that extend beyond one reboot. Finally, Coati should not impose a prohibitive overhead in terms of runtime or memory.

4.1 Interaction Between Tasks and Events

The main contribution of this work is defining how Coati’s tasks, events, and transactions interact. We first describe task-event interactions using the split-phase serialization model used by Coati’s final design. Split-phase serialization forces events to serialize after an interrupted task completes. Returning to Figure 2a’s example code, Figure 7 shows how events and tasks serialize.

**Split-phase Serialization.** Split-phase interactions decouple the asynchronous part of an event (i.e., the ISR and peripheral manipulations) and the shared data manipulation associated with the event. A split-phase event has a top, which runs asynchronously at the interrupt, and a bottom which is scheduled to run after the completion of the task that was interrupted by the top of the event. Similar to tasklets in Linux [50], event bottoms allow an interrupt to defer latency tolerant work and quickly return to the interrupted task. In Figure 7, the top of the event interrupts WindowStats and executes immediately, while the bottom executes after WindowStats completes. After the top of the event completes, execution resumes from the point of the interrupt in the interrupted task, as shown in the figure. The top of a split-phase event is not allowed to access any global, shared state, avoiding the risk of violating a task’s idempotence. Instead, the top of the event can privately buffer data to be processed (e.g., sensor data, received radio packets), preparing them for use by the event’s bottom. In the figure, private data have a ‘: ’x’, y’, z’, and x’ are only accessible by the interrupt’s top and bottom. The event’s bottom, serialized after the interrupted task, is allowed to access arbitrary state. The event bottom buffers all memory updates, like a task, and atomically commits those updates on completion. The figure shows how the event’s bottom adds to Win[] the data sensed by the event’s top and increments totCnt.

Split-phase serialization gives Coati flexibility on power failure. Coati has a configuration option that aborts event bottoms for programs in which an event’s bottom must execute in the same operating period as its top. However, Coati can avoid unnecessarily discarding events by repeatedly executing an event bottom if power fails during the event bottom. Because the part of the interrupt that must be timely is likely to occur in the top of the event handler, the bottom should be able to execute safely even with the delay of a power failure. If multiple split-phase events occur during a single task’s execution, Coati allows their event bottoms to queue and wait for the task to complete. When a task completes, it commits as usual and processes the event queue, executing queued event bottoms in order.

4.2 Multi-Task Transactional Execution

Coati allows the programmer to sequence multiple tasks together into a transaction. A transaction is atomic with respect to interrupting events in the same way that an individual task is atomic with respect to an interrupting event. A transaction is not atomic with respect to power interruptions, but its constituent tasks individually are. Applications need multi-task transactions because some operations that should be atomic with respect to interrupting events may individually consume more energy than the intermittently operating device can buffer. If tasks were the only unit of atomicity with respect to events, then a program using such energy-hungry tasks would have to decide: to split the operations across...
tasks, allowing an interrupting event to violate atomicity, or to include both operations in the same task, preventing the task from completing because it consumes more energy than the device can supply. Multi-task transactions allow tasks to execute sequentially without interruption by an event, eventually committing updates from all tasks.

**Coati Transactions.** Coati supports split-phase serialization for transactions as shown in Figure 8. The figure shows the case where multiple events interrupt during the transaction’s execution, each executing its top half and enqueuing its bottom half to execute after the transaction completes. Each pair of event top and bottom shares its own private buffer (distinguished in the figure using ‘ vs. ’). The operation of split-phase serialization with transactions is identical to the task case, except Coati does not execute event bottoms until after the entire transaction completes, as opposed to awaiting only the interrupted task. The simple implementation of transactions in the split-phase model avoid several key drawbacks of the buffered implementation that is discussed in Section 6.2.

Figure 8. A transaction using split-phase serialization. Each time a sensor event occurs, the top half captures data and enqueues a bottom half to execute after the transaction.

### 5 Implementation Details

Coati’s implementation is a runtime with an API (Table 1) for control flow, persistent state access, and synchronization of events and tasks. We assume hardware with byte-addressable non-volatile memory and atomic word writes.

