Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers

Eren Yıldız, *Ege University;* Lijun Chen and Kasim Sinan Yıldırım, *University of Trento*

https://www.usenix.org/conference/osdi22/presentation/yildiz

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers

Eren Yıldız
Ege University, Turkey

Lijun Chen
University of Trento, Italy

Kasım Sinan Yıldırım
University of Trento, Italy

Abstract

We introduce Immortal Threads, a novel programming model that brings pseudo-stackful multithreaded processing to intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution and write their applications in a multithreaded fashion using common event-driven multithreading primitives. Our compiler fronted transforms the stackful threads into stackless threads that waste a minimum amount of computational progress upon power failures. Our runtime implements fair scheduling to switch between threads efficiently. We evaluated Immortal Threads on real hardware by comparing it against the state-of-the-art intermittent runtimes. Our comparison showed that the price paid for the Immortal Threads is a runtime overhead comparable to existing intermittent computing runtimes.

1 Introduction

Advancements in low-power electronics and energy harvesters exploiting ambient sources (e.g., solar [20], indoor light [21], and radiofrequency [27]) paved the way for sustainable systems that can work without batteries. Recent studies have demonstrated promising examples of these systems, such as body implants [23] and long-lived wearables [51], where continuous power is not available and changing batteries is difficult. There are several microcontroller-based batteryless computing platforms (e.g., WISP [46], Flicker [24], Camaroptera [42] and Engage [16]) developed by the researchers. Instead of a battery, these platforms comprise a capacitor that powers all hardware components, including the ultra-low-power microcontroller (MCU), sensors, communication circuitry, and other peripherals. When a batteryless platform consumes the energy stored in its capacitor, it turns off due to a power failure. The platform charges its capacitor until the stored energy exceeds an operating threshold, which turns on the platform again. Therefore, the software on batteryless platforms runs *intermittently* due to frequent power failures and charge-discharge cycles.

Each power failure clears the CPU registers and the volatile memory during an intermittent execution. Hence, the computation might not progress forward since the control returns to the application's entry point [11]. Moreover, power failures may cause data stored in non-volatile memory to be partially updated, leading to memory inconsistency [43]. The prior art proposed mainly two approaches to overcome these issues. The first one is to place checkpoints in program source [6, 8, 26, 28, 30, 31, 33, 36, 44, 53], which store their continuation (i.e., the control state including the registers, stack and global data) in non-volatile memory. After a power failure, control resumes from the latest successful checkpoint location. Another approach is to employ a task-based programming model [10, 19, 25, 34, 37, 38, 45, 54], which requires programmers to implement their programs as a collection of tasks and transitions between them. This model eliminates the cost of checkpoints, since the all-or-nothing semantic of tasks, defined by the programming model, means that a function pointer to the current task is enough to represent the continuation of the program, which makes saving and restoring it from non-volatile memory extremely cheap [10].

Despite efficiency, the task-based model poses significant problems in developing *event-driven* applications [17, 32]. This situation prevents the widespread adoption of intermittent systems since most sensing applications are event-driven.

P1-Event Handling Complexity: Event handling, in general, is implemented in the form of state machines that require explicit management of states and transitions [18]. Implementing event-driven applications using the task-based model requires programmers to manage (i) task partitioning, (ii) task-based control flow, (iii) event states, and (iv) state transitions simultaneously. This situation creates an excessive cognitive burden concerning event-driven intermittent computing.

P2-Limited Concurrency: Existing event-driven task-based systems (e.g., [45, 54]) cannot fully support preemptive threads. Since tasks execute atomically, they voluntarily yield the control, and other tasks cannot preempt them. Moreover, tasks cannot block on events, trigger new threads of execution, and notify the completion of event processing. Therefore, pro-

grammers need to partition long-running computation (e.g., compression [29]) into a set of tasks to avoid missing events.

P3-Wasted Progress: Partial execution of tasks (due to power failures) leads to loss of computational progress within tasks since tasks have all-or-nothing semantics. This issue increases event response time, which is critical for event-driven systems. Recent work proposed loop continuation to preserve computational progress after each loop iteration by selectively *violating* the task-based model [22] (see Sections 1 and 6).

Problem Statement. Considering the mentioned problems, we seek a programming model that:

(Req.-1) removes the cognitive load of the task-based model while retaining its lightweight characteristics;

(Req.-2) brings the flexibility of preemptive multithreaded programming to intermittent systems;

(Req.-3) enables progress from the point where a thread has been interrupted due to a power failure.

Challenges. Fulfilling these requirements is not trivial. To satisfy (Req.-1), Kortbeek et al. [31] proposed giving up the task-based model and using lightweight and sparse checkpoints that save all registers and only the memory segments modified by the program. However, to fulfill (Req.3), checkpoints need to be placed almost at each line in the code. This situation creates an unmanageable overhead even with these lightweight checkpoints. Finally, concerning (Req.-2), we note that it is not enough to use a checkpoint runtime on top of an existing multithreaded OS, since OS primitives such as mutexes, semaphores, as well as interrupt handling, must be implemented taking the intermittence into account, to avoid memory consistency issues due to partial updates.

Contributions. In this paper, we introduce Immortal Threads that brings *pseudo-stackful* preemptive multithreaded programming model to event-driven intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution and write their applications in a multithreaded fashion using plain C without tasks (see Figure 1). Immortal Threads compiler fronted transforms stackful threads into stackless threads, inserts ultra-lightweight checkpointing mechanisms under the hood to minimize wasted progress, and maintains the memory consistency. The Immortal Threads library implements a preemptive scheduler to switch between threads and provides common event-driven primitives such as semaphores and blocking event wait operations. Our real-world experiments showed that Immortal Threads has runtime overhead comparable to the prior art intermittent runtimes InK [54], Alpaca [34] and TICS [31]. Moreover, during frequent power failures, Immortal Threads reduced execution time and wasted work by up to 40% and 90%, respectively.

In summary, Immortal Threads introduces the following contributions:

(1) Preemptive Multithreading: For the first time, we enable preemptive multithreading for event-driven intermittent systems, which provides programming flexibility and eliminates the cognitive burden of task-based programming.

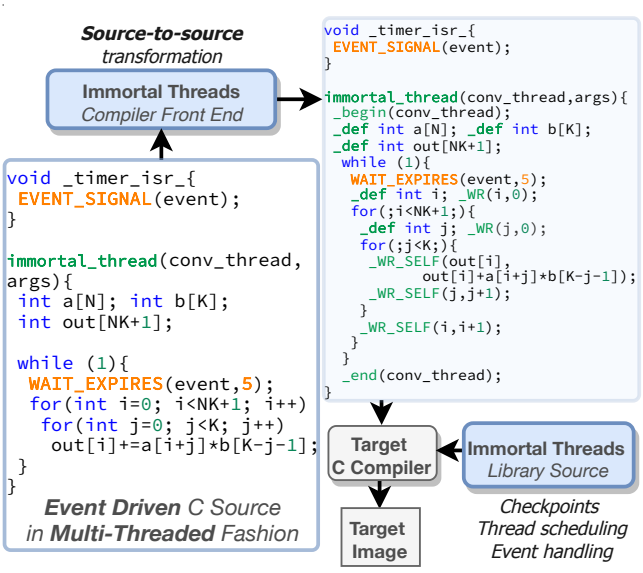


Figure 1: With Immortal Threads, programmers write applications in a multithreaded fashion without concerning intermittent execution, and focus **only** on *event-driven* aspects.

(2) Almost-Free Checkpoints: We propose a novel checkpointing technique, inspired by Dunkels et al. [18], that saves *only* the program counter rather than all registers and memory.

(3) Just-in-time Privatization: We propose a novel technique that eliminates the need for creating static versions of non-volatile program variables to keep memory consistent.

(4) Micro Continuation: Thanks to almost free checkpoints and just-in-time privatization, threads always progress from their latest memory update, and they do not waste computational progress upon power failures.

(5) Open-source Release: We release Immortal Threads as a C library with compiler support (via [55]) for the widespread adoption of intermittent computing.

2 Background and Related Work

Batteryless computing platforms comprise ultra-low-power MCUs with embedded non-volatile memory. For instance, MSP430FR5969 [48], one of the mainstream MCUs used in batteryless platforms, has 64kB of FRAM [50] and 2kB of SRAM memory. FRAM stores data that will persist upon power failures. The key challenges of intermittent computing are the loss of computational progress after power failures and memory inconsistency issues. Power failures reset the MCU, and the control returns to the application's entry point. Moreover, power failures might keep persistent variables (i.e., variables maintained in non-volatile memory) partially updated and in an inconsistent state. Code blocks with WAR

(Write-After-Read) dependencies on persistent variables are not idempotent, since they might produce different results when the MCU re-executes them after a power failure [43]. For example, assume that `x` is a persistent variable and the program executes `{x++; vector[x]=v;}`. A power failure after `x++` re-executes `x++` and leads `x` to be increased twice.

