Efficient and Safe I/O Operations for Intermittent Systems

Eren Yıldız
Ege University
Izmir, Turkey
eren.yildiz@ege.edu.tr

Saad Ahmed
Georgia Institute of Technology
Atlanta, USA
sahmed@gatech.edu

Bashima Islam
Worcester Polytechnic Institute
Worcester, USA
bislam@wpi.edu

Josiah Hester
Georgia Institute of Technology
Atlanta, USA
josiah@gatech.edu

Kasım Sinan Yıldırım
University of Trento
Trento, Italy
kasimsinan.yildirim@unitn.it

Abstract

Task-based intermittent software systems always re-execute peripheral input/output (I/O) operations upon power failures since tasks have all-or-nothing semantics. Re-executed I/O wastes significant time and energy and risks memory inconsistency. This paper presents EaseIO, a new task-based intermittent system that remedies these problems. EaseIO programming interface introduces re-execution semantics for I/O operations to facilitate safe and efficient I/O management for intermittent applications. EaseIO compiler front-end considers the programmer-annotated I/O re-execution semantics to preserve the task's energy efficiency and idempotency. EaseIO runtime introduces regional privatization to eliminate memory inconsistency caused by idempotence bugs. Our evaluation shows that EaseIO reduces the wasted useful I/O work by up to 3× and total execution time by up to 44% by avoiding 76% of the redundant I/O operations, as compared to the state-of-the-art approaches for intermittent computing. Moreover, for the first time, EaseIO ensures memory consistency during DMA-based I/O operations.

CCS Concepts: • **Computer systems organization → Embedded software**.

Keywords: Intermittent Computing, Energy Harvesting, Batteryless Internet of Things, Peripherals

ACM Reference Format:
Eren Yıldız, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasım Sinan Yıldırım. 2023. Efficient and Safe I/O Operations for Intermittent Systems . In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3587435

1 Introduction

With predictions of more than a trillion Internet-of-Things devices by 2035, concerns of maintenance costs and scale [4] have caused significant interest in energy harvesting for embedded sensing devices to extend lifetimes. These devices leave batteries behind, and instead power all operations from ambient energy such as solar, thermal gradient, or even microbes [11, 39]. Instead of batteries, they store harvested energy in a small capacitor to perform sensing, actuation, inference, computation, and communication [22, 53]. Batteryless devices compute *intermittently* since they suffer from frequent power failures due to the high spatio-temporal variability of ambient energy, as shown in Figure 1. Power interruptions can occur anywhere in the program, requiring these devices to save application state in durable non-volatile memory (e.g., FRAM, STT-RAM, or MRAM) so that it can restore it whenever energy is available again.

Prior art proposed system support to enable intermittent execution of programs by efficiently saving application state across reboots. Programmers can place checkpoints in the application code to save the entire volatile state (including registers and volatile main memory) across power failures [2, 7, 8, 10, 42], which might introduce considerable time and energy overhead. Another practice for programmers is to use a programming language-based model to divide an application into a set of atomic tasks with lightweight checkpoints at the end of each task boundary to ensure persistence [14, 26, 34, 54]. Together, these practices have enabled significant progress in intermittent computing: including a battery-free virtual machine used to play Nintendo Game Boy games [19], an online compiler for novice-focused block-based programming via MakeCode [31], and even batteryless devices shot in space [33] and used in medical implants [24].

Despite these commendable efforts on energy-efficient state retention, key limitations exist in adopting existing task-based systems for *peripheral bound* workloads, which prevents general applicability. We list the following problems regarding *always repeated* I/O operations.

P1: Wasteful Repetitive I/O Operations. By definition, tasks are atomic entities, and they have all-or-nothing semantics. If a power failure interrupts a task execution, the task is re-executed again in the next energy cycle. As a result, I/O operations inside tasks are always re-executed upon task restart after recovering from a power failure. However, it is not always necessary to repeat I/O operations. Some peripheral operations have *single-shot* semantics. For instance, a camera-based sensor does not need to re-capture the image if the capture operation was successful in the previous energy cycle. Besides, the output of some I/O operations can remain valid for a certain duration of *time*. For instance, a temperature sensor does not need to repeat its execution if the time between power failures is less than the freshness interval of the value. Unfortunately, existing task-based systems do not provide programming language and runtime support to capture re-execution semantics of I/O operations to avoid repeated I/O operations. Thus, they waste a significant amount of harvested energy [26, 30, 54, 55].

P2: Idempotence Bugs. Modern applications employ complex machine learning algorithms to enhance the inference capabilities of batteryless sensors [23, 27]. Such applications involve frequent and large data movements between volatile and nonvolatile memory, requiring peripheral operations, e.g., direct memory access (DMA) copy, that bypasses the CPU to perform a fast and efficient data transfer. Performing such operations within a task can lead to *memory inconsistencies* under intermittent energy supply if they modify nonvolatile memory directly [41]. The existing task-based programming model lacks appropriate interfaces to help programmers guard against idempotence bugs due to repeated I/O operations.

P3: Unsafe Program Execution. Embedded sensing devices interact with the environment via different peripherals, and programs may take different execution paths in the code based on the sensed input [46]. For instance, repeating an I/O operation might lead to incorrect behavior when a task's control flow depends on the result of that I/O operation [48]. While existing works rely on compile-time analysis to avoid such executions [48], no programming language support exists in task-based solutions to help programmers ensure the correct execution of programs.

Problem Statement. Existing task-based intermittent computing systems fail to address these problems. We need language semantics and runtime that complement the existing task-based programming model with the following features:

1. providing re-execution semantics for various peripheral operations to eliminate unnecessary repeated I/O operations and improve energy efficiency,
2. ensuring memory consistency, safe program execution during repeated I/O operations.

Challenges. Introducing these features to intermittent computing is not trivial. Depending on the application, each peripheral (and operation) requires different re-execution semantics. Some peripheral operations require re-execution after every reboot, while others only require re-execution if the previous peripheral operation expires. A group of I/O operations does not need repetition in the next energy cycle if they are fully executed once.

1.1 Our Contributions

This paper introduces EaseIO (**E**fficient **A**nd **S**af**E I/O**), a programming language and runtime for intermittent computing that eliminates the inefficiencies and memory inconsistencies due to repeated I/O operations. EaseIO allows programmers to annotate the re-execution semantics for their intermittent programs, as depicted in Figure 3. The EaseIO compiler front-end transforms the program source code by considering the programmer-annotated I/O re-execution semantics and preserving task idempotency. EaseIO runtime library handles the re-execution of I/O operations and prevents memory inconsistency due to idempotence bugs.

Briefly, we make the following contributions:

1. *Programming Language and Runtime.* We propose a low-overhead programming language model and a runtime system, named EaseIO, that enables programmers to introduce I/O re-execution semantics and avoid unnecessary peripheral operations after each reboot.
2. *Memory Consistency Management.* Our APIs and compiler annotations help programmers to annotate different code regions that can potentially cause memory inconsistencies across reboots.
3. *Regional Privatization.* DMA-based I/O bypasses the CPU and directly modifies non-volatile memory, which might lead to memory inconsistencies. We introduce a new method, named regional privatization, that allows the developers to guard against memory inconsistencies specific to DMA-based I/O.

We evaluate EaseIO against state-of-the-art solutions and show that it eliminates all memory inconsistencies due to repeated I/O operations and reduces wasted work by up to 68% and total execution time by up to 44% by avoiding 76% redundant I/O operations than InK [54] and Alpaca [34].

2 Background and Related Work

Recent advances in embedded systems have enabled the miniaturization of tiny sensing devices, enabling them to weave into the daily fabric of human life. A paradigm shift towards batteryless computing has already begun that allows these devices to power themselves using energy harvested from the environment that can be in any form, for example, solar or thermal. Such batteryless platforms like WISP [13], Flicker [25], Camaroptera [40], and Protean [5]

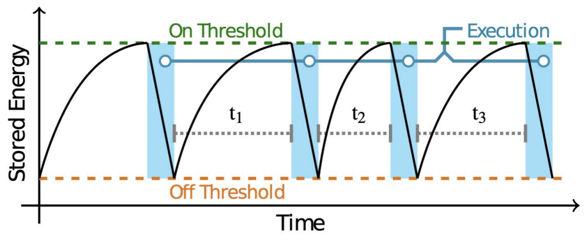


Figure 1. Energy harvesting devices compute intermittently with execution times unpredictably apart from each other. This makes it important to ensure efficient use of energy in order to ensure maximum program progress.

support heterogeneous peripherals, both internal (e.g., DMA) and external (e.g., accelerometers, microphones) peripherals, to plug in with the platform to sense the environment and perform necessary actuation while ensuring their deployment in different application settings [15, 19, 25, 40, 45]. In this section, we give the background of the problem by highlighting key challenges arising from the repeated execution of peripherals. We review existing state-of-the-art solutions for enabling the intermittent peripheral operation to identify their limitations to explain the motivation and design of EaseIO.

Batteryless system power themselves by harvesting energy from the highly variable and unpredictable environment [32], thus making these devices compute intermittently (see Fig. 1). Many existing runtime systems for batteryless devices use a checkpointing-based programming model [2, 10, 35, 51] to ensure the persistence of application state across power failures. Most approaches instrument the bare C code at compile time and decide to save a checkpoint at run time. Checkpointing, however, incurs significant overhead in terms of time and energy. Therefore, our focus in this paper is repeated I/O problems with task-based models.