#### 5.1 Control Flow

Coati maintains a program context that defines the current control state of the program, allowing the system to progress through tasks, enter and exit transactions, and execute and return from events. Tasks are C functions with no return value and no arguments. Programmers use the next_task or tx_next_task statements to transition between tasks, respectively, both outside and inside of a transaction. To ensure task transitions are robust to power failures, the runtime system explicitly maintains a task context object, stored in persistent memory. The task context object holds the address of the start of the current task and the current task’s commit state bit, which indicates whether the task is currently committing. The task context object also holds the context of an ongoing transaction containing the current task and the state of any queued event bottoms that are waiting to execute. Task transition statements update the task context atomically by double buffering the context and swapping the value of the runtime’s global current_context pointer, which points to the active context. To start a transaction, the programmer inserts the tx_begin keyword at the start of the first task in a transaction. tx_begin sets the current context’s transaction state to active. To commit a transaction, the programmer uses a tx_next statement, which atomically commits the updates from the last task in the transaction and redirects control to a specified task.

**Coati Event Implementation.** Coati’s implementation relies on the existing ISR control mechanism and does not require explicitly tracking control transfer between the interrupted task and an event’s top half. Instead, the programmer writes the event’s top half directly in the ISR. At the start of the event’s top half code, the programmer must include Coati’s th_start primitive. th_start ensures that there are resources available to buffer the event’s bottom half, aborting the interrupt if there are not. The programmer uses Coati’s th_return to register an event’s bottom half. In contrast to the top half, an event bottom half is a C function with no return value and no arguments and is associated with an event context object. th_return takes a pointer to an event context object as an argument and adds that pointer to Coati’s event queue, which tracks queued bottom halves in order. After an interrupted task or transaction completes, it checks the event queue. If the queue contains any event bottom halves, control transfers to each of them in FIFO order before moving on to the next task.

#### 5.2 Memory Access

Coati provides atomic updates by buffering all updates to persistent memory in nonvolatile buffers. Each buffer is a statically pre-allocated list of writes to memory (without duplicates). Each entry contains the address, size, and new value for each update. Coati maintains one buffer that is reused for tasks, tasks in transactions, and events because split-phase serialization does not allow an event bottom to run until after the ongoing task or transaction has committed. As we describe below, an update by an event or task of a non-volatile memory location (to a “task-shared” variable [55])
Table 1. A summary of the Coati API. t is a task, ev is an event bottom, x a variable, val a value, and type a C type.

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Data Access</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_task(t)</td>
<td>read(x,type)</td>
<td>tx_begin()</td>
</tr>
<tr>
<td>th_start()</td>
<td>write(x,val,type)</td>
<td>tx_next(t)</td>
</tr>
<tr>
<td>th_return(ev)</td>
<td>th_write(x,val)</td>
<td>bh_read(x)</td>
</tr>
</tbody>
</table>

is stored in its respective buffer until commit. Buffers are statically allocated (which is common in embedded systems) and it is an error to access more data in a task, event, or transaction than any buffer can hold.

**Writes.** To write memory, the programmer uses the write primitive. A task or event write performs a linear search in the private buffer for an entry for the variable being written. If the variable’s address is not in the buffer, Coati attempts to allocate space in the buffer. Exceeding buffer capacity is a programming error and the programmer should specify that Coati should use larger buffers and re-compile. After allocating space in the buffer, Coati writes the update’s address, size, and value to the buffer.

**Reads.** To read memory, the programmer uses the read primitive. The runtime first linearly searches the private buffer for an updated version of the variable. If the search succeeds, the read returns a reference to the buffered value. If the search fails, then the read returns a reference to the value stored in main memory.

**Split-Phase Accesses.** In Coati’s split-phase serialization model, events are split into top halves and bottom halves (Section 4.1). The top half of an event can share data with the bottom half of an event through fields in a Coati event context object that the programmer specifies using evt_var. Each field is automatically, statically replicated by Coati a number of times equal to the maximum number of enqueued event bottoms. The top half can store one of these shared fields using th_write. The bottom half can load from one of these shared fields using bh_read. When an event bottom accesses a field shared with its event top, Coati automatically maps the field to the correct replica based on the event bottom’s position in the event queue. All field data are statically allocated and the memory overhead of queue and fields is programmer-configurable; we used a queue of 16.