2.1 Intermittent Computing Approaches

The prior art focused on the forward progress and memory consistency aspects of intermittent execution but also considered the timeliness of data processing and event-driven concurrency.

Checkpoints. In energy-guided checkpointing, the device continuously monitors the capacitor to perform a checkpoint on imminent power failure, for example, as in Hibernus [6]. However, voltage monitoring is quite expensive in terms of energy consumption [52]. In software-only checkpointing (e.g., DINO [33], Chinchilla [36] and TICS [31]), the program source is instrumented with checkpoints, either by a programmer or a compiler. The checkpoints are double-buffered in non-volatile memory to prevent the latest consistent checkpoint from being superseded immediately by an inconsistent (i.e., partially updated) checkpoint. Moreover, a compiler analysis is required to determine the modified persistent variables between two checkpoints and create their versions to prevent violations of idempotency upon resumption [33]. After a power failure, the checkpointing runtime restores the versions that isolate the code from partially updated versions, and the control resumes from the latest successful checkpoint location. There are several other works that aim to reduce the overhead of checkpoints [3, 4].

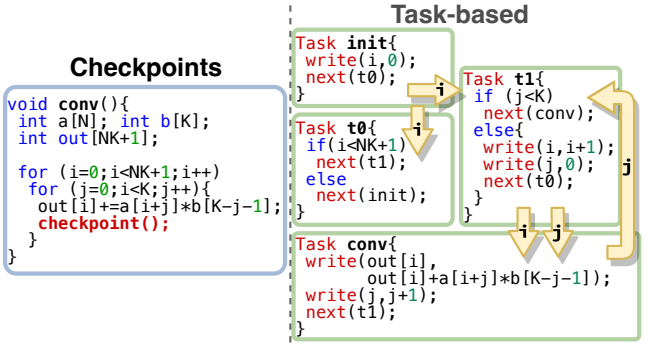


Figure 2: The task-based and checkpoint-instrumented versions of a 1-D convolution code. Arrows among the tasks denote channels that hold versions of task-shared variables.

Task-based Model. This model eliminates the overhead of checkpoints by proposing tasks that do not have a restoration cost. Tasks have read-only inputs and write-only outputs (called channels [10]), which are maintained in non-volatile memory separately. Tasks are inherently idempotent since separate channels avoid WAR dependencies in the task body. Runtimes execute tasks atomically with all-or-nothing semantics. The task-based model employs static multi-versioning by creating multiple copies of the data distributed over the inputs and outputs of the tasks (see Figure 2). Alpaca [34] avoided multi-versioning by proposing *privatization* that creates local copies of the task-shared persistent variables. Each task loads its local copies with the original values, manipulates local copies, and commits them to the original locations upon completion.

Timely Execution. Data (processing) might expire due to charging times during intermittent execution. Mayfly [25], InK [54] and CatNap [37] proposed extensions to the task-based programming model to express timely data constraints and time-critical code. TICS [31] added extensions to checkpointing systems to enable timely data processing.

Event-driven Intermittent Computing. InK [54] proposed task threads, which are triggered by events to execute a sequence of tasks. Coati [45] handles the event-driven concurrency issues by serializing concurrent interrupts with the tasks to keep the shared persistent state consistent. CatNap [37] isolated energy for reliable intermittent execution of periodic events, which are time-critical tasks in a task-based model. TICS [31] does not support event-driven constructs, but it can checkpoint event-driven legacy code—though these checkpoints do not guarantee the semantically correct operation [31] (see Section 5.1).

2.2 Embedded Concurrency Models

Several studies proposed different concurrency models for embedded systems [2, 7, 29, 32, 32, 40, 41]. We classify the main differences in these models into two spheres: whether the concurrency is cooperative or preemptive, and whether the concurrency units (i.e., "threads") are stackful or stackless.

Concurrency Approaches. Stackful concurrency has programming expressiveness, since continuations can be set anywhere in the thread's call stack by preserving the local variables [29]. It is costly since each thread requires its own stack, and thread preemption requires storing all registers into the continuation. In stackless concurrency, threads share the same stack, and local variables are not maintained when a thread blocks. As an example, Protothreads [18] implements a stackless cooperative concurrency model. Consequently, a protothread can yield control only within its main body, not within the body of a function it calls.

Concurrency and Checkpoints. The former sets continuations (a.k.a. execution context) to switch context among threads, while the latter sets them to restore the program state after reboot. Therefore, from the perspective of concurrency, software-only checkpointing systems are non-preemptive, in the same way stackful threads voluntarily set a continuation and yield control. Besides, energy-guided checkpointing systems are preemptive since they stop thread execution and set a

Intermittent Runtimes	Main Features							Timely, Event-Driven Intermittent Program Development
	Task-based or Checkpointing	*Run-time Overhead*	*Event-driven Support*	*Cognitive burden*	*Lost Work*	*Micro Continuation*	*Timely Execution*	
Dewdrop [9], Mementos [44], DINO [33], HarvOS [8]	Checkpointing	High ✗	No Support ✗	Low ✓	Low to High ✗	No ✗	No ✗	N/A ✗
Ratchet [53]	Checkpointing	Very High ✗	No Support ✗	Low ✓	Very Low ✓	No ✗	No ✗	N/A ✗
Chinchilla [36]	Checkpointing	Medium ✗	No Support ✗	Low ✓	Low ✓	No ✗	No ✗	N/A ✗
Chain [10], Coala [38], Alpaca [34]	Task-based	Low ✓	No Support ✗	High ✗	High ✗	No ✗	No ✗	N/A ✗
Mayfly [25]	Task-based	Low ✓	No Support ✗	High ✗	High ✗	No ✗	Yes ✓	N/A ✗
TICS [31]	Checkpointing	Medium ✓	No Support ✗	Low ✓	High ✗	No ✗	Yes ✓	N/A ✗
InK [54], Rehash [5]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	Yes ✓	Difficult ✗
Coati [45]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	No ✗	Difficult ✗
CatNap [37]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	Yes ✓	Difficult ✗
Immortal Threads (this work)	**Checkpointing** (almost zero overhead)	**Low** ✓	**Full Support** ✓ (multithreading)	**Very Low** ✓ (almost zero)	**Very Low** ✓ (almost zero)	**Yes** ✓	**Yes** ✓	**Easy** ✓(almost the same as in continuous systems)

Table 1: A comparison of the main features of Immortal Threads with the relevant intermittent computing approaches.

continuation upon an imminent power failure. Intuitively, the task-based model is a form of static non-preemptive stackless checkpointing system. Static and non-preemptive because task decomposition is done at programming time and checkpoints are taken only at task boundary, and stackless because only the active task's function pointer is checkpointed. Similar to stackless threads, the low-overhead of the task-based model comes at the cost of imposing a programming model with a high cognitive load.

2.3 Drawbacks of Prior Works

Table 1 presents a comparison of the main characteristics of this work and the existing intermittent computing approaches.

1- Event-handling Complexity with Tasks. The task-based implementation of a small deep neural network (DNN) inference in Gobieski et al. [22] has 18 tasks and 61 control flow declarations. Implementing an event-triggered state machine using tasks is even more complex. For example, a low-level radio driver depicted in Dunkels et al. [18, Table 1] has 26 explicit states and 32 state transitions. Implementing this driver using existing task-based event-driven intermittent runtimes [37, 45, 54] requires handling task partitioning and control flow, states, and transitions simultaneously, which is an unmanageable cognitive load.

2- Programming Model Violations. Power failures lead to the waste of computational progress (and energy) when they prevent the execution from reaching the successive checkpoint or the end of the current task. For example, a power failure in the middle of the convolution task while performing the DNN inference in Gobieski et al. [22] might lead to the loss of almost 150000 multiplications. Prior work proposed loop continuation [22] that avoids wasted work by allowing tasks to directly modify non-volatile memory in a loop nest, which is a violation of the task-based model.

3- Limited Concurrency. None of the existing intermittent systems supports the stackful preemptive concurrency model. As Yildirim et al. [54] comments, checkpointing an existing preemptive multi-threading operating system is not practicable due to the inefficiency issues and the memory inconsistencies caused by intermittence-unaware interrupt handling. Similar concerns hold for existing works (e.g., [41]) that can transform stackful threads into stackless continuations for continuously powered systems. For the sake of efficiency, many existing work on intermittent computing utilizes a lightweight stackless cooperative concurrency approach via tasks [37, 45, 54].