Task-based programming models [6, 14, 20, 34, 38] provide a lightweight mechanism for representing a program into a set of atomic and independent tasks that back up the program state in non-volatile memory only after they complete. Since tasks have all-or-nothing semantics, a power failure in the middle of the task execution leads to the re-execution of the same task in the next power cycle. This situation also leads to the repetition of peripheral operations already performed in the previous energy cycle, causing many challenges to intermittent computing.

2.1 Problems with Peripheral-bound Tasks

Existing task-based systems cannot differentiate between peripheral and compute (non-peripheral) operations. Thus, they do not provide abstractions to programmers to avoid redundant re-execution or capture appropriate re-execution semantics, leading to the following problems.

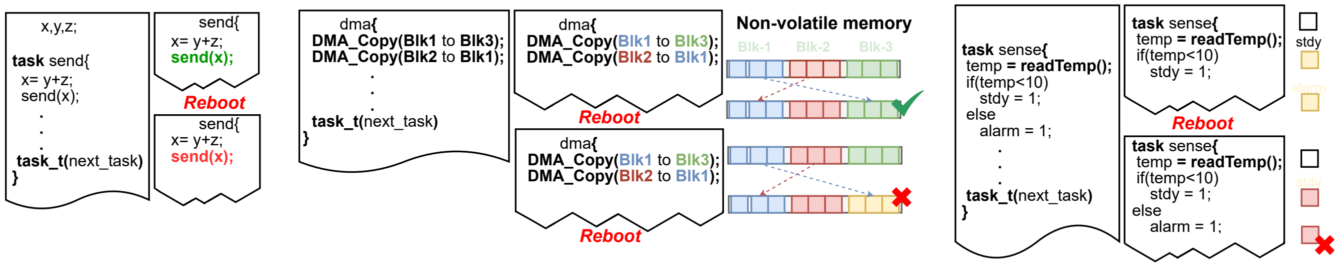
2.1.1 Wasted Energy Peripheral operations might have *single-shot* or *timely* re-execution semantics [26]. For instance, consider Figure 2a, in which the application was able to send data over the radio in the active cycle but was interrupted by a power failure. In this example, the task re-executes after a power failure and sends the data again, leading to wasted energy. Other operations may require re-execution only if a certain time has elapsed since the power failure. For example, sensor data has to be sensed again after a power failure if it is expired (e.g., not valid anymore). Redundant re-executions might even lead to a non-termination bug since I/O operations increase the task's energy requirements, which may exceed the energy buffer's capacity.

2.1.2 Idempotence Bugs Some peripheral operations, e.g, DMA bypasses the CPU and directly modifies non-volatile memory, which may lead to memory inconsistencies. Re-executing such peripheral operations can create idempotence bugs due to write-after-read (WAR) dependencies. Consider the example in Figure 2b, which shows a task containing two DMA operations that manipulate non-volatile memory. The first DMA operation copies *Block-1* to *Block-3*. The second DMA copies *Block-2* to *Block-1*. If a power failure occurs right after the second DMA operation, the first DMA will be re-executed after reboot. However, since *Block-1* is changed at the previous power cycle, this DMA operation would lead to memory inconsistency at *Block-3*. These WAR dependencies cannot be resolved at compile time since current runtimes can neither detect I/O operations nor track non-volatile memory locations manipulated by the peripherals.

2.1.3 Unsafe Program Execution I/O operations may determine the branch condition where each branch can manipulate different parts of non-volatile memory. If the task restarts after a power failure, it might take a different branch from the previous energy cycle since the repeated I/O operation might output a different value. For example, in Figure 2c, the application sets one of the persistent variables `stdy` and the `alarm` that will trigger the corresponding actuation (e.g., heating on/off), considering the temperature value. If the sensed temperature value is lower than 10, the `stdy` flag should be set. Otherwise, the `alarm` flag should be set. However, the power failure interrupts the task after setting the `stdy` flag. To make it worse, the `alarm` flag is also set in the next energy cycle since the sensed temperature value changed. Therefore, both flags were incorrectly set due to repeated I/O operations.

2.2 Limitations of Prior Arts

Table 1 presents a qualitative comparison of the main characteristics of EaseIO and the existing intermittent computing runtimes. Current task-based studies suffer from wasted work as they lack the necessary programming language support and fail to capture re-execution semantics for each



```
x,y,z;

task send{
 x= y+z;
 send(x);
 .
 .
 .
 task_t(next_task)
}
```

```
send{
x= y+z;
send(x);
```

```
send{
x= y+z;
send(x);
```

```
dma{
 DMA_Copy(Blk1 to Blk3);
 DMA_Copy(Blk2 to Blk1);
  .
  .
  .
 task_t(next_task)
}
```

```
dma{
DMA_Copy(Blk1 to Blk3);
DMA_Copy(Blk2 to Blk1);
```

```
dma{
DMA_Copy(Blk1 to Blk3);
DMA_Copy(Blk2 to Blk1);
```

```
task sense{
 temp = readTemp();
 if(temp<10)
   stdy = 1;
 else
   alarm = 1;
   .
   .
   .
 task_t(next_task)
}
```

```
task sense{
 temp = readTemp();
 if(temp<10)
   stdy = 1;
```

```
task sense{
 temp = readTemp();
 if(temp<10)
   stdy = 1;
 else
   alarm = 1;
```

(a) Wasteful I/O Operation **(b)** Idempotence Bugs **(c)** Unsafe Program Execution.

Figure 2. Task-based intermittent systems re-execute all peripheral operations regardless of their re-execution semantics. This leads to a waste of energy, inconsistencies in memory, and unsafe program execution.

Intermittent Runtimes	**Main Features of I/O Operations** *Repeated I/O Due to Power Failure*	*Wasted I/O Due to Power Failure*	Memory Inconsistency Due to Repeated I/O	*Safe DMA Operation*	*Timely I/O Operation*	Semantic Aware I/O Re-execution
Chain [14], Coala [38], Alpaca [34], Coati [44] Rehash [6], Mayfly [26], InK [54], CatNap [37], Immortal Threads [55]	Yes ✗	High ✗	Yes ✗	No ✗	No ✗	No ✗
IBIS [46, 48]	Yes ✗	High ✗	Yes (Due to DMA bugs) ✗	No ✗	No ✗	No ✗
Samoyed [36]	Yes (Atomic Functions) ✗	Medium ●	Yes (Atomic Functions) ✗	No (Atomic Functions) ✗	No ✗	No ✗
Ocelot [47]	Yes (Atomic Regions) ✗	Medium ●	Yes (Atomic Regions) ✗	No (Atomic Regions) ✗	No ✗	No ✗
EaseIO (this work)	**No** ✓	**Low** ✓	**No** ✓	**Yes** ✓	**Yes** ✓	**Yes** ✓

Table 1. A comparison of the main features of EaseIO with the relevant intermittent computing approaches.

peripheral operation, thus leading to repeated I/O operations [14, 26, 34, 54, 55]. These studies mainly capture data freshness by checking the inputs of the tasks and do not consider re-execution semantics for individual I/O operations within a task. If the input data is stale, they either do not execute the corresponding task or non-selectively repeat all I/O operations within the task body, leading to the problems we mentioned.

Samoyed [36] and Ocelot [47] rely on compile-time analysis to encapsulate peripheral operations in atomic function/regions as it ensures a consistent peripheral state across reboot. Both methods disable checkpoint interrupts before executing functions/regions to ensure atomicity, making it a task. If there is a power interruption in the region/function, all I/O operations are re-executed, causing the problems mentioned in Section 2.1. Recently, Surbatovich et al. [46, 48] proposed taint analysis at compile time to detect and circumvent memory inconsistencies. They do so by privatizing the tainted non-volatile variables at run time, thus avoiding unsafe program execution. However, such an analysis cannot cover DMA operations oblivious to the CPU and can directly modify non-volatile memory.

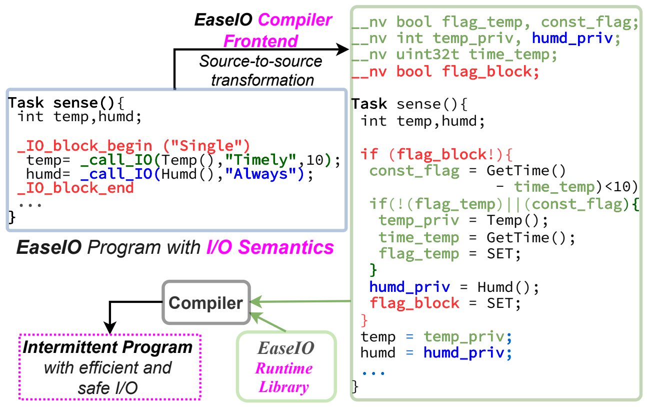
3 EaseIO Overview

EaseIO comprises two subsystems to make the task-based model safe and efficient for I/O-bound applications for batteryless devices. EaseIO's *programming language semantics* enables the programmer to express re-execution requirements for a peripheral operation. EaseIO *runtime* uses this information to re-execute peripheral operations while avoiding the challenges of redundant re-execution and memory inconsistencies. EaseIO has mainly the following features:

(1) **Programming expressiveness.** Programmers are aware of the timeliness and redundant re-execution constraints of peripheral operations. We expose APIs to the programmer to express these semantics, which reduces the burden of the programmer to handle these constraints manually.
(2) **Eliminate redundant I/O.** As peripheral operations are energy-hungry, their redundant re-execution wastes a significant amount of energy. EaseIO eliminates the unnecessary repetition of the peripheral operations annotated by the programmers.
(3) **Ensure safe I/O.** Application programmers are generally unaware of the challenges posed by intermittent energy supply when performing peripheral operations. By providing power-failure resilient APIs, EaseIO enables the programmer to write applications that can avoid memory inconsistencies that may arise due to the re-execution of peripheral operations.

