Timely Execution on Intermittently Powered Batteryless Sensors

Josiah Hester
Northwestern University
josiah@northwestern.edu

Kevin Storer
University of California, Irvine
storerk@uci.edu

Jacob Sorber
Clemson University
jsorber@clemson.edu

ABSTRACT

Tiny intermittently powered computers can monitor objects in hard to reach places maintenance free for decades by leaving batteries behind and surviving off energy harvested from the environment—avoiding the cost of replacing and disposing of billions or trillions of dead batteries. However, creating programs for these sensors is difficult. Energy harvesting is inconsistent, energy storage is scarce, and batteryless sensors can lose power at any point in time—causing volatile memory, execution progress, and time to reset. In response to these disruptions, developers must write unwieldy programs attempting to protect against failures, instead of focusing on sensing goals, defining tasks, and generating useful data in a timely manner. To address these shortcomings, we have designed Mayfly, a language and runtime for timely execution of sensing tasks on tiny, intermittently-powered, energy harvesting sensing devices. Mayfly is a coordination language and runtime built on top of Embedded-C that combines intermittent execution fragments to form coherent sensing schedules—maintaining forward progress, data consistency, data freshness, and data utility across multiple power failures. Mayfly makes the passing of time explicit, binding data to the time it was gathered, and keeping track of data and time through power failures. We evaluated Mayfly against state-of-the art systems, conducted a user study, and implemented multiple real world applications across application domains in inventory tracking, and wearables.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Architectures*; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering** → *Context specific languages*;

KEYWORDS

Task language, Time, Batteryless, Intermittent, Energy harvesting

ACM Reference Format:
Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of 15th ACM Conference on Embedded Networked Sensor Systems (SenSys'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3131672.3131673

SenSys'17, November 6–8, 2017, Delft, The Netherlands

ACM ISBN 978-1-4503-5459-2/17/11…$15.00
https://doi.org/10.1145/3131672.3131673

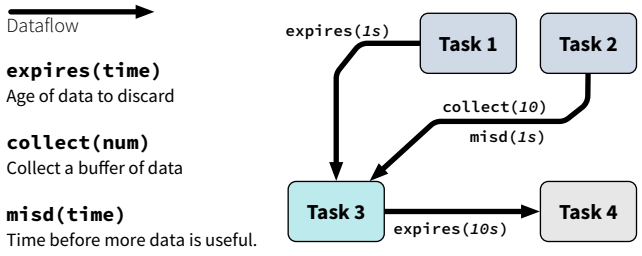

Figure 1: Mayfly programs are made up of a graph of connected tasks with clearly specified timing constraints on the data generated by each task.

1 INTRODUCTION

The vision of tiny, low cost wireless sensors that just work, maintenance free, for decades continues to elude us. This is because, nearly all sensors depend on batteries, and today's batteries all wear out after a few years, regardless of our efforts to recharge and conserve them. Batteryless energy-harvesting sensors offer hope, but irregular power supplies and meager energy storage lead to frequent power failures, and creating software that works reliably in spite of frequent failures is challenging.

An intermittently-powered sensor may fail at any time, between any two lines of code, with little warning, for unpredictable lengths of time. Time can be difficult to measure, and execution times can be difficult to predict. Programs may lose data, corrupt data, or fail to make forward progress on long-running computations. Forward progress can be preserved, if applications checkpoint their state to nonvolatile memory (*e.g.*,Flash or FRAM) before a failure [26]. With programmer defined memory fences [19] programmers can keep non-volatile data structures consistent. Breaking programs into tasks and putting global data in channels, can help lighten the developer's cognitive load [9]. Developers can even use physical hardware properties to estimate how long a device spends without power [25]; however, responding to dramatically unpredictable execution delays remains a daunting challenge for developers, in spite of these advances.

As a sensor executes over time, data age, priorities change, and opportunities come and go. Sensor data, once urgent, may only be useful for a few minutes or even seconds. When data expire after long outages, partial computations may need to be discarded and possibly restarted. After short outages, an application may pick-up where it left off. On power-up, an application's priorities may have changed. A time sensitive task may take precedence over a work-in-progress. A task that has repeatedly failed to complete (due to expired data), might be swapped out for a low-power alternative.

Each of these cases are easy to understand and easy to handle with traditional battery-powered sensors, but difficult to implement with today's languages and runtime tools on intermittently powered, batteryless sensors. Common imperative programming languages, like C, ignore time and how it relates to data. Traditionally, this has not mattered as programmers expect tasks to run quickly and sequential instructions to execute close together in time. An intermittent C program that discards expired data, schedules tasks appropriately in spite of power failures, or adapts to changing energy conditions will invariably be full of explicit time checks and cluttered with extensive branching logic.

This paper argues that 1) those who use existing languages to program intermittently-powered devices are doomed to frustration and complication, 2) those complications prevent capable programmers from creating compelling, sophisticated intermittent applications, and 3) programming for intermittent power can (and should) be simple. We call on the research community to rethink how we write programs for intermittent devices, and we specifically address the issue of time as a key to enabling sophisticated applications to intermittent batteryless sensors.