3 Immortal Threads: Overview

Immortal Threads consists of a programming interface, a compiler frontend, and a small run-time library, which bring *pseudo-stackful* preemptive multithreading model into intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution, and they develop their programs in a multithreaded fashion as they are programming a continuously powered system. The compiler frontend transforms the source code into stackless continuations that handle intermittency without programmer intervention.

As depicted in Figure 1, the main building block of an intermittent event-driven application is the thread of execution that continues running from where it left upon power failures, which we call *immortal thread*. Unlike the task-based model (which requires explicit idempotent code generation via task splitting), the unnecessary details of the intermittent execution are not visible to the programmers. The duties of the programmers are to (i) identify the events in their system, (ii) design their systems as a set of threads that are the handlers of these events, (iii) manage the necessary state management and state transitions, and (iv) consider timing aspects during event-driven intermittent execution. It is worth mentioning that duties (i)–(iii) are identical to the steps followed to develop event-driven applications in continuously powered systems [17, 32]. Differently, in (iv), programmers embed (if required) the necessary program logic to check event expiration due to the delays stemming from the charge/discharge cycles during the intermittent execution.

Language Construct	Explanation
`_SEM_WAIT(sem)/_SEM_POST(sem)`	wait on/post semaphore **sem**
`_SEM_POST_ISR(sem)`	post semaphore **sem** in an ISR
`_EVENT_SET_TIMESTAMP(e, t)`	sets the timestamp of **e** as **t**.
`_EVENT_GET_BUFFER(e)`	returns a pointer to the data buffer of **e**
`_EVENT_GET_TIMESTAMP(e)`	returns the timestamp of **e**
`_EVENT_WAIT(e,buf)`	blocking wait on **e** w/o expiration time, returns the event data via **buf**
`_EVENT_WAIT_EXP(e, buf, t)`	blocking wait on **e** w/ expiration time **t**, returns the event data via **buf**
`_EVENT_SIGNAL(e)`	signals the event and unlocks the thread waiting on the event

Table 2: Immortal Threads core language constructs.

Language Construct	Explanation
`_begin(name)/_end(name)`	immortal body start/end
`_def/_gdef`	pseudo local variables and persistent global variables
`_WR(arg,val)`	**arg** = **val** (variable assignment operations w/o W-A-R, e.g., `x=5`)
`_WR_SELF(type,arg,val)`	**arg** = **(type) val** (variable assignment operations w/ W-A-R, e.g., `x++`)
`_call(name,...)`	call immortal function **name** with appropriate arguments

Table 3: Main interfaces used by the compiler frontend.

3.1 Programming Model

Immortal Threads supports the common multithreaded event-driven language constructs, as presented in Table 2.

Timely Events and Blocking Wait. Immortal Threads provides an event primitive that builds a bridge between threads and ISRs (interrupt service routines). Threads can block (i.e., wait) on events using `_EVENT_WAIT` and `_EVENT_WAIT_EXP` interfaces, which suspend threads until the relevant event occurs. Signaling an event via the `_EVENT_SIGNAL` interface unblocks the waiting thread to continue its execution. Immortal Threads cannot guarantee event handling deadlines, but programmers can provide an expiration time to catch outdated events and prevent unnecessary event processing. To detect event expiration, programmers can use `_EVENT_WAIT_EXP`, which subtracts the current time from the timestamp of the event. This interface unblocks the corresponding thread if the result of the subtraction is less than the expiration time provided by the programmer. Blocking wait interfaces also pass a pointer to the event data to let the waiting thread copy these data into its thread-local buffer.

Wait and Post Semaphores. Immortal Threads provides a binary semaphore implementation for inter-thread signaling. A thread can block (wait) on a semaphore using the `_SEM_WAIT` interface. Another thread can post this semaphore using `_SEM_POST` interface to unblock that thread. ISRs can also post semaphores using a separate interface `_SEM_POST_ISR`.

ISRs and Event Signaling. In Immortal Threads model, interrupts have *all-or-nothing* semantics. ISRs interface with the hardware, obtain the data, and deliver it to threads. Each ISR has an associated `event` structure. When an interrupt (i.e., an event) occurs, the ISR obtains a pointer to the event data buffer via the `_EVENT_GET_BUFFER` interface. ISRs store the event data (e.g., the sensor reading) into this buffer and set the event timestamp via `_EVENT_SET_TIMESTAMP`. ISR commits these changes atomically and notifies the waiting thread via the `_EVENT_SIGNAL` interface. A power failure up to this point might lead to an event loss. Otherwise, the notified thread will obtain the event data and perform the necessary processing.

```
_immortal_thread(thread1){
 /* the main thread body */
 _begin
 /* variable declerations */
 _def uint8_t cnt;
 ...
  while(1){
   _EVENT_WAIT(event);
   /* necessary actions */
   _WR_SELF(uint8_t,cnt,cnt+1);
   ...
   /* notify thread 2 */
   _SEM_POST(sem);
   ...
  }
 _end
}
```

```
void _timer_isr_{
 /* notify event */
 _EVENT_SIGNAL(event);
}
```

```
_immortal_thread(thread2){
 /* the main thread body */
 _begin
 ...
  while(1){
   _SEM_WAIT(sem);
   /* necessary actions */
   _call(other_function);
   ...
  }
 _end
}
```

Figure 3: Output of the compiler frontend. Initially `thread1` and `thread2` are blocked. The timer ISR signals the event and unblocks `thread1` that unblocks `thread2`.

3.2 Execution Model and Multi-threading

Immortal Threads employs a multithreaded preemptive execution model by implementing a simple but efficient Round-Robin scheduling mechanism. Threads are initially blocked on events, waiting for ISRs to signal them. When an interrupt is triggered, the corresponding ISR signals an event, and the event handler thread wakes up and performs the computation. Therefore, there might be several threads running simultaneously during the execution of programs. Thanks to the compiler frontend, Immortal Threads manages the forward progress and memory consistency without programmer intervention.

3.3 Pseudo-Stackful Threads

The Immortal Threads compiler frontend performs a source-to-source transformation to convert the stackful threads into stackless continuations by employing almost-free checkpoints and just-in-time privatization. To do so, it uses the interfaces in Immortal Threads library (see Table 3). After the compiler pass, the transformed source code is linked with the Immortal Threads library. Figure 3 presents the output of the compiler frontend for a multithreaded event-handling example.

The compiler frontend instruments all programmer-defined functions, including thread entry points, to create immortal

functions. More specifically, an immortal thread is a concurrency unit whose entry point is an immortal function.

Instrumentation of an Immortal Function. The compiler frontend instruments all local variables by using `_def` followed by the data type and name (i.e., the ordinary way of variable declaration in C language). This operation converts programmer-defined local variables to persistent static variables with local scope. Compiler frontend instruments variable manipulations using `_WR` and `_WR_SELF` interfaces to ensure memory consistency. These interfaces manage WAR dependencies, perform checkpoints, and keep functions idempotent. `_WR` manipulates variables when the update operation does not include any WAR dependency. `_WR_SELF` manipulates variables when there is a WAR dependency during the update operation. For example, the Immortal Threads library implements the assignment `{x=0}` using `_WR(x,0)` since there is no WAR dependency during this update. For `{x=x+5}`, the necessary operation becomes `_WR_SELF(uint32_t,x,x+5)` since the variable `x` is read first and then written. Immortal Threads provides different interfaces for variable manipulations with WAR dependencies to implement *just-in-time privatization*, which we will explain in Section 4. Additionally, calls to other immortal functions in an immortal function body are instrumented with the `_call` interface, which makes setting micro-continuations inside called immortal functions possible. Finally, the compiler frontend also instruments the function body by wrapping it using `_begin/_end` block. When a thread starts running for the first time, the first instruction in its entry immortal function is executed. If a power failure interrupts thread execution, the thread continues from the last checkpoint performed by the underlying Immortal Threads runtime, which can also be deep down in the call stack.

Thread Preemption. Unlike common preemptive models, where the continuation is saved on preemption, Immortal Threads saves the continuation (i.e., checkpoint) on each memory update. This guarantees the idempotence of the execution until the next checkpoint. Therefore, the scheduler can simply interrupt the execution of a thread and switch to the other one.

4 Implementation of Immortal Threads

We implemented Immortal Threads library mainly using standard C macros and preprocessor directives. The library also includes functions for system initialization and scheduling operations. We implemented the source-to-source transformation using the LLVM & Clang LibTooling library [1].

Target Hardware. The current implementation of Immortal Threads library targets MSP430FR5994 [48] microcontroller from Texas Instruments that is equipped with 256KB FRAM and 8KB SRAM memory. Immortal Threads library uses a persistent time circuitry (which keeps track of time across power failures [14, 15]) to handle events and data expiration. It is worth mentioning that a persistent timekeeper is not a mandatory requirement for Immortal Threads runtime since it will work even without checking event time constraints. On the other hand, de facto intermittent computing platforms, e.g., Flicker [24], already include a persistent timekeeper circuit.