### 5.3 Commit

Coati uses two-phase commit to atomically commit buffered updates. Commit begins when execution traverses a programmer inserted next_task or event_return statement, which ends a task or event respectively. Ending a task or event sets its commit state bit and prepares the task context object for the next task that will execute. The runtime points the current_context pointer at this new context and begins writing the buffered updates from the just-completed task or event-bottom to memory. During commit, the runtime uses a non-volatile counter to track the number of buffer entries remaining to commit, decrementing the counter only after the write is complete to ensure that all entries correctly commit, despite power failures.

### 6 Buffi: A Buffering-Based Alternative

To explore the design space of systems supporting intermittent execution and transactional concurrency, we developed an alternative implementation called Buffi. Coati uses split-phase serialization to order events with tasks and transactions. Buffi, in contrast, uses buffered serialization. Buffi buffers all event, task, and transaction state to serialize concurrent updates to shared memory. Buffered serialization necessitates several design and implementation changes.

#### 6.1 Buffering and Serialization

Buffi tasks and events buffer all shared memory updates. As a result, each event may be written as a single block of code that executes immediately after its associated interrupt fires. Events may perform timely operations and manipulate task-shared variables. In Figure 9, the event reads new data from the sensor and updates Win[] and totCnt immediately after it is triggered by the interrupt. The event commits the updates when it completes. After an event commits its updates to memory, control resumes from the beginning of the interrupted task and execution continues; in the figure, the event completes and WindowStats restarts from the beginning. If power fails during a Buffi event, Buffi discards its updates and does not attempt to re-execute it.

#### 6.2 Buffi Transactions

To support buffered-serialization for transactions, Buffi introduces a transaction buffer. Buffi tasks in a transaction consecutively commit their updates to a transaction-private commit buffer that is distinct from main memory. On completion, the transaction commits this transaction commit buffer to main memory, making the transaction’s tasks’ updates

![Figure 9. Task & buffered event interaction. Buffered serialization forces tasks to restart after an event. If the event interrupts a transaction, only the interrupted task must restart.](image-url)
visible to subsequent tasks and events. Buffi events may preempt transactions, but only the interrupted task must be restarted. Completed tasks in the transaction do not need to be rerun. However, events’ updates must serialize after the entire preempted transaction. To preserve the atomicity of the transaction, an event that preempts a transaction commits its updates to a private commit buffer on completion, rather than committing directly to memory.

Before committing buffered event updates, Buffi performs conflict checking, which ensures that the event did not read data that the transaction wrote. If an event that executed during the transaction read from a memory location that a task in the committing transaction wrote, the event would not see the value updated by the transaction because the update would be buffered. Reading this stale value is inconsistent with the event’s serialization after the task, so Buffi discards the events’ buffered updates.

### 6.3 Buffi Implementation

Buffi’s implementation is similar to Coati’s, but is generally more complex. The major differences arise from maintaining additional state to allow for concurrent access to task-shared variables by events and transactions.

**Buffi Events.** Buffi statically allocates an event context object for each programmer-defined event function. The event context object holds a pointer to the start of the event and its commit state. To trigger an event in response to an interrupt, the programmer inserts an event_handler call in the ISR after any device-specific code required to clear the interrupt that was triggered. event_handler sets Buffi’s return_context pointer to point to the interrupted task’s context, and transitions to the event by setting the current_context pointer to point to the triggered event’s context. The programmer places an event_return statement at the end of the event, which returns control to the stored return_context.

**Buffi Memory Accesses.** Buffi maintains three statically allocated buffers: the task buffer, the event buffer, and the transaction buffer to ensure isolation between concurrent updates to task-shared variables. In Buffi, a task in a transaction writes its buffered updates into the transaction’s buffer when the task commits. While a transaction executes, a single event buffer persists across events, allowing events to see updates written by previous events. In tasks and events outside an active transaction, Buffi accesses task-shared variables the same way as Coati (as described in Section 5.2). If a read occurs in a task within a transaction, the runtime first searches through the task’s buffer, then the transaction’s buffer, before reading the value from main memory. In Coati, there is no transaction buffer; the read immediately returns the value from memory. Buffi also tracks the set of writes performed by a transaction and the set of reads performed by events for use in conflict detection. A write in a Buffi transaction updates its write set, adding the address of the written location. A read in a Buffi event first checks if there is an active transaction, and then updates the event’s read set. Recall that Coati need not track write or read sets.