3.1 Keywords for I/O Re-execution Semantics

EaseIO introduces three keywords to express peripheral re-execution semantics and constraints. Figure 3 shows the use


EaseIO Compiler Frontend
Source-to-source transformation
EaseIO Program with I/O Semantics
Compiler
Intermittent Program with efficient and safe I/O
EaseIO Runtime Library

```
Task sense(){
 int temp,humd;

 _IO_block_begin ("Single")
  temp= _call_IO(Temp(),"Timely",10);
  humd= _call_IO(Humd(),"Always");
 _IO_block_end
 ...
}
```

```
__nv bool flag_temp, const_flag;
__nv int temp_priv, humd_priv;
__nv uint32t time_temp;
__nv bool flag_block;

Task sense(){
 int temp,humd;

 if (flag_block!){
  const_flag = GetTime()
                - time_temp)<10)
  if(!(flag_temp)||(const_flag){
   temp_priv = Temp();
   time_temp = GetTime();
   flag_temp = SET;
  }
  humd_priv = Humd();
  flag_block = SET;
 }
 temp = temp_priv;
 humd = humd_priv;
 ...
}
```

Figure 3. The programmer calls I/O functions using EaseIO language interfaces and semantics. The compiler front-end handles the re-execution I/O semantics transformation considering task idempotency. EaseIO runtime library handles the re-execution of I/O functions and prevents memory inconsistency due to idempotence bugs.

of these keywords in the EaseIO subsystem. It is worth mentioning that with a continuous power supply, programmers do not need to specify such constraints since each peripheral operation is executed only once.

`Single` tells the runtime to execute a peripheral operation *only once*, i.e., if the operation was successful in the previous energy cycle, it does not require re-execution in the next cycle. Typical examples are data copying operations from volatile memory to non-volatile memory using DMA or sending the same data in the previous power cycle.

`Timely` tells the runtime if the data involved in the peripheral operation has timeliness constraints [21, 30]. If the data of the last I/O operation is still valid, then the runtime does not need to re-execute that operation again. As we mentioned in Section 2.2, existing works (e.g., Mayfly [26]) do not enable timeliness for peripheral operations within the tasks.

`Always` tells the runtime to re-execute peripheral operations after each power failure. This is the default policy of task-based systems since interrupted tasks are always repeated.

3.2 Programming Interfaces

EaseIO exposes interfaces for the programmer to make the previously mentioned semantics easier to use. Using the following three interfaces, developers can write a peripheral-bound intermittent application without dealing with repeated I/O challenges arising from frequent power failures. The main interfaces provided by EaseIO are depicted in Table 2.

`_call_IO` is an abstraction that enables the programmer to execute a peripheral operation along with the re-execution semantics. If a power failure occurs after the I/O function is executed, `_call_IO` decides to re-execute it after reboot,

Language Construct	Explanation
`_call_IO(`**`name,type`**`,...)`	call I/O function **name** according to the re-execution **type** with appropriate arguments
`_IO_block_begin(`**`type`**`,...)`	start I/O block according to the re-execution **type**
`_IO_block_end`	end of the I/O block
`_DMA_copy(`**`*src, *dst, size`**`)`	copy **size** of data from ***src** to ***dst** via DMA

Table 2. EaseIO language abstraction for I/O functions

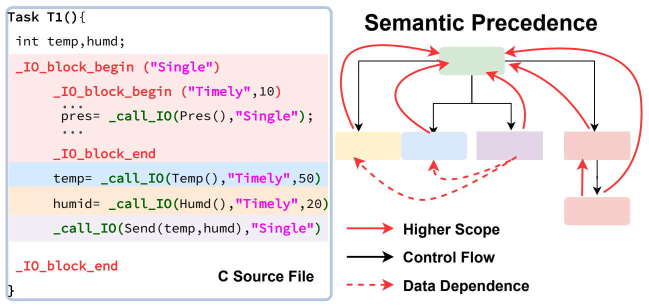
considering the annotated semantic at run time. `_call_IO` can invoke both void functions and functions that return a value. If the function returns a value, the compiler front-end creates a private copy of the variable in non-volatile memory to restore the last returned value after power failure.

`_IO_block_begin/_IO_block_end` defines the start/end of an atomic execution for multiple I/O functions considering EaseIO I/O semantic annotations. Since some I/O functions have temporal relation to each other, EaseIO presents `_IO_block_begin/end` blocks to annotate multiple I/O functions that need to be executed atomically with the same re-execution semantic. For example, in Figure 3, the programmer provided a code block that requires measuring humidity within 10 milliseconds after sampling the temperature. If the time constraint is violated, the temperature and humidity measurements will re-execute. The programmer wrapped `_call_IO(temp, Timely,10ms)` and `_call_IO(humid,Always)` in a `_IO_block_begin/end` structure so that these operations are atomically executed with the *Single* semantics. If these operations are executed successfully, EaseIO will not re-execute the whole block since it has single-shot semantics.

`_DMA_copy` performs block data copy from source to destination address via DMA peripheral. `_DMA_copy` detects the memory type of the source and destination address and resolves the re-execution semantics of the DMA operation at run time to prevent memory inconsistency due to WAR dependencies. For instance, if the copy is from non-volatile memory to non-volatile memory, the DMA operation is handled as *Single* since the non-volatile memory is persistent. On the other hand, if the copy is from volatile memory to volatile memory, the DMA operation is handled as *Always* since it should be always repeated.

3.3 Semantic Precedence

I/O operations outside an I/O block hold their own semantics. However, within an I/O block, I/O operations can have any level of nesting and can also contain another I/O block, as shown in Figure 4. In the given example in the figure, since the innermost I/O block has `Timely` semantic, all operations inside it should be re-executed when the time constraint of the block is violated. Therefore the `Timely` semantic of the block should have higher precedence than the `Single` semantic of the pres(). However, the `Single` semantic of



```
Task T1(){
 int temp,humd;
 _IO_block_begin ("Single")
      _IO_block_begin ("Timely",10)
       ...
       pres= _call_IO(Pres(),"Single");
       ...
      _IO_block_end
      temp= _call_IO(Temp(),"Timely",50)
      humid= _call_IO(Humd(),"Timely",20)
      _call_IO(Send(temp,humd),"Single")

 _IO_block_end
}
```

Figure 4. EaseIO handles precedence of re-execution semantics in the task considering the two types of dependency, the scope of the semantic and data dependency.

the outmost IO block should have higher precedence than all IO functions inside the block since the `Single` semantic should prevent re-execution of the block. Therefore we can not build precedence based on the semantic type between IO functions and IO blocks. To resolve such a scenario, we decide the precedence of each re-execution semantic based on two things: The scope of the semantic and data dependence between I/O operations.

3.3.1 Scope of the Semantic. In an I/O block, the precedence of each re-execution semantic is decided by its scope: higher scope means higher precedence. Consider the two I/O blocks as an example shown in Figure 4. The innermost I/O block contains an I/O operation that has `Single` whereas the I/O block has `Timely` semantic with a time interval of 10ms. Any interruption longer than 10ms after block operation requires re-execution of the whole block. However, since the `Pres()` function is annotated as `Single`, it will never re-execute. Since the re-execution semantic for the block has higher scope, we override the semantics of the internal I/O operation with `Timely` semantic.

3.3.2 Data-dependent I/O Operations. Some I/O operations may have data dependencies between each other thus forcing them to resolve the precedence between them. For example, there exists a data dependency between `Temp()`, `Humd()` and `Send()` functions represented in Figure 4. `Send()` function receives outputs of `Temp()` and `Humd()` as its inputs. In this case, a long interruption after `Send()` operation can lead to re-execution of `Temp()` or `Humd()` functions since they have `Timely` annotation. However, since `Send()` is `Single`, it never re-sends the updated values. In this scenario, the data inconsistencies between the last sent data and the stored data in the memory. To avoid such a scenario, EaseIO re-executes an I/O operation if the I/O operation it is dependent on was re-executed. We discuss the implementation details of this in Section 4.

3.4 Regionalizing

DMA operations can manipulate the nonvolatile variables indirectly. Therefore, EaseIO needs to handle DMA with care or otherwise, it can cause WAR bugs to arise because of DMA operations handling nonvolatile variables. Existing state-of-the-art work [34, 54, 55] creates private copies of non-volatile variables to avoid WAR bugs to make non-volatile variables consistent at the beginning of the task. However, since EaseIO runs the DMA operation with different re-execution semantics, including `Single` semantics, the existing privatization technique is not viable for EaseIO. For instance, if a `Single` annotated DMA manipulates a non-volatile variable `temp`, the traditional privatization techniques make `temp` consistent at the region before the DMA. But the `temp` would be inconsistent at the region after the DMA since the DMA couldn't update the `temp` again due to the Single annotation.