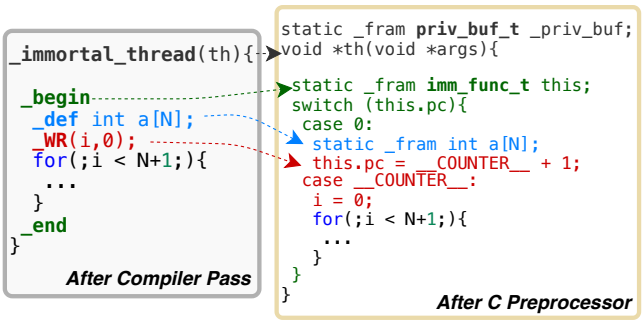


Figure 4: The structure of a source file after C preprocessor.

4.1 Immortal Function and Threads

Each immortal function (which can also be the entry point of a thread) maintains `imm_fn_t` structure that comprises a program counter (`pc`) to enable *micro continuations*, and a pointer to the same structure (`callee`) for calling other functions (via `_call`). For the *just-in-time privatization* operations, a privatization buffer (represented by `priv_buf_t`) is also maintained. All local variables are allocated in non-volatile memory as variables with static storage duration, which makes immortal threads (based on immortal functions) stackless. Figure 4 presents a sample output of the C preprocessor, which depicts how the privatization buffer (`__priv_buf`) and the function structure (`this`) are allocated in non-volatile memory.

4.2 Enabling Micro Continuations

Threads can be interrupted at any time by power failures, and their execution continues from the latest checkpoints. Immortal Threads library performs an almost-free checkpoint at each memory update via the `_WR` and `_WR_SELF` interfaces. Memory updates that lead to WAR dependencies (i.e., `_WR_SELF`) require *just-in-time privatization* to keep memory consistent.

Almost Free Checkpoints. Figure 4 presents how `_begin/_end` blocks (which are just C macros) are transformed into `_switch/_case` structures in C. Since the `imm_fn_t` structure for each immortal function is statically allocated, the `pc` field of is initialized with zero. Therefore, a function initially starts by executing its first case block `case 0:`. The almost-free checkpoint is just adding a new case statement at compile-time and modifying `pc` of the function. We implemented almost-free checkpoints using the standard GNU C preprocessor macro `__COUNTER__`, whose value increments each time the preprocessor encounters it. We implemented the core checkpoint code as follows:

```
#define _CP() \
    this.pc = __COUNTER__ + 1; case __COUNTER__:
```

If the thread (in which the function is running) restarts, the execution will continue from the `case` statement of the last checkpoint. However, only checkpoints are not sufficient to keep memory consistent. As indicated previously, `_WR` performs single-memory updates that do not lead to WAR dependencies. However, a sequence of operations `_WR(x,y);` `_WR(y,z);` form a WAR dependency. Therefore, we need to take a checkpoint before each memory update operation using `_WR`. We implemented this macro as follows:

```
#define _WR(arg,val) _CP(); arg=val;
```

Just-in-Time Privatization. Single memory updates that include WAR dependencies, e.g., `x++`, require a *two-phase commit* operation to keep memory consistent. In the first phase, Immortal Threads library creates a *private* version of the variable in the privatization buffer (`__priv_buf`) and updates the private version. Then a checkpoint is taken. In the second phase, Immortal Threads library commits the private version to the original variable. We present the implementation of the `_WR_SELF` macro that captures these steps below:

```
#define _WR_SELF(type,arg,val) \
    _CP(); *((type*)&__priv_buf.buffer)=val;\
    _CP(); arg=*((type*)&__priv_buf.buffer);
```

Thanks to just-in-time privatization, Immortal Threads does not require a compiler analysis to detect idempotent code blocks, as in Woude et al. [53]. Furthermore, there is no need for *static versioning*, as in Colin et al. [10]. Immortal Threads library forms a continuous sequence of idempotent code blocks by connecting them using almost-free checkpoints on the fly.

Calling other functions. When there is a power failure while a callee executes, the control should resume from the last memory update in the callee body. To call an immortal function, Immortal Threads library first checkpoints, saves the pointer to the callee in the caller's `_imm_fn_t` structure, and then makes the call as shown in the following pseudo-code:

```
#define _call(name,args) \
  _CP(); this.callee=name(args); \
  _CP(); this.callee->pc = 0; _CP();
```

If the callee successfully returns, the caller sets the program counter (`pc`) of the callee to zero and checkpoints. Consequently, the function will be able to be called again. If there is a power failure before the callee returns, the thread execution will restart from the entry immortal function, which will perform a set of nested function calls to reach the callee that has not finished yet. Therefore, the execution resumes from the last memory update in the leaf callee's body.

```
__asm__ volatile (
 "MOVA SP, %0" /* save SP on __sp variable */
 : "=m" (__sp) /* output */);

while (1) {
 functions[__th](0); /* call thread */
 __asm__ volatile (
  "ISR_return: \n" /* _schedISR will jump here */
  "MOVA %1, SP \n" /* restore old stack */
  "INC.B %0 \n"    /* __th++ */
  "CMP.B %2,%3 \n" /* if (__th == size) */
  "JNZ cont \n"
  "CLR.B %2 \n"    /* __th = 0 */
  "cont: \n"
  : "=m" (__th)    /* output */
  : "m" (__sp), "m" (__th), "m" (size)/* input */);
}

void __interrupt(TIMER0_A0_VECTOR) _schedISR(void){
 /* write return address ISR_return */
 __asm__ volatile ("MOV.W #ISR_return, 2(SP)");
}
```

Figure 5: The Round-Robin scheduler, which is the only platform-specific code in Immortal Threads library.

4.3 Thread Scheduling Implementation

The Immortal Threads scheduler includes platform-specific assembly code that switches the execution from the current thread to the next one. Figure 5 presents a part of our scheduler implementation. When the system restarts, the value of the stack pointer is saved in the variable `__sp`. The array `functions` contains pointers to the thread entries that are ready to run. The while loop indexes the threads with the persistent variable `__th` and calls them in order.

When the periodic timer of the scheduler fires, it interrupts the current thread, and the execution jumps to the `_schedISR` ISR routine. This ISR modifies the stack to replace the interrupt return address with the address of the label `ISR_return` in the scheduler loop. Upon interrupt return instruction (ISR routines execute `iret` upon return), the execution jumps to `ISR_return` label. At this point, the stack pointer is restored (using the old stack pointer in `__sp`) to continue the execution of the scheduler loop using its stack frame. Then, the value of the index `__th` is incremented, and the corresponding thread function is called. It is worth mentioning that thanks to micro continuation, the interrupted thread will not be in an inconsistent state. When the scheduler loop starts running again, it will continue execution from its latest checkpoint. Indeed, the ISR interruption acts as an artificial power failure.

Semaphores, Events and Data Races. Power failures might break the atomicity of operations on semaphores and events (e.g., post and wait) and might lead to data races. For example, if a thread modifies the semaphore but does not checkpoint due to a power failure, it will post the semaphore again after recovery. This situation leads to incrementing the semaphore twice. To prevent such issues, Immortal Threads library im-

plements event and semaphore operations by employing two-phase commit and double buffering, which are the main techniques proposed in the prior art to keep memory consistent despite power failures [31,34,54]. These operations firstly update the temporary values dedicated to events and semaphores. Then, the temporary values are atomically committed to their original locations. Upon system reboot, the scheduler checks if there is an uncommitted semaphore or event operation. If this is the case, it commits this operation. Then, it enables the interrupts and starts executing the threads. Immortal Threads library manages the data races between ISRs and threads by employing the same approach. Each event has a double-buffered event data buffer. An ISR does not modify the original buffer and immediately overwrites the event data. It uses the temporary buffer and then atomically commits it using a two-phase commit operation. These operations prevent the data races and inconsistency issues.

4.4 Compiler Frontend Implementation

We implemented Immortal Threads compiler frontend using the LLVM & Clang LibTooling framework. The AST produced by Clang is generally immutable, and source code rewriting cannot be directly reflected on the AST and its associated metadata. Therefore, it is necessary to keep track and manage the position offsets introduced by the transformations and solve conflicts when these transformations overlap. This limitation of Clang libraries, combined with the relative complexity of the entire source transformation for Immortal Threads, led us to adopt a multi-pass architecture inspired by the LLVM IR Pass framework. Each pass matches some parts of the AST and performs the appropriate source code rewriting. The rewritten source code is used to generate a new AST, on which the next pass operates.