In the following sections, we describe why it is difficult to program intermittent sensors and introduce the Mayfly[1] Language and Runtime. The Mayfly *language* is a declarative, graph based programming language that enables developers to focus on application policy, sensing goals, and timing of sensing tasks while reducing the cognitive burden of intermittent programming (shown in Figure 1. The Mayfly *runtime* is a task scheduler that maintains temporal aspects of data automatically across power failures. We evaluate Mayfly against state-of-the-art systems and explore the language in the context of real-world applications in active RFID based inventory tracking, and activity recognition on wearables. Additionally, we present the results of a user study comparing the utility of our approach and the standard approach to programming batteryless sensors.

2 BATTERYLESS SENSING

The swarm of tiny devices—one day numbering in the trillions—that will make up the edge of the Internet-of-Things (IoT), *have a massive energy problem.* These tiny sensing devices have traditionally been powered by batteries. Batteries are large, heavy, expensive, and environmentally hazardous. Even rechargeable batteries wear out after a few years. The environmental and financial cost of manufacturing, maintaining, replacing or disposing of trillions of batteries every few years is not sustainable.

Batteryless sensors are a smaller, lighter, less-expensive, and more environmentally friendly alternative. These devices harvest energy from their environment opportunistically, store this free energy in tiny capacitors (orders of magnitude smaller than a battery or super capacitor), and intermittently execute tasks when energy is available. Powered by solar, RF, thermal, or kinetic energy harvested from the environment, batteryless sensors promise decades of maintenance-free data gathering at unprecedented scale in remote or impractical locations.

[1]Mayfly is named after the short-lived aquatic insect.

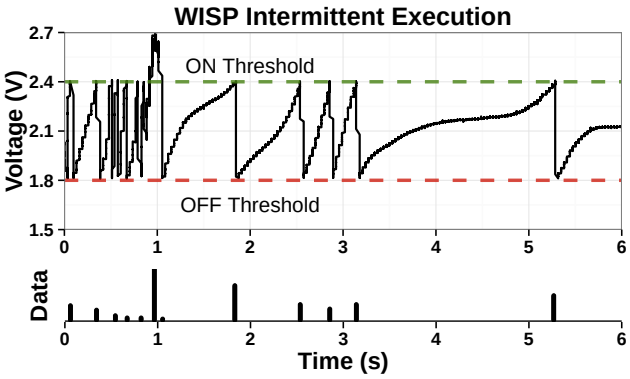

Figure 2: Voltage supply of a WISP, powered by RFID reader gathering acceleration readings across power failures for activity recognition. Data is gathered at a variable sample rate depending on the *available energy*. This volatility may negatively influence quality of applications and reduce accurate recognition.

2.1 Intermittent Operation

Despite the promise, batteryless sensors are difficult to program, debug, and deploy, because they lose power frequently and often unpredictably. Each power failure resets the device's volatile memory, stack pointer, registers, and sense of time. Once enough energy is harvested to turn back on, the sensor returns control to the start of the program (`main`). This is shown in Figure 2. This style of execution is shown (from programmers perspective) in Figure 3, where a source program is executed using both a continuous power supply and an intermittent power supply. With frequent failures and unreliable power, programming becomes a best-effort probabilistic game. As energy becomes available, developers piece together execution fragments hoping to satisfy developer constraints. This **intermittent computing model** causes developers to struggle with what are generally simple tasks—like timestamping data, ensuring that data structures remain consistent, and maintaining forward progress on long-running computations.

When a power failure is approaching, developers can preserve forward progress with checkpointing — saving some [26, 31] or all [1] of the program's state to nonvolatile memory (like Flash or FRAM). Checkpointing, as shown in Figure 3, allows programs to correctly continue where they left off after the last power failure. However, checkpointing is costly in energy and memory, as demonstrated in Chain [9]. Chain proposes that developers divide programs into discrete *tasks* that share data (kept consistent by Chain) through *channels*, eliminating checkpointing cost. Each of these methods work well in stringing together execution fragments into cohesive programs, enabling long running computation. However, each of these methods ignores the fact that an application's success or failure often depends on when tasks are executed and when data are collected, processed, and communicated. Application designers often understand these constraints (albeit imperfectly), but lack effective tools for communicating them in code.

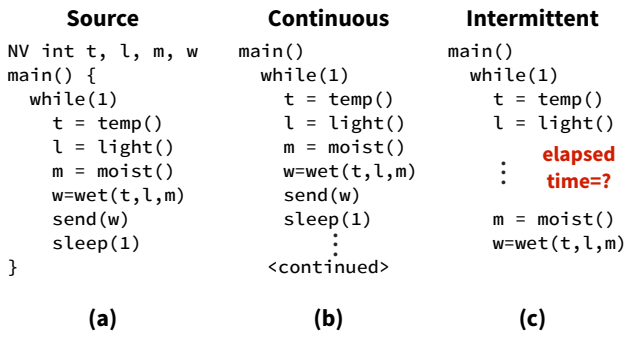

Figure 3: Execution of a greenhouse monitoring program (a). The expected continuous execution (b) is shown versus the intermittent execution (c) caused by volatile energy harvesting environments. Forward progress is preserved—however if enough time has elapsed between lines of code, it may not be worth continuing with execution as intermediate results may not be relevant.