To solve this problem, EaseIO separates tasks into regions according to the DMA positions and number in the task body, thus keeping non-volatile variables consistent in each region by privatizing them in each region.

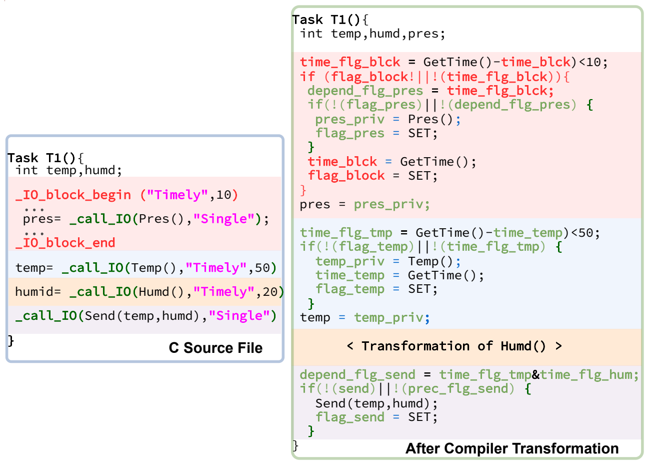
3.5 Correctness

EaseIO prevents unsafe program execution by holding a private copy of the successful I/O outputs. On each power failure, EaseIO avoids re-execution of the I/O function and restores the last successful/valid output value of the I/O operation before restarting task execution. This mechanism helps the program to avoid taking branches that would not have been taken in continuous execution. To avoid memory inconsistencies, EaseIO decides the re-execution of the DMA operation considering the memory type of the source and target data. If it decides to re-execute a DMA that has source data in non-volatile memory, EaseIO separates the first execution of the DMA into two DMAs by using a privatization buffer to hold a private copy of the source data. EaseIO uses the private copy as source data during the re-execution to avoid memory consistency.

EaseIO re-execution semantics ensure that the programmer avoids non-termination. If a task's energy cost exceeds the energy budget of the capacitor, the system would encounter a non-termination bug. When an I/O operation is complete, EaseIO prevents its re-execution in the next energy cycle thus allowing the system to spend this energy on performing useful computations and making application progress in the next energy cycle. Thus, EaseIO avoids non-termination by preventing redundant re-execution of I/O operations.

4 EaseIO Implementation

We implemented EaseIO using macros, compiler directives, and a compiler front-end that utilizes the LLVM and Clang LibTooling [1] framework for source-to-source transformations. The transformation converts the I/O call into C code and identifies redundantly executed I/O operations with a control block.



```
Task T1(){
 int temp,humd;
 _IO_block_begin ("Timely",10)
  ...
  pres= _call_IO(Pres(),"Single");
  ...
 _IO_block_end
 temp= _call_IO(Temp(),"Timely",50)
 humid= _call_IO(Humd(),"Timely",20)
 _call_IO(Send(temp,humd),"Single")
}
```

```
Task T1(){
 int temp,humd,pres;
 time_flg_blck = GetTime()-time_blck)<10;
 if (flag_block!||!(time_flg_blck)){
  depend_flg_pres = time_flg_blck;
  if(!(flag_pres)||!(depend_flg_pres) {
   pres_priv = Pres();
   flag_pres = SET;
  }
  time_blck = GetTime();
  flag_block = SET;
 }
 pres = pres_priv;
 time_flg_tmp = GetTime()-time_temp)<50;
 if(!(flag_temp)||!(time_flg_tmp) {
   temp_priv = Temp();
   time_temp = GetTime();
   flag_temp = SET;
  }
 temp = temp_priv;
       < Transformation of Humd() >
 depend_flg_send = time_flg_tmp&time_flg_hum;
 if(!(send)||!(prec_flg_send) {
   Send(temp,humd);
   flag_send = SET;
  }
}
```

Figure 5. EaseIO transformation for the programmer annotated source code.

4.1 Target Hardware

The current EaseIO library implementation targets the TI-MSP430FR5994 microcontroller with 256KB FRAM and 8KB SRAM. EaseIO library supports a persistent time circuitry [18] to ensure timely re-execution of I/O operations where necessary. However, it can be easily extended for other resource-constrained microcontrollers.

4.2 Semantic-aware I/O Re-execution

This section discusses the implementation details of enabling semantic-aware re-execution, which is key to avoiding redundant execution of peripheral operations. We propose three semantics – `Single`, `Timely`, and `Always` – each handling a different context.

`Single` semantic holds a dedicated boolean flag; e.g., `flag_pres` in Figure 5; for the corresponding I/O call in the task. After the successful execution of each I/O operation, this flag gets set, allowing the runtime to track I/O operation completion. This transformation requires careful consideration, as functions with return values may lead to data inconsistencies if not correctly restored. EaseIO avoids such a scenario by maintaining a non-volatile private copy of the returned value. If the function fails to be re-executed ever or until the time violation, EaseIO restores the latest value, e.g., `pres_priv`, shown in Figure 5. Therefore, if the runtime decides not to re-execute the I/O operation, the private copy of the return value is restored, ensuring the correct execution of the program.

`Timely` semantic captures a timestamp at the time of the last execution of the operation, in addition to maintaining a boolean flag for the I/O completion check. This enables the runtime to re-execute the operation if a specific time has elapsed since its last execution. The same logic applies to an `IO_block` definition allowing multiple re-execution semantics for the multiple I/O functions in the same task.

`Always` semantic does not add any new logic and relies on task-based models to help re-execute I/O operations. We add this semantic to remain consistent with the task-based programming model.

EaseIO stores the output of all `call_IO` operations during reboot and ensures that the program always takes the same path as it would on a continuous power to avoid unsafe program execution.

4.2.1 Semantic Precedence.

As discussed in Section 3.3, within a block, an `IO_block` has precedence over execution semantics for individual I/O operations. Therefore, all function semantics are replaced during a block semantic violation with the block's re-execution semantics.

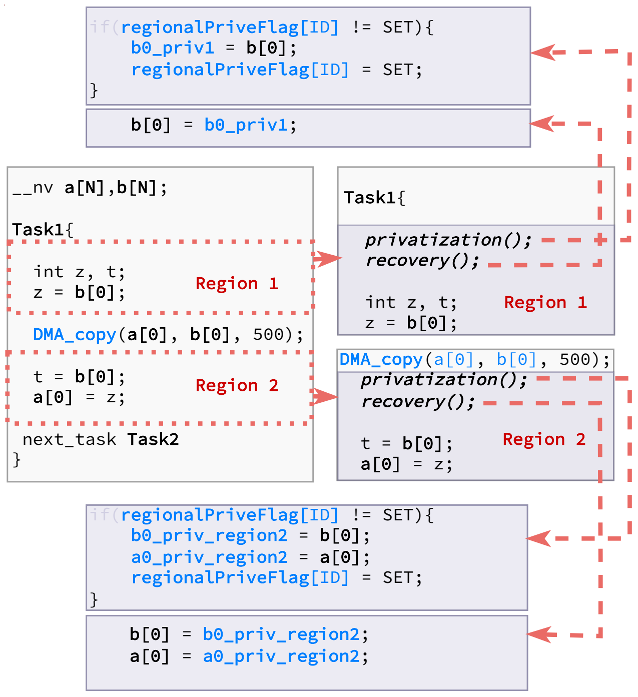
EaseIO implements this precedence with an additional boolean flag that checks the validity of the block's semantics. If they are invalid, EaseIO executes all I/O operations within the block regardless of whether the functions were already executed. To illustrate, let us assume a `Timely` annotated block that includes a `Single` annotated function. EaseIO adds a `depend_flg_#fn_name#` variable in the code to check the validity of the I/O block's time constraint. This way, EaseIO can re-execute `Single` annotated functions more than once, depending on the block's semantics.

4.3 Enabling Memory-Safe DMA Operations

DMA operations can modify non-volatile memory without CPU intervention, which may lead to idempotence bugs and memory inconsistencies, as discussed in Section 2.2. EaseIO runtime decides the semantics for DMA operations based on the source and destination memory address of the DMA copy operation in order to circumvent these inconsistencies:

(i) Volatile/non-volatile to non-volatile. If the destination address points to the non-volatile memory, in that case, the data will reside at the same location if the operation was successful in the previous energy cycle, irrespective of whether the source memory address is in volatile or non-volatile memory. EaseIO annotates such DMA copy operations as `Single` at run time to prevent re-execution after a power failure.