Syntax Decomposition. One of the main challenges for source-to-source transformation is the `switch` constructs used in Immortal Threads lightweight checkpoints, which allow checkpoints only with a statement granularity. However, C programs might include expressions with WAR dependencies inside them, so it must be possible to perform checkpoints inside expressions. To this end, we decomposed these syntax constructs into separate statements so that it is possible to perform Immortal Threads lightweight checkpoints. In doing so, we paid special attention to aspects such as operator precedence and short-circuit evaluation.

Pessimistic Privatization due to Aliasing. The `_WR_SELF` interface, which performs JIT privatization, must be used for assignments where the left-hand side operand aliases with the right-hand side operand. In general, it is not possible to deduce all such aliases at compile time, e.g., when pointers are involved. Our current implementation pessimistically uses `_WR_SELF` instead of `_WR` when at least one operand contains a pointer dereference. We left integrating more advanced aliasing analysis as future work.

Shim API replacement. While a significant portion of the Immortal Threads operations is hidden from the programmer by the compiler frontend, primitives such as semaphores, mutexes, etc. (see Table 2) are visible to the programmer. These primitives have C macro-based implementations that generate `_case` statements for `_switch` blocks, which are inserted by the compiler frontend via `_begin` and `_end` statements later. Therefore, Clang fails to generate the initial AST that sets off the transformation pipeline. We solved this issue by providing these primitives as shim functions, and the compiler frontend replaces them with their actual macro-based implementations.

Pass Grouping Optimization. While the compilation time of the C language by modern compilers such as Clang is generally fast enough, having to re-parse the translation unit after each transformation is still a noticeable overhead when long source files are involved. Some of the presented passes depend on others. For example, instrumenting assignments with `_WR` and `_WR_SELF` is easily performed once syntax decomposition is done. On the other hand, some passes operate on orthogonal elements of the AST, for which we don't need to worry about source rewriting conflicts. We grouped these passes and executed them using the same AST.

Compiler Directives and Code Optimization. In exceptional cases, the programmer can modify the behavior of the Immortal Threads compiler using custom attribute directives (`__attribute__`). For example, we allow the programmer to mark idempotent functions so that they are not instrumented for the sake of some manual optimizations. This feature reduces the overhead of frequent checkpoints but creates a risk of wasted work. In Section 6 we discuss ways to improve this aspect. Furthermore, we also implemented a specific compiler optimization to coalesce successive `WR` and `WR_SELF` macros in the code to eliminate frequent checkpoints that might degrade the execution time of time-sensitive computational loads. The programmer can enable this optimization by passing a flag to the compiler frontend. In this case, the compiler puts the best effort to reduce the number of checkpoints in basic blocks. In summary, Immortal Threads compiler frontend enables the developers to select the trade-off between the checkpointing overhead and wasted work based on the specific requirements of their applications.

Switch Statements. We allow programmers to use a subset of the `switch` statement in their code (unlike Protothreads, which does not permit programmers to use switch statements). Specifically, we support switch statements in which all the statements associated with case labels either finish with a `break` statement or are empty, i.e., the case directly falls through to the next case. Given the constraint we put, it is straightforward to transform such use of the switch into an equivalent if/else based code. The compiler frontend terminates with an error message if it encounters an unsupported usage of the switch statement.

Function Reuse. Immortal Threads needs to create different instances of thread-shared immortal functions to prevent data

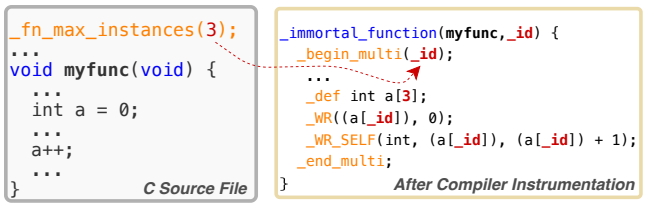


Figure 6: The compiler instrumented version of a sample function `myfunc` that is shared among multiple immortal threads.

races and memory inconsistencies. We implemented function reuse through a combination of compiler and Immortal Threads library support. Figure 6 presents a sample function that is shared among several immortal threads, and its instrumented version. The programmer uses a compiler directive (`_fn_max_instances` as indicated in the figure) to declare the maximum number of concurrent callers for the shared functions in the application. If the number of instances is not provided, the compiler can also use a default number to avoid programmer intervention. Our compiler modifies the signature of each shared immortal function by prepending an id parameter. Moreover, the compiler also transforms all local variables into arrays whose lengths correspond to the number of instances. Thus, each access to any local variables becomes an array access, where the index is the id. Alternatively, to avoid the overhead of accessing the array, the Immortal Threads compiler can also create copies of the same immortal function at the source code level. Therefore, it can replace the original immortal function's body with a call table that calls the appropriate function copy depending on the id parameter. This support lets the developer trade executable size for runtime efficiency. Besides, for each shared immortal function, the compiler allocates an associated metadata data structure containing a bitmap to represent unused instances, where each bit that is one represents a free instance. We present the pseudo-code of the macro that is used for calling shared functions as follows:

```
#define _call_multi(name,args) \
  _CP(); get_instance_id(&this->callee_id); \
   this.callee=name(this->callee_id, args); \
  _CP(); release_instance_id(&this->callee_id); \
   this.callee->pc = 0; _CP();
```

The caller of an immortal function must first get a free instance, that is, access the bitmap and clear a bit that is set (using `get_instance_id`). It is worth mentioning that not getting a free instance should not happen by design. The programmer must ensure to provide the correct `_fn_max_instances` number configuration. If no free instance is available at runtime, it's an assertion failure. Once the immortal function returns, the caller must release the called immortal function instance by setting the previously cleared bit (using `release_instance_id`). As a side note, recursive functions are not supported in the current implementation of Immortal Threads. We argue that this is not a significant limitation, as recursion is generally avoided in embedded systems.

5 Evaluation

We proceed with the evaluation of Immortal Threads by presenting a performance comparison against three state-of-the-art runtimes Alpaca [34], InK [54], and TICS [31].

Benchmarks. We selected Bitcount (BC), Cuckoo Filter (CF), and Activity Recognition (AR) as the main benchmarks since they are widely used in previous works [31, 34, 54]. We also considered the DNN inference presented in Gobieski et al. [22] as a benchmark since the inference operations are computationally intense (e.g., the first convolution layer requires 150080 multiplications) and access non-volatile memory excessively (FRAM is more expensive compared to SRAM access). We used the BC, CF, and AR implementations in publicly available code repositories of Alpaca, InK, and TICS during our evaluations. We also considered the publicly available plain C versions of these benchmarks, which we call *Plain-Ram*, where all variables are in SRAM (no FRAM access). Therefore, they do not have overheads regarding non-volatile memory access, checkpoints, privatization, etc. Moreover, we created the *Plain-Fram* versions of these benchmarks where all variables are maintained in FRAM. Therefore, they have an additional FRAM access overhead compared to Plain-Ram versions. Note that the Plain-Ram and Plain-Fram implementations do not guarantee forward progress and memory consistency.

Compiler Directives. In the task-based implementations of the benchmarks (in Alpaca and InK), we observed that some functions are not declared as tasks to reduce task transition overheads and employ a *manual* compile-time optimization for these task-based systems. These functions are idempotent since they do not modify their inputs or global variables. Moreover, they are mostly small in size and frequently called at runtime. Similarly, in Immortal Threads implementations of these benchmarks, we annotated these idempotent functions (using the compiler directives mentioned in Section 4.4) to bypass unnecessary compiler instrumentation and reduce the number of checkpoints and the execution time overhead of Immortal Threads. Using annotations, we marked eight functions in BC, six functions in CF, three functions in AR, and seven functions in DNN, respectively. Furthermore, for the DNN benchmark, we enabled the checkpoint coalescing feature of our compiler frontend to reduce the memory access overhead of the data-intensive computations. These features of Immortal Threads compiler frontend allowed us compile-time optimizations *without programmer involvement* (excluding the annotation of idempotent functions).

Target Platform and Tools. We used the MSP-EXPFR5994 evaluation board [48], which includes 256kB FRAM and 4kB SRAM memory and can operate at up to 16MHz. We

	Bitcount (BC)		Cuckoo (CF)		Activity (AR)		DNN	
	Time (ms)	*Energy (µj)*	*Time (ms)*	*Energy (µj)*	*Time (ms)*	*Energy (µj)*	*Time (ms)*	*Energy (µj)*
Plain-Ram	24.73	41.54	36.81	63.30	822.78	1415.53	✗	✗
Plain-Fram	213.23	344.57	48.03	87.71	1053.75	2073.65	33624.60	59710
Alpaca	285.29	690.46	79.25	210.25	1897.90	5175.50	41787.88	77537
InK	497.19	1287.05	376.12	1016.49	3100.97	8707.40	46994.33	91961
TICS	482.38	1205.20	1229.30	2025.70	2667.16	7106.80	✗	✗
Immortal Threads (IT)	274.43	456.31	53.91	108.15	2503.45	4917.10	69215.54	147595

Table 4: Execution time and energy consumption of the benchmarks on **continuous power**.