**Buffi Commit.** For tasks and events that occur when no transaction is active, commit follows the procedure described in Section 5.3. To commit a transaction, the transaction must first commit updates made by the last task in the transaction to its transaction buffer. Next, Buffi compares the set of addresses that are updated in the transaction buffer to the set of addresses that were read by any event. If the two sets of addresses overlap, Buffi detects a conflict and discards the event buffer. Next, Buffi writes the updated values in the transaction buffer back to memory. Last, if there was no conflict between an event and the transaction, Buffi commits the event buffer, and continues to the next task after the transaction. Recall that Coati simply commits the updates from the last task in the transaction, and then unconditionally processes the event queue.

### 6.4 Buffer Design

To study the effect of buffer design on Buffi’s memory access latency, we implemented the transaction buffer both as a linear buffer and alternatively as a fixed size, chaining hash tables. The hash table design results in speedup when the transaction commit list is long because linear search in a long list is slower than hash lookup. Table 2 shows the average and maximum number of entries committed at the end of each task and transaction in several benchmark applications (described in Section 7). The data show that a simple, linear buffer often works well because the average number of entries committed at the end of a task tends to be small. The data also show that a hash table is the best option for RSA and CF (full evaluation details are in Section 7).

**Table 2. Commit Statistics.** The average and maximum number of entries committed at the end of each task and transaction in benchmark applications.

<table>
<thead>
<tr>
<th>App</th>
<th>BC</th>
<th>AR</th>
<th>RSA</th>
<th>CEM</th>
<th>CF</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Avg</td>
<td>3</td>
<td>2.4</td>
<td>4.8</td>
<td>9.4</td>
<td>1.6</td>
<td>-</td>
</tr>
<tr>
<td>Task Max</td>
<td>3</td>
<td>6</td>
<td>18</td>
<td>194</td>
<td>135</td>
<td>-</td>
</tr>
<tr>
<td>Tx Avg</td>
<td>4.3</td>
<td>6.2</td>
<td>97.7</td>
<td>29.4</td>
<td>147</td>
<td>-</td>
</tr>
<tr>
<td>Tx Max</td>
<td>5</td>
<td>10</td>
<td>98</td>
<td>32</td>
<td>149</td>
<td>-</td>
</tr>
</tbody>
</table>

### 7 Evaluation

We evaluated Coati using applications from prior work on a real energy-harvesting device and directly compared to a state-of-the-art intermittent computing system. Since no prior systems correctly support concurrent interrupts in an intermittent execution, we compare Coati’s split-phase transactions to three additional concurrency control strategies. Our evaluation shows that Coati avoids failures permitted by existing systems and does so with low runtime overhead and
reduced programming effort, permitting responsive event-driven applications.

### 7.1 Benchmarks and Methodology

We evaluated Coati using the Capybara hardware platform [15]. We used a collection of full applications from prior work [55], modified to use event-driven I/O. The applications collect input, process data and communicate results. To make experiments repeatable, we emulate peripherals with logged data. Like TinyOS [48], we assume that events are short and that they are triggered relatively infrequently. To emulate event arrival, we trigger events via a GPIO pin driven by a continuously powered Arduino [4] for 20ms pulses with an interarrival time drawn from a Poisson distribution with $\lambda = 100ms$ [79]. We use the same event arrival emulation parameters for all benchmarks. We measured runtime using a logic analyzer to capture GPIO pulses at the start and end of each run. We use an attenuated bench supply as a harvested-energy source providing under 10mA, like prior work [15].

We evaluate six configurations: Ideal (which does not complete on harvested energy), Alpaca (which breaks with interrupts on harvested energy), Coati, Buffi, and two additional comparisons Atomic, and Hand-Op. We compare against Alpaca because, at the time of writing, it is the fastest task-based intermittent computing model that supports arbitrary computation. Alpaca avoids the channel management overhead of Chain [13], and allows loops in the task graph unlike Mayfly [31]. To provide idempotent task re-execution in the case of power failures, Alpaca buffers only the task-shared variables involved in write-after-read dependences. At the end of each task, Alpaca commits the buffered variables to memory. Ideal runs the same application code as Alpaca, but the runtime has been modified to remove all buffering and commit code. Ideal fails on harvested energy, but on continuous power it represents a task-based system with the minimum possible overhead. Atomic and Hand-Op use fully buffered tasks, like Buffi, but neither uses transactions. Atomic masks interrupts during critical regions, representing a naive solution to multi-task atomicity. Hand-Op uses hand-optimized code to synchronize tasks and events, demonstrating the programming effort required without Coati.