(ii) Non-volatile to volatile memory. If the source address points to non-volatile and the destination address points to volatile memory, in that case, we can not annotate the DMA operation as `Always` since a power failure clears volatile memory. Therefore, this DMA can be re-executed after each reboot. However, since the source address points to the non-volatile memory, a successive DMA operation might manipulate the source address, which leads to WAR dependency. Thus, we separate the DMA copy operation into the two DMA operations at run time via a *two-phase commit*. EaseIO copies the source data to a privatization buffer in the first phase. In the second phase, it copies from the



```
if(regionalPriveFlag[ID] != SET){
   b0_priv1 = b[0];
   regionalPriveFlag[ID] = SET;
}
```

```
   b[0] = b0_priv1;
```

```
__nv a[N],b[N];

Task1{
  int z, t;
  z = b[0];              Region 1

  DMA_copy(a[0], b[0], 500);

  t = b[0];              Region 2
  a[0] = z;

  next_task Task2
}
```

```
Task1{

  privatization();
  recovery();

  int z, t;          Region 1
  z = b[0];
```

```
DMA_copy(a[0], b[0], 500);
  privatization();
  recovery();

  t = b[0];          Region 2
  a[0] = z;
```

```
if(regionalPriveFlag[ID] != SET){
   b0_priv_region2 = b[0];
   a0_priv_region2 = a[0];
   regionalPriveFlag[ID] = SET;
}
```

```
   b[0] = b0_priv_region2;
   a[0] = a0_priv_region2;
```

Figure 6. Regional Privatization: EaseIO compiler front-end separates Tasks which contain DMA into the regions considering the DMAs position in the Task. The EaseIO runtime only considers the DMA operation complete when Regional Privatization successfully ends.

privatization buffer to the destination address. If any DMA operation modifies the source data, we reserve the private copy in the privatization buffer. Thus, EaseIO annotates such DMA operations as `Private`.

(iii) Volatile to volatile. If the source and destination addresses point to the volatile memory, the DMA copy does not lead to any memory inconsistencies when re-executed. The EaseIO runtime annotates this DMA operation as `Always`.

DMA operations on constant data, e.g., copying constant coefficients from non-volatile memory to volatile memory, do not have WAR dependencies and, thus, are safe when re-executed after power failure. Thus, this DMA operation into the two DMAs with `Private` annotation wastes valuable time, memory, and energy. Therefore, EaseIO supports a `Exclude` annotation that allows programmers to exclude DMAs from the privatization process to reduce system overhead. When the programmer uses this annotation for a `_DMA_copy` interface, the EaseIO compiler sets the `DMA_Type` of that DMA operation as `Always` at compile time. Thus, EaseIO skips the annotating and privatization processes at run time for that DMA.

4.3.1 I/O and DMA Data Dependencies.

Some DMA operations may depend on the output data of an I/O operation. For all such cases, EaseIO's runtime annotates these DMA operations with the same re-execution semantics as the I/O operation. For example, let us assume a DMA operation copies the output of the `Always` annotated I/O operations to the non-volatile memory. EaseIO annotates this DMA operation as `Single` since the destination is in the non-volatile memory. If a power failure occurs after the DMA operation is completed, the `Always` annotated I/O operations re-executes. However, the new result can not be saved to non-volatile memory since `Single` annotated DMA does not re-execute. A similar case ensues when DMAs copy `Timely` annotated I/O operation results. To handle this scenario, EaseIO's compiler front-end parses the task body to detect data dependencies between `_call_IO` and `_DMA_copy` interfaces. EaseIO assigns the `constraint_check` flag of the `_call_IO` interface to the `RelatedConstFlag` of the `_DMA_copy` interface. If the related I/O operation is *Always*, EaseIO sets the `RelatedConstFlag` of `_DMA_copy` to 1. The `RelatedConstFlag` is set to 0 if the corresponding I/O operation is `Single` or if there is no related I/O operation.

These runtime decisions enable EaseIO to ensure memory safety and consistency under intermittent energy supply.

4.4 Regional Privatization

Non-volatile variables manipulated by the CPU can also be part of the source or destination of a DMA operation, which is problematic for memory consistency. Consider a task that copies a value from a non-volatile buffer to another non-volatile buffer. By the rules mentioned above, the runtime would label such a DMA operation as `Single`. However, avoiding re-execution of such DMA operation would cause data inconsistencies. For example, if a power failure interrupts the task after the `a[0]` is modified in region 2 in Figure 6, the existing task-based privatization method [34] would restore the initial value of the non-volatile variables. Since the DMA operation is annotated as `Single`, it would not be re-executed in the next energy cycle, thus leading to an inconsistent value due to WAR dependency for the variable `b[0]` after reboot. In addition, changed I/O results due to re-execution can cause memory inconsistency on the non-volatile during the branch operations variables, as mentioned in Section 2.1.

To ensure safe program execution, EaseIO compiler handles the WAR dependencies between non-volatile variables and DMAs using a novel approach named *Regional Privatization.* EaseIO separates the task into multiple regions considering the DMA operation locations. EaseIO separates a task that contains *N* number of DMAs into the *N+1* number of regions. If the task includes no DMA operations, EaseIO considers the whole task a single region for the privatization process. For instance, since `Task1` has one DMA operation in Figure 6, the compiler divides it into two regions. EaseIO compiler places the regional privatization and recovery processes at the beginning of each region, respectively. The compiler creates dedicated private copies for each region to ensure regional idempotency. EaseIO sets the regional privatization flag, which shows the privatization process is completed at the end of the privatization process to signal the `_DMA_copy` function. Thus, EaseIO achieves the DMA operation and the privatization process atomically. If a power failure interrupts the task after regional privatization is completed, EaseIO recovers non-volatile variables utilizing the private copies. Since EaseIO accomplishes the regional privatization after the DMA, it avoids the inconsistencies due to no re-execution of the DMA in the next energy cycle.

As regional privatization allows private copies of all non-volatile variables in the region, it overcomes unsafe program execution problems by avoiding program paths that would not have been visited in continuous execution.

4.5 Compiler Frontend

We implemented the compiler front-end using the LLVM and Clang LibTooling [1] framework, which is responsible for performing the source-to-source transformations. The transformations include converting the I/O call into C code and the control block required to check the redundant execution of the I/O operation. We implemented our compiler front end as a combination of `AST-matcher` to match the call to our API when traversing the AST. For each match, we bind a callback to insert the necessary code blocks required to insert control logic which keeps the information about the previous execution of the I/O operation. Figure 5 presents how the compiler front-end transforms `_call_IO` and `_IO_block_begin/end` into the `if` structure. Since the `if` structure conditions change according to the re-execution semantic types, the compiler considers the annotation type during source-to-source transformation. Each `if` structure controls the *lock flag* dedicated to the called function to comprehend I/O function completion. However, multiple tasks can invoke the same I/O function, or a task can invoke the same I/O function multiple times. Therefore, the compiler front-end creates a *lock flag* in non-volatile memory named `lock_##functionName##taskName##num`, considering each `_call_IO`'s function name, task name, and calling number.

4.5.1 Regional Privatization for DMA operations.

EaseIO's compiler front-end parses the body of each task in the application to extract non-volatile variable accesses to avoid idempotence bugs due to repeated I/O operations. For each non-volatile variable access, the compiler creates a privatization buffer to undo any temporary change in the buffer that may be there due to power failure.

5 Evaluation

We evaluate EaseIO against two state-of-the-art runtimes (Alpaca [34] and InK [54]) in two phases using an emulated energy environment. In the first phase, we evaluate EaseIO on a uni-task application, where an application performs a single type of I/O operation. The second phase evaluates EaseIO with a multi-task application that performs three different types of I/O operations. Our results show that by preventing unnecessary re-executed I/O operations and ensuring safe DMA operations across power failures, EaseIO reduces wasted work **up to 3 times**.

	Alpaca		**InK**		**EaseIO**	
	Tasks	*I/O func.*	*Tasks*	*I/O func.*	*Tasks*	*I/O func.*
LEA	3	1	3	1	3	1
DMA	3	1	3	1	3	1
Temp.	3	1	3	1	3	1
FIR filter	5	1	5	2	5	2
Weather App.	11	5	11	5	11	5

Table 3. Tasks and I/O functions of evaluated applications.

5.1 Target Platforms and Tools

We use the MSP430FR5994 [49] at 1 MHz operating frequency, which includes 256kB FRAM and 4kB SRAM memory. We compile our applications using the GNU GCC v9.3.1.0 and measure execution time and energy consumption using a logic analyzer and TI EnergyTrace software [50]. Our evaluation board connects to a P2110-EVB RF receiver [17] which harvests power from the Powercast TX91501-3W RF transmitter [16] operating at 915 MHz center frequency.

We emulate the energy conditions, including power failures, to reproduce and repeat the comparative measurements. The power failure is simulated by random soft resets triggered by an MCU timer with a uniformly distributed firing period in the interval of [5ms, 20ms].

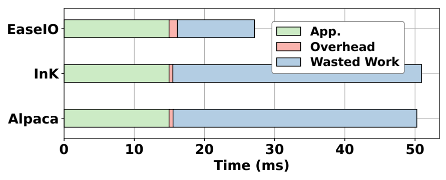
5.2 Evaluation Metrics

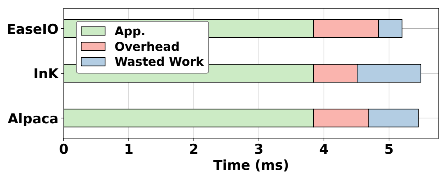
We consider *five* evaluation metrics – (1) *Wasted Work* represents the computational progress lost due to power failure as the number of and time corresponding to unnecessary re-executions and power failures; (2) *Energy Consumption* is the energy consumed to finish a single execution of the target application; (3) *Execution Correctness* denotes the number of correct execution of DMA operations with WAR dependencies; (4) *Runtime Overhead* is the additional time added to the execution time to progress computation and keep memory consistency across power failure; and (5) Memory Overhead denotes the additional memory and code size required by the runtime.