	Immortal Threads			Alpaca			InK			TICS		
	Avg. Task Size (µs)	*Avg. Checkpoint Overhead (µs)*	*Tot. Invoked Checkpoint*	*Avg. Task Size (µs)*	*Avg. Task Trans. Overhead (µs)*	*Tot. Task Transition*	*Avg. Task Size (µs)*	*Avg. Task Trans. Overhead (µs)*	*Tot. Task Transition*	*Avg. Task Size (µs)*	*Avg. Checkpoint Overhead (µs)*	*Tot. Invoked Checkpoint*
BC	∼50.33	∼14.44	4236	280.97	127.56	709	169.69	537.78	709	2028.38	475.31	709
CF	∼31.97	∼9.52	1299	61.92	121.08	451	129.23	776.50	419	1110.58	454.72	518
AR	∼23.22	∼17.00	30223	829.87	124.55	2001	880.36	670.06	2007	245.14	411.14	1130
DNN	∼20.57	∼14.45	1865103	16742.40	570.05	2412	31765.28	1130.26	1486	✗	✗	✗

Table 5: Average execution time of a task, and task transition/checkpoint overhead.

used the 1 MHz frequency during the experiments on performance comparison (to be compatible with existing studies [31, 34, 54]. We used the GNU GCC v9.2.0.50 to compile our applications. To measure the time overhead and energy consumption, we used a logic analyzer and TI EnergyTrace software [49], respectively. We used the Powercast TX91501-3W [12] RF transmitter operating at 915 MHz center frequency to power wirelessly our evaluation board connected to the P2110-EVB [13] RF receiver. We used the 1mF and 50mF capacitors on P2110-EVB as energy storage to observe different power failure patterns. We also emulated power failures for the repeatability and replicability of comparative measurements. We generated a random soft reset triggered by an MCU timer with a uniformly distributed firing period in the interval of [5ms, 20ms] (as in Yildirim et al. [54]).

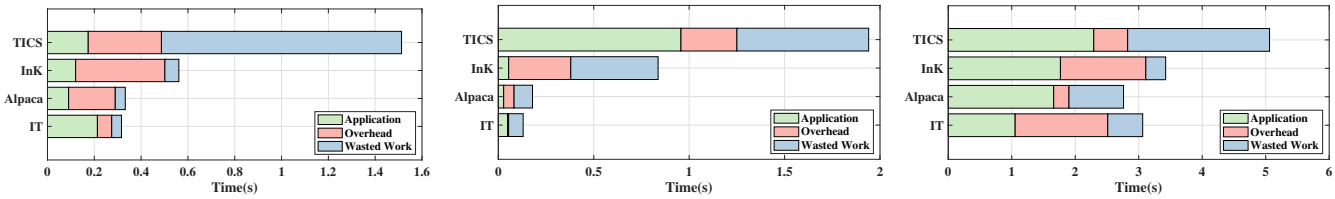
Evaluation Metrics. We considered *execution time* and *energy consumption* as the main metrics to evaluate the benchmarks. We also measured *wasted work* (which denotes computational progress lost due to power failures), *runtime overhead* introduced to progress the computation and keep memory consistent, and the memory requirements and code sizes of the benchmark implementations.

5.1 Evaluation Using BC, CF and AR

The InK and Alpaca implementations of the benchmarks have identical task boundaries. We placed TICS checkpoints aligned with task boundaries in the InK and Alpaca implementations for the sake of a fair comparison.

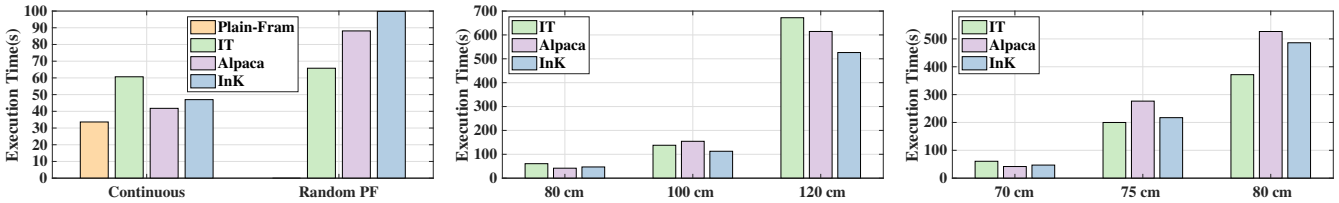
Continuous Power. Table 4 presents the continuously-powered execution time and energy consumption of the benchmarks. These benchmarks have different characteristics; for example, BC accesses memory more frequently to manipulate variables, and CF is more computationally dense. The differences in the time and energy overheads of the plain-Ram and plain-Fram versions show that intermittent computing, which requires frequent FRAM access, comes with significant overheads. Immortal Threads, InK, Alpaca, and TICS introduce additional overhead to Plain-Fram versions of the benchmarks to ensure forward progress and memory consistent. We observed that the performance of Immortal Threads is quite comparable to that of InK, TICS, and Alpaca during the continuous execution of the benchmarks. The reason is that InK and Alpaca need to perform bulk copy operations to commit the temporary buffers atomically during task transitions. Similarly, TICS needs to copy the stack and registers upon each checkpoint. Even though Immortal Threads maintains all variables in FRAM (which increases the time and energy overhead), almost-free checkpoints reduce the checkpointing cost, and just-in-time privatization eliminates block FRAM copy operations. Table 5 summarizes the average execution time of a task and the overhead of task transitions and checkpoints.

Intermittent Power. Figure 7 presents the wasted computational progress due to power failures and runtime overheads during intermittent execution with randomly generated power failures. The runtime overhead in InK and Alpaca is mainly due to the undo and redo logging operations performed by the tasks to recover computation upon power failures. Alpaca has a lower task transition overhead since it only double-buffers the task-shared variables with WAR dependencies and commits them upon task completion. Similarly, the overhead of TICS is due to the checkpoints and their restoration. TICS has more commit overhead since it checkpoints at the end of each task boundary, which requires a large bulk memory copy operation compared to task transitions in InK and Alpaca (see Table 5). During our experiments, Alpaca implementations of the benchmarks led to shorter task execution times and reduced wasted work since Alpaca introduced a lower runtime overhead compared to InK and TICS. In Immortal Threads, the runtime overhead is the total overhead of almost-free checkpoints, just-in-time privatization, and restoring compu-



(a) Bitcount (b) Cuckoo Filter (c) Activity Recognition

Figure 7: Total execution time, runtime overhead and wasted work with **controlled power failures**.



(a) Cont. power and controlled power failures. (b) RF-powered, **1mF** capacitor. (c) RF-powered, **50mF** capacitor.

Figure 8: DNN benchmark with **continuous power (Cont.)**, **controlled power failures** and **RF power** (at different distances).

tation. Conceptually, idempotent code blocks between two successive memory updates in Immortal Threads can be considered as a tiny task. These micro-continuations reduced wasted work and the total execution time significantly compared to existing runtimes.

5.2 Evaluation Using DNN Inference

Immortal Threads showed promising performance with relatively small benchmarks. To evaluate it under excessive memory access and computational load, we used the deep neural network (DNN) inference presented in [22]. This DNN model requires approximately 180kB FRAM to maintain the DNN weights and input matrix. The Alpaca DNN implementation in [22] employs *loop continuation* and has 18 tasks (2 tasks are for specific initialization operations). It is again worth mentioning that loop continuation relies on manually eliminated WAR dependencies and **violates** the task-based model. The implementation of TICS (from its public repository) could not support DNN inference, since its checkpoints lead to memory inconsistencies when the application accesses the higher regions of FRAM.InK requires DNN weights and the input matrix to be allocated in task-shared memory regions. However, InK double buffers all task-shared variables and commits them non-selectively at each task completion. Therefore, the implementation of DNN in InK is not feasible since it needs to commit a large amount of task-shared data at each task transition. However, by **violating** the InK model, we provided loop continuation support, allowed tasks to manipulate FRAM directly, and managed to implement DNN, which has 16 tasks (2 tasks specific to Alpaca are not required). Since our platform has only 4kB SRAM, we could not implement the Plain-RAM version of DNN.