We modified the applications to use the Coati API to handle events, preserving their task decomposition [55].

**BC**: Bitcount (BC) counts bits set in an array using various algorithms. Its event changes an array index and a transaction ensures each algorithm sees a consistent array. Execution is correct if each algorithm reports the same count. **AR**: Activity Recognition (AR) is a simple machine learning model that classifies data from a three axis accelerometer as moving or stationary. We emulate an accelerometer with our interrupt providing random data. A transaction ensures that the sample window and count remain consistent. **RSA**: RSA Encryption (RSA) uses a fixed, 64-bit key to encrypt a string updated by an event. A transaction prevents string updates during encryption. **BF**: Blowfish (BF) uses a block cipher to encrypt a string updated by an event, using transactions to prevent updates during encryption. We omit BF for Buffi because the device runs out of memory. **CEM**: Cold-chain Equipment Monitoring (CEM) LZW compresses a stream of temperature data generated by an event. The code synchronizes a double buffered sample buffer and ready flag indicating data are available. **CF**: Cuckoo Filtering (CF) stores and queries for values in cuckoo filter. Events insert data, while tasks insert and query the filter. A transaction prevents concurrent, conflicting updates from being lost.

### 7.2 Correctness

We demonstrate that prior task-based intermittent systems do not correctly support interrupts. For each benchmark, we attempted to add synchronization manually to the Alpaca implementation to support interrupts that concurrently modify task-shared state. We used a combination of careful flag synchronization and short atomic (i.e., interrupts disabled) blocks. Alpaca tasks behaved as normal but we prevented Alpaca from privatizing data in ISR code because allowing it causes Alpaca’s atomic commit to fail. The ISR directly updates memory.

Table 3 shows that Alpaca fails for all benchmarks except BC. Columns 6–8 show min/mean/max time to failure. The time to failure varies because failure is dependent on specific experimental event timings and harvested-energy recharge time. We investigated each failure and verified its root cause.

**Table 3. Correctness.** Coati prevents incorrect behavior during intermittent execution. Using Alpaca alone, most applications crash or hang. ✓ indicates correct execution, ✗is incorrect. Mean time to failure (MTTF) varies with event timing, and application behavior, not measurement error.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/✓/✓</td>
<td>0.02/1.5/3.5</td>
</tr>
<tr>
<td>AR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>4.6/26.9/45.4</td>
</tr>
<tr>
<td>RSA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>0.6/0.7/1.4</td>
</tr>
<tr>
<td>CEM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>1.8/18.1/68.8</td>
</tr>
<tr>
<td>CF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>2.8/3.8/5.0</td>
</tr>
</tbody>
</table>

was the interaction of intermittent operation and interrupts. The "False Flag" problem was most common: privatization hides updates to synchronization bits set in the ISR or a task. Consequently, AR, RSA, CEM and BF all overwrite updates made by the event. The error causes CEM to stall indefinitely and the lost update causes visible corruption of AR’s output. In CF, privatization breaks synchronization, leaving event counting statistics inconsistent. BC does not fail even using Alpaca because the program is simple, the event is very short, and the event timing does not lead to a failure.
7.3 Programming Effort

We compared the programming effort required to correctly synchronize the benchmarks with Coati and hand (Hand-Op). We found that transactions in Coati simplify reasoning about correct synchronization in an intermittent execution. Table 4 quantifies the additional code required to manage synchronization for each benchmark with Coati (C) and hand (H). We counted the number of variables, tasks and transitions that had to be added or modified. In Coati, the lines of code were primarily used to start and end transactions. No more than 4 new lines of code had to be added to correctly split the event into a top and bottom. Hand-Op requires additional tasks and carefully written transitions to avoid the "False Flag" problem described in Section 3. CEM and BF nominally required only a few more lines of code to manage double buffers of data by hand, but the accesses to the new variables had to be carefully placed.