We implement applications for InK and Alpaca's runtime with atomic tasks having all-or-nothing semantics. Table 3 shows the number of tasks and I/O functions for each application.

5.3 Phase 1: Uni-Task Application

In this phase, we execute three applications introduced in Samoyed [36] representing each I/O semantic described in Section 3.1. For example, temperature sensing (temp) requires `Timely` semantics, whereas DMA operation copying data from non-volatile to non-volatile memory has a `Single`


(a) `Single` semantic - NVM to NVM DMA


(b) `Timely` Semantic - Temperature Sensing

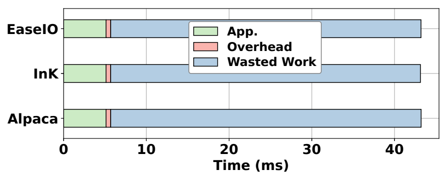

(c) `Always` Semantic - LEA

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

	`Single` (DMA)		Timely (Temp.)		Always (LEA)	
	PF	*Re-exe.*	*PF*	*Re-exe.*	*PF*	*Re-exe.*
Alpaca	3749	3624	402	334	1359	1314
InK	3749	3624	403	335	1359	1321
EaseIO	1996	863 (↓76%)	385	190 (↓43%)	1359	1314 (0%)

Table 4. EaseIO reduces the number of power failures since it save energy from unnecessarily executed I/O operations allowing it to spend more energy on performing useful computations.

re-execution semantics. For applications performing LEA operations, we annotate the task with `Always` semantics. Each application is executed 1000 times with pseudo-random seeds, and we report the average results. Each application contains a minimum number of shared variables to fairly observe the overhead of the handling I/O semantics of EaseIO.

5.3.1 Wasted Work and Runtime Overhead Results

Figure 7 shows each uni-task application's total execution time, highlighting the application's execution time (APP), runtime overhead, and wasted work. We can observe that, despite having **108**% higher runtime overhead than InK and Alpaca, EaseIO has **216**% lower wasted work, which reduces the total execution time by **85**% (Figure 7a). While InK and Alpaca re-execute all I/O operations, EaseIO avoids redundant re-execution with the help of semantics re-execution integrated with EaseIO's programming interfaces to execute an I/O operation only if it is necessary.

Energy saved by avoiding redundant re-execution allows the device to go father on the same charge, thus making the application make more progress in a single energy cycle and reducing the number of power failures before completing the workload. Table 4 shows that avoiding redundant re-execution results in **76**% less I/O operations, reducing power failures by **0.1% − 46%** compared to InK and Alpaca. The lack of shared variables in the DMA application produces similar overhead for InK and Alpaca. EaseIO requires higher overhead than InK and Alpaca for `Timely` operations since it requires time stamping and checking during the `Timely` I/O operations, as shown in Figure 7b. Despite this increased overhead, EaseIO reduces the wasted work overall due to power failures by preventing the re-execution of valid I/O operations after reboot.

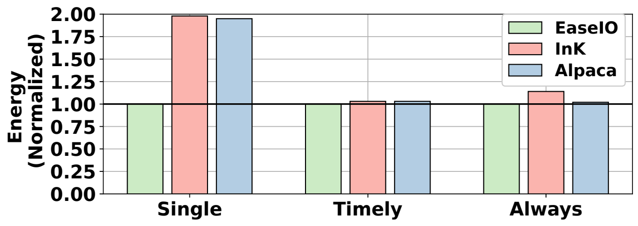


Figure 8. Average energy consumption of for re-execution I/O semantics with controlled power failures.

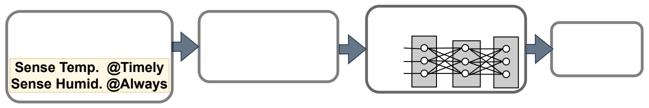


Figure 9. Weather Classification Application

5.3.2 Energy Consumption Results.

Reducing power failures and execution translates into reduced energy consumption of the system. Figure 8 demonstrates a one-half average reduction in energy consumption of the task.

5.4 Phase 2: Multi-Task Applications

In phase 2, we develop two multi-task applications containing all three types of I/O semantics, including DMA operations with WAR dependencies.

5.4.1 Applications.

We developed an FIR filter and DNN-based weather classifier application as they require multiple I/O operations with different execution semantics.

The FIR filter includes three DMA and one LEA operation that we exploit to evaluate application correctness. The first two DMA fetch input and filter coefficients data from non-volatile memory to volatile LEA-RAM. The input signal is divided into four samples, and four LEA calls complete the filtering operation in a loop. The last DMA saves the results from LEA-RAM to the non-volatile memory. The input and output of the application use the same buffer in the non-volatile memory. Therefore, we call the LEA using the `Always` semantic, and EaseIO annotates the first two DMA as `Private` and the last DMA as `Single` at run time.

The weather classifier is a complex application that contains multiple I/O operations and hardware accelerators to

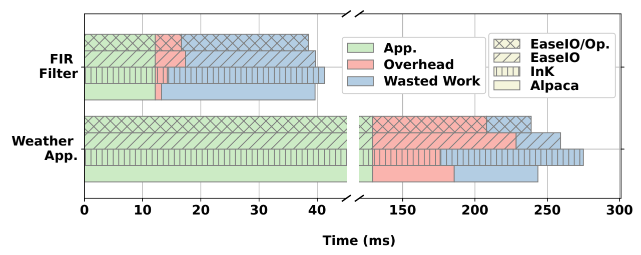


Figure 10. The execution time, runtime overhead and wasted work of weather classifier and FIR filter with controlled power failures.

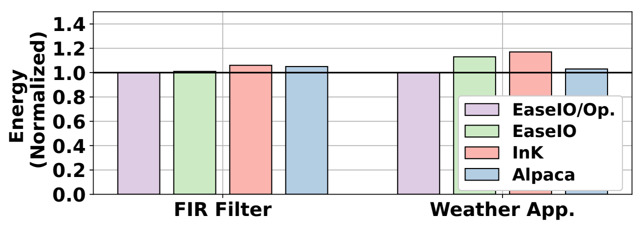


Figure 11. Average energy consumption of multi-task applications with controlled power failures.

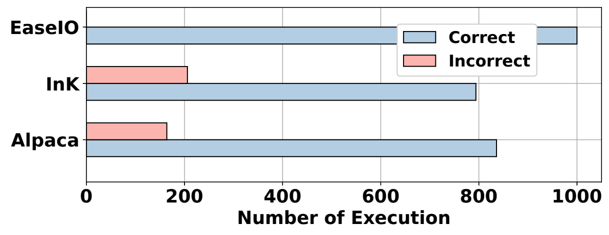


Figure 12. The numbers of correct and incorrect execution of the FIR Filter.

evaluate all system performance. The application consists of four main steps, as shown in Figure 9: (i) sense temperature and humidity values, (ii) capture an image of the weather, which we simulate by running the microcontroller in a delay loop. (iii) infer weather conditions using DNN, (iv) send the inference result, temperature, and humidity value. Since temperature and humidity sensing operations are time-dependent, we used the `Timely` semantic for the temperature sensing in a `Single` annotated `IO_block`. In contrast, the send operation is annotated as `Single`. Figure 9 shows the workflow of the application, which is divided into 11 tasks.

The DNN contains five layers: (i) the 1x4x4 convolution layer, (ii) the RELU layer, (iii) the 1x4x4 convolution layer, (vi) the fully connected layer, and (v) the inference layer. The convolution and fully connected layers utilize the LEA and DMA hardware like in TAILS. We send the temperature, humidity, and weather inference in the last step. We also simulate this transmitter operation using the same method as in step two.

5.4.2 Wasted Work and Runtime Overhead Results. Figure 10 shows the execution time, overhead, and wasted work of the FIR Filter and DNN-based weather classifier. The two `Private` DMAs of FIR filter require privatization resulting in higher overhead than Alpaca and InK. However, with the reduced wasted work by preventing unnecessary execution, EaseIO saves overall execution and energy by avoiding redundant I/O operations. To reduce overhead further, we also employ `Exclude` API for the DMAs that copy constant nonvolatile coefficients of the filter, decreasing the privatization process. We call the results for this `Exclude` API as "EaseIO /Op" in our evaluation. "EaseIO /Op" completes application execution almost simultaneously as Alpaca.

The DMA fetches input data from non-volatile memory to LEA-RAM for weather applications before triggering the LEA. EaseIO annotates these two DMA as `Private` at runtime because of a probable WAR dependency, as mentioned in Section 4.3. EaseIO separates them into two DMAs using a privatization buffer to protect memory consistency. Figure 10 shows that EaseIO has a more significant overhead as the destination buffers of the DMAs are in volatile memory; they re-execute and require privatization after each reboot. Since Alpaca and InK can not build a re-execution semantic or temporal relation between I/O operations, they execute the temperature and humidity sensing operations after each reboot. On the contrary, EaseIO's timely and temporally links these sensing operations using `IO_block` and re-execution semantics and reduces wasted work due to power failures by **up to 3 times**.