Continuous Power. Due to the increased number of memory write operations, the overhead of Immortal Threads is more visible in this case (see Figure 8a). Immortal Threads introduced almost twice more overhead compared to Plain-Fram DNN (see Table 4). The main reason for performance degradation is committing each memory update atomically via the JIT privatization. INK and Alpaca performed better since they eliminated memory commit overheads by **violating** the task-based model via loop continuation. This violation allowed for larger tasks, which reduced the number of task transitions.

RF Powered. We used 1mF and 50mF capacitors as energy storage and three different distances from the RF power transmitter to observe different power failure patterns. The charging time of the capacitor increases with the distance between the receiver and the power transmitter. The charging time of the 50mF capacitor is longer than that of the 1mF capacitor. On the other hand, the 50mF capacitor provides a longer operation time. We observed significantly higher power failure rates with the 1mF capacitor. We conclude from Figure 8b and Figure 8c that Immortal Threads's performance becomes superior to the other runtimes as the power failure rate increases—it wastes less computational progress thanks to the micro-continuations.

Unviolated Task Model. We implemented DNN in Alpaca **without** the loop continuation to answer the question of what the DNN performance is without violating the task-based model. Our implementation, which we call the original Alpaca (*Alpaca (Org.)*), introduced an additional 11 tasks to the DNN implementation with loop continuation. Figure 9 presents execution time, runtime overhead, and wasted work during controlled power failures. We observed that Immortal Threads outperformed the original Alpaca significantly, i.e., led to a twice shorter execution time. Furthermore, even though

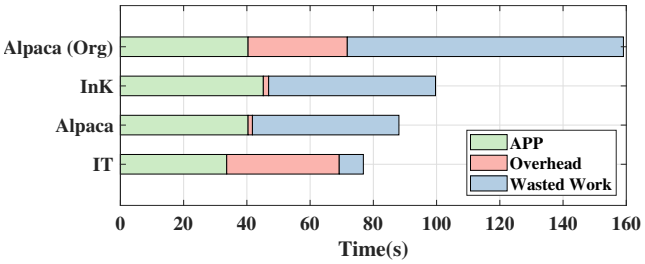


Figure 9: DNN execution time, runtime overhead and wasted work with **controlled power failures**.

Immortal Threads has more overhead compared to the InK and Alpaca implementations, it wasted significantly less work.

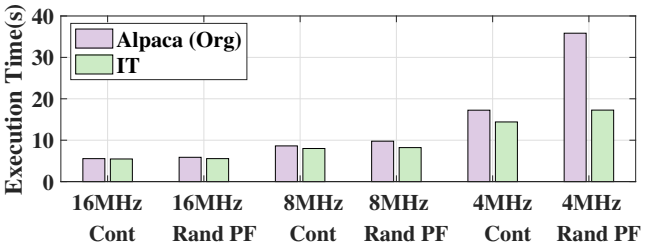


Figure 10: Performance of DNN at different frequencies.

Fram-CPU Bottleneck. As CPU speed increases, the FRAM access latency becomes more dominant in system performance. The FRAM in our platform can operate at a maximum speed of 8 MHz. We evaluated the original Alpaca and Immortal Threads DNN inference performances using clock frequencies of 4MHz, 8MHz and 16MHz with controlled power failures, as presented in Figure 10. We observed that as the clock frequency increases, the performances of both systems come closer to each other due to the latency of FRAM access, but Immortal Threads still performs better.

	Alpaca			InK			TICS		Immortal Th.	
	Tasks	*Trans.*	*Lines*	*Tasks*	*Trans.*	*Lines*	*Chkpts.*	*Lines*	*Chkpts.*	*Lines*
BC	11	24	251	10	26	326	10	238	82	188
CF	16	23	279	15	27	326	14	353	68	149
AR	12	20	330	11	20	449	8	411	123	309
DNN	18	48	2412	16	39	2214	✗	✗	276	1486

Table 6: Num. lines of code, num. of tasks and transitions (Alpaca and InK), num. of checkpoints (TICS and InK).

5.3 Cognitive Load, Code Size, and Memory Requirements

We define the *cognitive burden* of intermittent computing as the effort put to split code into idempotent sections, i.e., implementing tasks and task-based control flow in task-based systems and inserting checkpoints for checkpointing systems.

	Alpaca			InK			TICS			Immortal Th.		
	.text	*Ram*	*Fram*	*.text*	*Ram*	*Fram*	*.text*	*Ram*	*Fram*	*.text*	*Ram*	*Fram*
BC	2254	2	856	3356	0	4712	7160	4446	5572	10175	345	478
CF	3148	348	1070	4242	318	3000	11160	4655	6322	9831	370	498
AR	2258	0	784	3576	0	4474	11416	759	5430	11885	346	542
DNN	13898	224	192K	843	0	168K	✗	✗	✗	19394	356	149.5K

Table 7: Memory and Code Size requirements (in B).

We used the number of tasks (and checkpoints) and task-based control-flow declarations (in addition to the number of lines of code) shown in Table 6 as a metric to measure the burden. Thanks to the Immortal Threads compiler frontend, programmers write their programs without focusing on the details of the intermittent execution. The compiler frontend automatically wraps variable manipulations using the macros shown in 3, inserts checkpoints and creates idempotent code sections on the fly. Programmers use only the interfaces in Table 2, which are almost *identical* to the interfaces found in continuously powered event-driven systems [17, 32]. It is worth mentioning that Table 6 presents the number of lines after the Immortal Threads compiler pass (which has additional code inserted by the compiler). Even in this case, the number of lines in the implementations with Immortal Threads is almost half of that in the implementations with task-based models. As shown in Table 7, the code size of the application implemented in Immortal Threads library is larger than others. The main reason is that Immortal Threads library is implemented using C macros. It is worth mentioning that just-in-time privatization eliminates data versioning, reflected as considerably reduced data section requirements.

5.4 Summary of Evaluation Results

Our results showed that Immortal Threads has comparable runtime overhead to the existing runtimes. The runtime overhead and benefits of Immortal Threads depend on the application's memory access patterns and the frequency of the power failures. Compared to Immortal Threads, InK (violated task-based model) had approximately ten times more wasted computation and 1.5 times more execution time DNN inference under power failures. Besides, the original Alpaca (the unviolated task-based model) had approximately 17 times more wasted computational progress and 2.4 times more execution time. Therefore, we observed that during frequent power failures Immortal Threads reduced execution time and wasted work by up to 40% and 90%, respectively. We conclude that Immortal Threads brings pseudo-stackful multithreaded programming with acceptable overhead and no cognitive burden.

5.5 Greenhouse Monitoring Application

We proceed with greenhouse monitoring, which is a common application shown in intermittent computing studies (e.g., [31]), to demonstrate a time-constrained event-driven

scenario. To this end, we used a temperature sensor on the MSP430FR5994 MCU, a solar panel for energy harvesting, and an eZ430-RF2500 [47] board equipped with a CC2500 transceiver to transmit and receive data. MCU used a UART connection to send commands to eZ430-RF2500 for data transmission. We used the DS1302 [39] Real-Time Clock for time tracking despite power failures. As energy storage, we used the 50 mF supercapacitor of the P2110-EVB since it has a voltage regulator.Figure 11 shows our experimental setup.

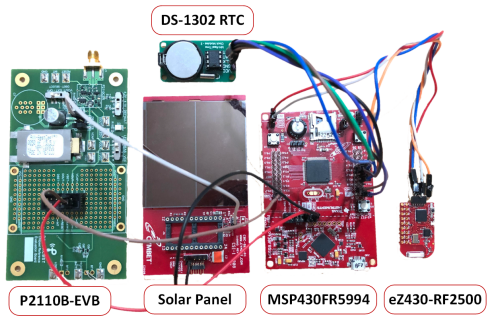


Figure 11: Greenhouse monitoring experimental setup.

GHM Implementation. A timer thread checks the RTC to signal timer events every 6 seconds. The sense thread blocks on the timer event to sense the temperature and store it in a buffer with a timestamp. When the number of samples reaches 10, the sense thread calculates the average and signals the send event. The send thread unblocks, checks the event timestamp (via `_EVENT_WAIT_EXP`), and sends data to eZ430-RF2500 if the event has not expired. Otherwise, it ignores the event.

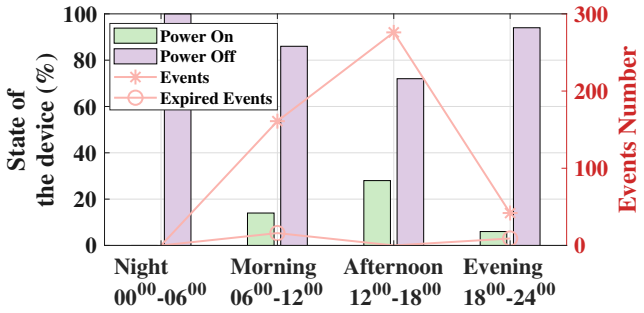


Figure 12: The number of expired events and the on/off time percentage of the device during different parts of the day.