Table 4. Programming Effort. Coati (C) reduces the effort to write correctly synchronized code. Synchronizing the code by hand (H) required up to 10x more lines of code to manage extra variables, tasks and transitions.

<table>
<thead>
<tr>
<th>App:</th>
<th>BC</th>
<th>AR</th>
<th>RSA</th>
<th>CEM</th>
<th>CF</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config:</td>
<td>C</td>
<td>H</td>
<td>C</td>
<td>H</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>lines</td>
<td>15</td>
<td>65</td>
<td>7</td>
<td>74</td>
<td>8</td>
<td>70</td>
</tr>
<tr>
<td>variables</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>transitions</td>
<td>7</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>tasks</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

7.4 Events Captured

We evaluated Coati’s ability to effectively capture events in an intermittent execution. Coati avoids losing events due to a power failure while synchronization or a disabled interrupt blocks an event.

![Figure 10. Events Captured](Image)

Figure 10. Events Captured. Under harvested energy, Coati consistently processed 100% of the events that occurred while the device was powered on.

Figure 10 shows the fraction of events that are processed (executed and committed to memory) compared to the total number observed while the device is intermittently powered-on. The key observation is Coati does not disable interrupts and allows the application to process all events that arrive.

Buffi discards updates from events that conflict with transactions, so benchmarks with long transactions such as CF and RSA process fewer events. Disabling interrupts (Atomic), causes the application to lose events without running the ISR code. The effect is exacerbated under intermittent execution because pending interrupt signals stored in volatile memory are cleared on reboot. Hand-Op processes a high fraction of events in all benchmarks except BC. Hand-Op uses a try-lock mechanism in BC to prevent the event from writing to the shared index during a critical region, and the event is rarely able to acquire the lock.

7.5 Performance

We evaluated Coati’s performance on harvested energy and continuous power showing that it has practical overheads compared to an ideal system. On continuous power, Coati is competitive with an "Ideal" Alpaca system that avoids all buffering and commit overheads, but does not run correctly on harvested energy. On harvested energy, our experiments show that all Coati configurations have similar performance.

![Figure 11. Runtime on continuous power](Image)

Figure 11. Runtime on continuous power. For applications that are bound by event arrival frequency, Coati performs as well as the ideal baseline.

Figure 11 shows the runtime on continuous power for each application and configuration and Figure 12 shows the percentage of the runtime spent in different parts of the computation. The data show that progress in AR and CEM are bound by the frequency of events arriving, not by the performance of computing. As a result, Coati performs as well as the ideal case. BF and RSA perform more memory accesses per event, and Coati’s overheads slow it relative to the idealized baseline.

BC computes continuously, regardless of the arrival rate of interrupts and Coati’s overhead on accesses to shared memory degrades its performance relative to the idealized baseline. Buffi’s complicated commit protocol for tasks in transactions contributes to a 2.5x slowdown of Buffi over Coati for RSA.

Figure 13 shows the end to end runtime for each of the applications while the device is powered by harvested energy. The runtimes include the time to recharge and to reinitalize the device on each reboot, which account for approximately 85% of the total runtime (the device is operational for about
85ms before spending about 900ms recharging). The Ideal and Alpaca runtimes are omitted because all of the applications failed to complete correctly. The fastest configuration varies because the overhead incurred by each of the configurations is determined by specific application characteristics. Overall, Coati provides performance that is better or within standard deviation of Hand-Op on all benchmarks without incurring the additional programming overhead of Hand-Op or losing events like Atomic.

Figure 13. Runtime on harvested energy. Coati's performance is comparable to Hand-Op's without the additional programming overhead.

8 Related Work

There are several areas of work related to Coati. Intermittent computing is most relevant. Coati relates to work defining synchronization in embedded systems and research on non-volatile memory systems. Transactional memory shares some mechanisms with Coati.

Intermittent Computing. Intermittent computing is a field that emerged at the intersection of energy-harvesting [39, 62, 63, 83, 85] and computer systems [37, 74], including computational RFID [17, 51, 64, 72, 78]. Hardware emerged enabling researchers to use energy-harvesting computers [1, 15, 29, 34, 72, 75, 88]. Leveraging these, prior work developed software that operates intermittently [6, 7, 10, 35, 58, 69]. Programming and execution models [13, 31, 33, 52, 54, 55, 82] emerged that enable correct operation with non-volatile memory. These efforts identified and solved key problems of memory consistency using privatization analysis that buffers and commits only data involved in WAR dependences. A contribution of this paper is to show that the underlying assumptions of privatization are violated by interrupts.