5.4.3 Energy Consumption Results. Reducing wasted and redundant re-execution reduces the system's overall energy consumption significantly. Figure 11 shows both applications' energy consumption reduction. EaseIO reduces the average energy consumption of FIR filter and weather classification applications **by up to 5% and 17%**, respectively.

5.4.4 Execution Correctness. Figure 12 shows that EaseIO protects the FIR application from memory inconsistencies during intermittent execution. Since the DMA operations have WAR dependencies, InK and Alpaca can produce **21% and 16%** incorrect results, depending on the power failure interruption point. However, EaseIO uses a privatization buffer to handle WAR dependencies and produces correct results despite power failures.

For applications using DNNs, programmers use two buffers for input and output data of each DNN layer [23, 55] to avoid WAR dependency. However, our *Regional Privatization* of EaseIO allows the programmer to use a single buffer for the DNN implementation without having idempotence violations. Table 5 shows the execution time

	Double Buffer			Single Buffer		
	Cont.(ms)	*Int.(ms)*	*Corr.*	*Cont.(ms)*	*Int.(ms)*	*Corr.*
Alpaca	185.63	243.54	✓	185.63	242.78	✗
InK	175.94	275.65	✓	175.94	274.92	✗
EaseIO	228.45	259.14	✓	228.45	257.63	✓

Table 5. Execution times and correctness of Weather Classification App for double-buffered and single-buffered DNN.

	Alpaca			InK			EaseIO		
	.text	*Ram*	*Fram*	*.text*	*Ram*	*Fram*	*.text*	*Ram*	*Fram*
LEA	1344	2064	956	2992	2056	4226	2123	2065	4980
DMA	914	8	8242	2188	8	12064	1772	9	12242
Temp.	784	8	314	3042	8	3231	1566	9	4320
FIR Filter	1220	2048	3554	2338	2048	5225	2432	2049	7564
Weather App.	2838	2384	776	2978	2384	4310	3548	2385	4886

Table 6. Memory and Code Size requirements (in B).

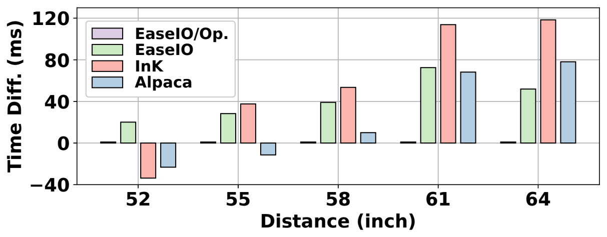


Figure 13. The execution time performance normalized by EaseIO /Op. that includes *Exclude* annotation for real energy harvester.

and correctness for the DNN implementations using the single and the double buffer. InK and Alpaca cannot ensure correct execution with a single buffer under intermittent energy supply, whereas EaseIO completes the execution. For double buffers, EaseIO's execution time is close to both systems.

5.4.5 Memory Overhead. Finally, we investigate the memory overhead of EaseIO as a complex DNN application with a larger memory footprint, which can lead to more considerable memory overhead. The EaseIO runtime requires almost 1 KB more `.text` size than Alpaca due to Regional Privatization and DMA handling, as shown in Table 6. The programmer can define the DMA privatization buffer size at compile time. For example, we use a 4 KB privatization buffer in our evaluation. However, when an application has no DMA operation, this buffer size can be set to zero. For example, the temperature sensing application does not include the DMA operation and thus has no DMA privatization buffer. Therefore, EaseIO loads a 6-byte overhead for the I/O semantic implementation.

5.5 Real-World Evaluation

We also evaluate the performance of our system in a real-world scenario. We employ an RF power transmitter to harvest energy and charge a 1mF capacitor as energy using five different distances between the transmitter and the MCU.

Figure 13 shows the execution time results difference between "EaseIO /Op." and the rest. When the transmitter is close to the MCU, the system can harvest enough energy, so there are no power failures. However, as the distance increases, power failures begin to occur. The results show that in such a scenario, EaseIO consistently performs better than Alpaca and Ink as it prevents unnecessary re-execution of the I/O functions.

6 Discussion

EaseIO-Blocks vs Atomic Regions. Subartovic et al. [47] proposed atomic regions to ensure I/O freshness. These regions contain instructions dependent on the results of an I/O operation and execute them atomically to avoid memory inconsistency problems. However, atomic regions do not prevent repeated I/O operations. On the contrary, the EaseIO-block definition can avoid repeated I/O as long as the semantics of the operation remain valid.

Tasks vs Regions. EaseIO-regions are conceptually different from atomic and idempotent tasks in task-based programming models since they don't have all-or-nothing semantics. They are re-executed (within the task) after reboot to ensure memory consistency, application safety, and liveness.

Regional Privatization vs Task-based Privatizations. Traditional task-based systems use privatization for variables with WAR dependencies or double buffers for all non-volatile variables to ensure memory consistencies. Both methods swap original and private copies or indexes of buffers at the beginning or end of the task. However, these methods are insufficient for EaseIO since EaseIO might skip some DMA operations after power failures. These methods can only ensure memory consistency until the first ignored DMA in the task. However, they can not guarantee memory consistency if the DMA manipulates any non-volatile variable in the task. Therefore, EaseIO introduces regional privatization, which ensures memory consistencies by taking private copies of non-volatile variables in each region.

DMA Privatization Buffer Limits. EaseIO uses a set of privatization buffers to ensure safe DMA copy operation, significantly increasing memory overhead. Buffer sharing between DMA operations reduces this overhead. However, programmers need to ensure that the size of the DMA copy operation does not exceed the size of the privatization buffer to enable this sharing. In our future work, we plan to implement a compile-time analysis to keep track of the size of `_DMA_copy` interfaces and generate a compiler error message if it exceeds the limit.

Re-execution Semantics in Loops. EaseIO can be extended to ensure the semantic safety of nested I/O operations that use loops to fill elements of an array. For example, the programmer can nest an I/O operation in a loop to collect several samples from the sensor. EaseIO compiler creates the `lock_flags` of `_call_IO`s in the loop as a loop-sized array. The compiler creates a private copy of each array element to restore them in the next energy cycle. Therefore if a power failure occurs in the loop, each sample is re-executed considering the EaseIO language semantic. For sequential I/O operations like collecting periodic samples, the programmer should use a dedicated `_call_IO` interface for each sample.
Programmer Burden. EaseIO is the first step towards allowing re-configurable I/O support as it provides the API and runtime support to define I/O re-execution semantics. However, it is the programmer's responsibility to identify semantics for each I/O operation, introducing an additional burden on the programmer. An automated system requires identifying time-dependent data, power failure prediction, and WAR dependencies in the program to correctly identify bugs arising from I/O re-execution, which requires a compile-time analysis. In the future, we aim to build compiler support for such analysis and, thus, automate the annotation process.
Asynchronous Peripheral Operations Peripheral state management is orthogonal to our work. Therefore, as in Samoyed, EaseIO currently targets arbitrarily restartable peripherals, which do not have an internal non-volatile state. The current implementation of EaseIO can support only synchronous peripherals, due to its flagging techniques to keep track of the completion of I/O functions. Since EaseIO sets the completion flag at the end of the `call_IO` function, EaseIO might set the completion flag while the peripheral is still processing during the asynchronous operations, which is problematic. Similarly, the block flag might be set for the I/O block operations while nested I/O functions are still processing. Therefore, the peripheral operations should be synchronous to guarantee to set the completion flag after completing the I/O operation.

7 Additional Related Work

This section briefly discusses a few previous works related to our work and highlights similarities and key differences from the proposed work.

7.1 Intermittent Computing

A vast majority of existing literature enabling software support for batteryless devices focus on checkpointing the computational [3, 10, 42] and peripheral [9, 12, 43] states before power failure. Such a system either works with statically placed checkpoints in the program code [2, 10, 42] or adopts JIT checkpointing model [8, 10, 29] to back up the system state just before the power failure is imminent. While static checkpoints suffer from code re-executing, the JIT checkpoints strategy avoids repeated I/O and reduces wasted work. However, it fails to guarantee timely I/O operations and requires additional hardware and constant pooling of the energy buffer to detect to trigger a checkpoint, resulting in additional energy overhead.

7.2 Regional Privatization

Our regional privatization is similar to JustDo [28] and Clobber logging (undo+Justdo) [52] proposed in the existing literature. JustDo logging strategy tracks necessary program states to resume execution from the interrupted instruction. However, JustDo increases runtime overhead by keeping track of every STORE instruction to the non-volatile memory. Furthermore, it does not allow volatile memory usage since it does not roll back any instruction. Clobber logging detects overwritten input of the non-volatile memory and includes them in the recovery process to reduce undo-logging overhead. However, both these approaches focus on main-stream computing devices that have significantly more energy when compared with batteryless devices. Energy is scarce, and efficient energy use is essential in ensuring correct application execution for batteryless sensors.