Results. We placed our setup in an outdoor location on our campus for 24 hours. Figure 12 shows the results according to the parts of the day. To expose the effects of energy availability, we split the results into 6-hour timeframes. Since there was not enough energy at night, no events occurred. Due to power failures and charging times, 16 out of 177 events expired in the morning. The available environmental energy was high during the afternoon, and none of the 276 events expired since there were no power failures. Similarly, 9 out of 42 events have expired in the evening. We conclude that Immortal Threads successfully caught these expirations and stopped data processing to save precious harvested energy.

6 Discussion and Future Work

Programming models. Protothreads [18] is an abstraction designed for continuously-powered sensors. Its local continuation concept enables blocking threads, but such continuations can be saved only in the thread's entry function. InK task-threads provide a solution for intermittent event-driven applications, but they have mentioned drawbacks in this paper, e.g., the cognitive load of the task-based systems. Immortal Threads is an abstraction that provides micro continuation in intermittently powered systems, which is as lightweight as Protothreads's local continuation. In addition, they can be saved anywhere in the call stack of a thread, not only in the entry function. However, the current implementation of micro continuations achieves pseudo-stackfulness by employing `switch`-based constructs. An essential question for our future work is whether it is possible to control the compiler's usage of registers so that continuations can be composed of only the program counter and stack pointer while maintaining the remaining state in the (non-volatile) memory. Checkpointing at the statement boundary, combined with declaring variables as `volatile` or compiler fences, may provide a possible direction.

Peripheral operations support As previous works (e.g., [35]) point out, peripheral interaction should be atomic,which means no power failure can be allowed in between. In order to enable atomic execution of I/O handling operations, Immortal Threads compiler frontend can be extended to support a new compiler directive to mark atomic I/O functions, i.e., functions that should not contain checkpoints. The compiler frontend can add the necessary code that performs privatization of the parameters passed by address to such functions.

Checkpoint Optimization. Immortal Threads performs frequent checkpoints. Compiler analysis can be performed to merge and reduce unnecessary checkpoints. However, this may lead to more wasted work. In general, there is always a trade-off between checkpoint frequency (and the associated overhead) and the amount of waste work. Maeng et al. [36] proposes an adaptive approach: checkpoints are disabled at runtime when the system has still enough energy. However, this approach is effective only when *determining whether to checkpoint* has much less cost than *taking the checkpoint*, which does not apply to Immortal Threads, where checkpointing is merely an atomic write. One possible way to introduce adaptive checkpointing is to have multiple versions for each immortal function with a different checkpoint density. The runtime can then determine which version to call based on the energy availability. We leave this issue for future work.

7 Conclusions

Immortal Threads is the first intermittent computing runtime that enables pseudo-stackful multithreaded programming. Using Immortal Threads, programmers focus only on their multithreaded program logic that handles events instead of focusing on managing intermittent execution. Immortal Threads brings the missing event-driven primitives to intermittent computing, e.g., semaphores and event expiration handling. All these features come with an overhead comparable to the overhead of existing intermittent computing runtimes. We observed that, depending on the application and power failure frequency, Immortal Threads can even reduce execution time and wasted work by up to 40% and 90%, respectively.

Acknowledgments

We thank the anonymous reviewers of OSDI 2021, SOSP 2021, ASPLOS 2021 and OSDI 2022 for their valuable comments and feedback. We would like to thank Przemysław Pawełczak (TU Delft, The Netherlands) for encouraging us to send this work to OSDI 2022. We are also grateful to Rodrigo Bruno for shepherding our final draft.

Availability

We release Immortal Threads as an open source project for the community, whose artifacts can be downloaded from https://tinysystems.github.io/ImmortalThreads.

References

[1] Clang 7 libtooling. https://github.com/llvm-mirror/clang/blob/master/docs/LibTooling.rst, March 2019. Last accessed: May. 7, 2021.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[3] Saad Ahmed, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, Naveed Anwar Bhatti, and Luca Mottola. Towards smaller checkpoints for better intermittent computing. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 132–133. IEEE, 2018.

[4] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 70–81, 2019.

[5] Abu Bakar, Alexander G Ross, Kasim Sinan Yildirim, and Josiah Hester. Rehash: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–42, 2021.

[6] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016.

[7] Richard Barry. Freertos, a free open source rtos for small embedded real time systems. Available at: "https://www.freertos.org/", 2003.

[8] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 209–220. IEEE, 2017.

[9] Michael Buettner, Benjamin Greenstein, and David Wetherall. Dewdrop: An Energy-Aware runtime for computational RFID. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[10] 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*, pages 514–530, 2016.

[11] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 116–127, 2018.

[12] Powercast Corp. Powercast hardware. http://www.powercastco.com, 2014. Last accessed: Dec. 10, 2020.

[13] Powercast Corp. Powercast hardware. https://www.powercastco.com/wp-content/uploads/2016/11/p2110-evb1.pdf, 2015. Last accessed: Dec. 10, 2020.

[14] Eren Çürük, Kasim Sinan Yıldırım, Przemyslaw Pawelczak, and Josiah Hester. On the accuracy of network synchronization using persistent hourglass clocks. In

Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, pages 35–41, 2019.

[15] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–67, 2020.

[16] Jasper De Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):1–34, 2020.

[17] Adam Dunkels, Bjorn Gronvall, and Thiemo 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. IEEE, 2004.

[18] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, 2006.

[19] Caglar Durmaz, Kasim Sinan Yildirim, and Geylani Kardas. Puremem: a structured programming model for transiently powered computers. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1544–1551, 2019.

[20] Kai Geissdoerfer, Raja Jurdak, and Brano Kusy. Long-term energy-neutral operation of solar energy-harvesting sensor nodes under time-varying utility. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 156–157. IEEE, 2018.

[21] 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.

[22] 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*, pages 199–213, 2019.

[23] 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.

[24] 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*, pages 1–13, 2017.

[25] 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*, pages 1–13, 2017.

[26] Matthew Hicks. Clank: Architectural support for intermittent computation. *ACM SIGARCH Computer Architecture News*, 45(2):228–240, 2017.

[27] 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*, pages 193–204, 2019.

[28] 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. IEEE, 2014.

[29] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-E., Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2009.

[30] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Bfree: Enabling battery-free sensor prototyping with python. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4):1–39, 2020.

[31] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–99, 2020.

[32] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[33] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, 2015.

[34] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[35] 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, pages 1101–1116. Association for Computing Machinery.

[36] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[37] 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*, pages 1005–1021, 2020.

[38] 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 Transactions on Sensor Networks (TOSN)*, 16(1):1–24, 2020.

[39] Maxim Interated. Ds1302 trickle-charge timekeeping chip. https://datasheets.maximintegrated.com/en/ds/DS1302.pdf, 2019. Last accessed: September 2019.

[40] William P McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, 2006.

[41] William P. McCartney and Nigamanth Sridhar. Stackless Multi-Threading for Embedded Systems. *IEEE Transactions on Computers*, 64(10):2940–2952, October 2015. Conference Name: IEEE Transactions on Computers.

[42] 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*, pages 8–14, 2019.

[43] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–3, 2014.

[44] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 159–170, 2011.

[45] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1085–1100, 2019.

[46] 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.

[47] Texas Instruments. ez430-rf2500 development tool user's guide. https://www.ti.com/lit/ug/slau227f/slau227f.pdf, 2015. Last accessed: September 2015.

[48] Texas Instruments. Msp430fr58xx, msp430fr59xx, msp430fr68xx, and msp430fr69xx family user's guide. http://www.ti.com/lit/ug/slau367o/slau367o.pdf, 2019. Last accessed: September 2019.

[49] Texas Instruments. EnergyTrace Technology. https://www.ti.com/tool/energytrace, 2021.

[50] Texas Instruments, Inc. FRAM faqs. http://www.ti.com/lit/ml/slat151/slat151.pdf, 2014. Last accessed: 2018.

[51] 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*, pages 54–67, 2018.

[52] Harrison Williams, Michael Moukarzel, and Matthew Hicks. Failure sentinels: ubiquitous just-in-time intermittent computation via low-cost hardware support

for voltage monitoring. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 665–678. IEEE, 2021.

[53] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, Savannah, GA, November 2016. USENIX Association.

[54] 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*, pages 41–53, 2018.

[55] Eren Yildiz, Lijun Chen, and Kasim Sinan Yildirim. Immortal Threads GitHub Repository. https://tinysystems.github.io/ImmortalThreads/, 2022. Last accessed: June. 1, 2022.