Mayfly [31] studied I/O in an intermittent context, but unlike Coati focused on the timeliness of I/O processing; Mayfly is thus complementary to Coati. InK [86] is a reactive kernel that supports event-driven data processing and scheduling on batteryless devices. InK, like Coati, allows programmers to write intermittent applications that rely on interrupts. However, InK presents a rigid memory model that restricts the scope of shared variables to within "task threads", series of tasks scheduled by an event. Pairs of task threads may only communicate using unidirectional "pipes". In contrast, Coati supports arbitrary access to shared memory. InK's task threads can be preempted at a task granularity, so multi-task atomic regions have to be designed implicitly by carefully placing pipe writes. Coati allows the programmer to explicitly define transactions composed of multiple tasks. Finally, InK's dynamic scheduler complicates reasoning about when tasks execute, while Coati gives control to the programmer.

Concurrency in Embedded Systems. There is a long history of concurrency control research [18, 42, 43]. Prior work [77] provides a survey. The most related efforts on synchronization for embedded systems are TinyOS [47, 48] and nesC [21]. These are frequently-used and provide atomic statements to serialize synchronous tasks and asynchronous events. Other work follows suit [20, 46, 49].

Non-Volatile Memory. The emergence of byte-addressable, non-volatile memory has led to the development of strategies for improving the performance of fault-tolerant, persistent data structures. Persistency models define allowable reorderings of persists to non-volatile memory [23, 41, 65, 66]. Hardware [36, 41, 61, 89] and software [2, 11, 40, 84] support for multi-threaded application programming with persistent memory have been explored. Like these prior works, Coati provides crash consistency for concurrent updates to persistent memory. However, prior work targets large scale systems while Coati ensures data consistency and forward progress under extreme resource constraints.

Transactional Memory. Coati's synchronization model — especially its multi-task transactions — takes a direct cue from a long history of work in transactional memory systems. Transactional memory systems started as mechanisms for manipulating small multi-byte data structures atomically [26, 27, 73], but have grown into a broad research area with support in hardware [28, 59, 71], support for unbounded transactions [3, 8], and support for exotic serialization models [9, 70, 76]. While similar in principle to Coati, the purpose of most prior work on transactions was to synchronize systems that use multiple concurrent threads to perform parallel computations. Between Coati and these prior efforts there...
are many common ideas: update buffering and commit, conflict detection, atomicity, serialization, and speculation and rollback. Coati draws inspiration from work on transactions, recruiting mechanisms to the specific purpose of synchronizing event-driven interrupts with task-based intermittently executing programs on tiny, energy-harvesting devices.

9 Conclusion

This work is the first to study interrupt-driven concurrency in intermittent systems, showing that a naive attempt to combine the two will cause programs to misbehave. Coati is the first system to correctly support interrupt-driven concurrency in an intermittent system, providing a familiar interface with synchronous computational tasks and asynchronous interrupt-driven events, all robust to intermittent operation. Coati provides transactions allowing sequences of tasks to be atomic with respect to events, and provides two different serialization mechanisms for events, tasks, and transactions. An evaluation comparing to a state-of-the-art task-based intermittent system showed that Coati safely enables interrupts in applications that otherwise catastrophically fail in seconds. Coati has practical overheads, comparable to an idealized baseline system.

Acknowledgments

We thank the anonymous reviewers for the valuable feedback and we thank June Andronick for shepherding our work. We thank members of the Abstract research group at CMU for insightful early discussion on these ideas. This work was supported in part by the CONIX Research Corporation (SRC) program sponsored by DARPA.

References

Transactional Concurrency Control for Intermittent...

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


[60] Sayan Naderiparizi, Aaron N Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R Smith. 2015. WISPCam: A battery-free RFID camera. In RFID (RFID), 2015 IEEE International Conference on. IEEE.


[85] A. Wickramasinghe, D.C. Ranasinghe, and A.P. Sample. 2014. WIND-Ware: Supporting ubiquitous computing with passive sensor enabled RFID. In RFID (IEEE RFID), 2014 IEEE International Conference on.