8 Conclusion

EaseIO is the first intermittent runtime that introduces re-execution semantics for the I/O operations and handles idempotence bugs due to repeated I/O operations. The programmer can use EaseIO language interfaces and semantics to express re-execution requirements for a peripheral. Thanks to the *Regional Privatization* and `_DMA_copy` interface, the programmer can design their code without considering WAR dependencies on the DMA operations and non-volatile variables. Our evaluation shows that EaseIO can reduce the wasted work due to power failures **up to 3 times** against the state-of-the-art task-based runtimes and can ensure the safe DMA copying operation while the other runtimes can not.

Acknowledgments

We thank the anonymous reviewers of ASPLOS 2022 and EuroSys 2023 for their valuable comments. We are also grateful to Aurojit Panda for shepherding our final draft. Eren Yildiz was a visiting Ph.D. at Northwestern University to carry out this study and he was supported by the Scientific and Technological Research Council of Türkiye (TUBITAK) under the 2214-A program. This work was also partially funded by the European Union under Horizon Europe Programme - Grant Agreement 101070537 — CrossCon, and by the National Science Foundation under award numbers CNS-2145584 and CNS-2107400. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the granting agencies: the European Union, European Climate, Infrastructure, Environment Executive

Agency (CINEA), or the National Science Foundation. Neither the European Union nor the granting authorities can be held responsible for them.

References

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

[2] 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.

[3] Khakim Akhunov and Kasim Sinan Yildirim. Adamica: Adaptive multicore intermittent computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(3):1–30, 2022.

[4] ARM. An update on arm's ai journey toward a trillion connected devices, September 2019.

[5] Abu Bakar, Rishabh Goel, Jasper de Winkel, Jason Huang, Saad Ahmed, Bashima Islam, Przemysław Pawełczak, Kasım Sinan Yıldırım, and Josiah Hester. Protean: An energy-efficient and heterogeneous platform for adaptive and hardware-accelerated battery-free computing. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 207–221, 2022.

[6] 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.

[7] 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.

[8] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1):15–18, 2014.

[9] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Sytare: a lightweight kernel for nvram-based transiently-powered systems. *IEEE Transactions on Computers*, 68(9):1390–1403, 2018.

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

[11] Paolo Bombelli, Anand Savanth, Alberto Scarampi, Stephen JL Rowden, David H Green, Andreas Erbe, Erland Årstøl, Ivana Jevremovic, Martin Frank Hohmann-Marriott, Stefano P Trasatti, et al. Powering a microprocessor by photosynthesis. *Energy & Environmental Science*, 15(6):2529–2536, 2022.

[12] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 55–67, 2019.

[13] Michael Buettner, Richa Prasad, Alanson Sample, Daniel Yeager, Ben Greenstein, Joshua R. Smith, and David Wetherall. Rfid sensor networks with the intel wisp. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, page 393–394, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proc. OOPSLA*, pages 514–530, Amsterdam, Netherlands, 2016. ACM.

[15] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 767–781, 2018.

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

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

[18] 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.

[19] 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.

[20] Çağlar Durmaz, Kasım Sinan Yıldırım, and Geylani Kardas. Virtualizing intermittent computing. *IEEE Internet of Things Journal*, 2022.

[21] Ferhat Erata, Eren Yildiz, Arda Goknil, Kasim Sinan Yildirim, Jakub Szefer, Ruzica Piskac, and Gokcin Sezgin. Etap: Energy-aware timing analysis of intermittent programs. *ACM Transactions on Embedded Computing Systems*, 22(2):1–31, 2023.

[22] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *NSDI*, pages 439–455, 2021.

[23] 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.

[24] Robert Herbert, Hyo-Ryoung Lim, Bruno Rigo, and Woon-Hong Yeo. Fully implantable wireless batteryless vascular electronics with printed soft sensors for multiplex sensing of hemodynamics. *Science advances*, 8(19):eabm1175, 2022.

[25] 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.

[26] 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.

[27] Bashima Islam and Shahriar Nirjon. Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3), 2020.

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[29] 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.

[30] 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.

[31] Christopher Kraemer, Amy Guo, Saad Ahmed, and Josiah Hester. Battery-free makecode: Accessible programming for intermittent computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(1):1–35, 2022.

[32] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, 2017.

[33] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. Computational nanosatellite constellations: Opportunities and challenges. *GetMobile: Mobile Comp. and Comm.*, 25(1):16–23, jun 2021.

[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. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 129–144, 2018.

[36] 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*, pages 1101–1116, 2019.

[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] Gabriel Marcano and Pat Pannuto. Soil power? can microbial fuel cells power non-trivial sensors? In *Proceedings of the 1st ACM Workshop on No Power and Low Power Internet-of-Things*, LP-IoT'21, page 8–13, New York, NY, USA, 2022. Association for Computing Machinery.

[40] 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.

[41] 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.

[42] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *Proc. ASPLOS*, Newport Beach, CA, USA, 2011. ACM.

[43] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V Merrett, and Alex S Weddell. Restop: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 18(1):172, 2018.

[44] 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.

[45] Phillip Stanley-Marbell and Martin Rinard. Warp: A hardware platform for efficient multimodal sensing with adaptive approximation. *IEEE Micro*, 40(1):57–66, 2020.

[46] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/o dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019.

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

[48] Milijana Surbatovich, Brandon Lucia, and Limin Jia. Towards a formal foundation of intermittent computing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[49] 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.

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

[51] 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, 2016.

[52] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 346–359, 2021.

[53] Fan Yang, Ashok Samraj Thangarajan, Sam Michiels, Wouter Joosen, and Danny Hughes. Morphy: Software defined charge storage for the iot. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 248–260, 2021.

[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: Multithreaded event-driven intermittent computing on ultra-low-power microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

A Artifact Appendix

A.1 Abstract

EaseIO is a novel programming model that introduces re-execution semantics of IO operations for intermittent computing. Re-executed IO operations in intermittent systems' tasks might lead to memory inconsistencies and high energy consumption. EaseIO offers new interfaces to call IO functions to overcome memory inconsistencies and save energy. Programmers can call IO functions with re-execution semantics annotations thanks to the EaseIO interfaces. EaseIO consists of two main components EaseIO compiler fronted and EaseIO runtime.

A.2 Description & Requirements

A.2.1 How to access All the project source code and the instructions on how to build EaseIO and run an example are available in the following git repository: https://github.com/tinysystems/easeIO. The artifact evaluation materials used in this paper can be reached at:https://github.com/tinysystems/easeIO/commit/1136f0c6c051516f77291383a93d4e6e7ca03ce2. Also these materials are indexed by Zenodo with DOI 10.5281/zenodo.7735158.

A.2.2 Hardware dependencies EaseIO requires Texas Instruments MSP430FR series FRAM-enabled microcontrollers. Our evaluation is performed on MSP430FR5994 LaunchPad Development Kit.

A.2.3 Software dependencies EaseIO compilerrequires LLVM9.0.1.

A.3 Set-up

A.3.1 Compiler-frontend EaseIO compiler-frontend is responsible for performing semantic analysis to inject appropriate code in the original file.

You can perform transformation using the following steps and commands.

1. Run the following command to install Clang: sudo apt install clang lldb lld
2. Use the llvm-build.sh to download and compile the llvm-9.0.1: sudo llvm-build.sh
3. Afterwards, put the compiler frontend code in the /llvm/tools/clang/tools/easeIO folder.
4. Add path of this subdirectory in the cmakelist file located one folder above this one i.e. /llvm/tools/clang/tools.
5. Run make using the following commands. For subsequent make commands, you can simply call make easeIO-c:
   cd /easeIO/llvm-9.0.1-build/build
   sudo make
6. Now run easeIO-c.sh script to run the transformation for all the codes.

Note: easeIO-c.sh contains path to the source file that the user wants to parse and location of the destination folder where the output will be written. You can change the path of the LLVM folder and benchmark folders as per location on your machine. However, please make sure you have sperate folders for **Originals** and **Transformed**. EaseIO is programmed to keep these two files separate for the ease of use. So the folder for transformed codes should on the same level and path (similar to how it is in the given code structure).

A.3.2 Runtime The transformed code is then linked with the EaseIO runtime before burning on the microcontroller.

We provide the ready to run project for one of the benchmarks (FIR filter). Following are the steps to run the code. Please note that we use Code Composer Studio to run the project and tested our benchmarks on Ubuntu20.04 linux environment.

1. Select the CCSProject folder as the workspace and launch
2. Copy the transformed file from EASEIO-compiler/test to the Benchmarks folder in the project
3. Just click the debug button. Now the project is ready to go.

A.4 Evaluation workflow

A.4.1 Experiments In EaseIO-compiler/test/Transformed/ directory, there are sample benchmark applications implemented using EaseIO. The Timely_Temp_Org_transformed.c file is one of our uni-task benchmark applications which shows an example of the Timely re-execution semantic of the EaseIO. The application gets hundred temperature sensor measurements. The time constraint of this application is finishing the task within 10 msec after the sensor is read. If the power failure time interval exceeds 10 ms, then EaseIO runtime gets the temperature value again. Otherwise, EaseIO runtime skips measuring temperature and finishes the remaining part of the task.

We keep track of the application execution via LEDs on P1.0 (red) and P1.1 (green). During the whole application, you will observe that red LED is turned on. When the application is completed, the red LED turns off and the green one turns on. To intermittently run the application, INTERMITTENT macro should be defined. The LEDs run the same logic during the intermittent execution.