2.2 Complexity of Timekeeping

With remanence-based timing techniques [13, 25], batteryless devices can measure time across power outages; however, *reasoning* about the impact of unexpected delays in intermittent software is a complicated and error-prone process. Nearly all sensing applications have temporal requirements, and whether we are monitoring a user's heart rate or a plant's water needs, the data we gather is often only useful when those requirements are met. A batteryless program's progress is difficult to predict, and current programming models ignore the relationship between data and time, forcing programmers to deal with the added complexity. As developers add explicit checks that consider data expirations, sensing rates, and temporal signal properties, their programs become difficult to maintain, debug, and understand. We discuss the specific issues below:

Real data often ages. If a sensor gathers data, dies, then turns back on, the time before that data is delivered (and can be used) might be seconds, minutes, or even hours. If data is used to control a process (plant watering in a greenhouse) or report in-the-moment information (like the user's current heart rate), data gathered may not be useful if they are too old. After a power outage, applications may continue processing fresh data but save energy and time by discarding incomplete results derived from old data. One task may, of course, depend on another, and preserving data freshness requires time stamp checks throughout an application's code. This is shown in Figure 4—an accelerometer based wearable activity recognition program that **samples** data, **processes** that data to identify an activity, then **sends** its results over the radio. Because of long power failures, that data has aged significantly and may or may not not be useful to the application. Either way, it is not specified explicitly in the program. Compounding the problem, the data gathered in the **sample** phase has a *mixed sampling rate with an untrustworthy clock*—untrustworthy because the device may lose power before recording the time and assigning that time to the data point, mixed sampling rate because power outages may affect the sampling rate. Using timing information to inform sensing allows batteryless sensors to use energy more effectively, sense

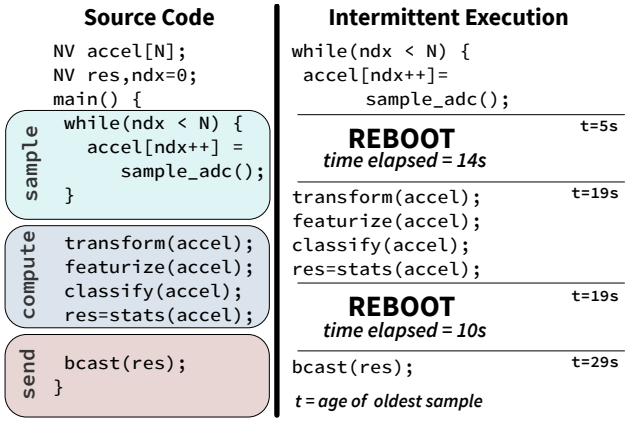

Figure 4: On left, source code of wearable activity recognition program, labeled into tasks *sample, compute, send.* On the right, a possible execution of the activity recognition shows how data ages through power failures. If the system that uses this data makes data-driven decisions every 10s, the 29 second old data may not be worth transmitting. Sensing data becomes mixed sample rate, and clocks become unreliable because of intermittent execution and variable length outages.

only when they need to, and not transmit or process old data. However, how this timing information is used is currently application- and developer-dependent because timing information is implicit in current programming models.

Real applications often tolerate temporal variation. A pedometer's step-counting algorithm may call for 30 accelerometer samples collected at 10 Hz. Depending on how it detects steps, that same algorithm may give accurate results as long as the 30 readings fall within a 4 s window and as long as no two readings are taken within 80 ms of each other. These relaxed requirements will be much easier to satisfy with frequently-failing hardware, but more complicated for application designers to implement. Meeting stringent timing requirements is difficult in batteryless sensors that exist in unpredictable energy situations.

Intermittent data complicates programs. When devices are tethered or battery-powered, power supplies are stable and power failures are rare—data collection and management is difficult, but straightforward. When devices operate intermittently, programmers must add code to track when data are generated, when data have expired, and when it is advantageous to gather more data. Developers can use remanence timekeeping methods [13, 25] to timestamp data; but writing code that knows when to discard the whole, or part of a buffer of data, and that properly timestamps data and checkpoints execution is tedious. Tedious programming tasks lead to sloppy implementations, which leads to bugs in deployment. Short sensing programs that perform simple sensing tasks quickly explode with timing and consistency checks, recovery methods, data collection, and age management heuristics, making implementation painful and applications difficult to debug, maintain, and verify.

Languages assume continuity. When using current programming models, programmers assume that sequential instructions will execute one after another with effectively zero time between them. With intermittent execution, the actual time between two instructions (or tasks, for languages like Chain) could be microseconds, seconds, minutes, hours, or even days because of unpredictable placement of power failures. If the time between failure, and resuming execution (the time between the device losing power, checkpointing, and then resuming once power is restored) is *almost zero*, then most likely continuing from where execution left off is best. However, when longer delays occur, intermediate execution and results might not be useful going forward. The programmer's **intent** is not explicit—as regards time—with current programming models, so the **runtime** has to make assumptions. Current languages ignore the timeliness property of data caused by intermittent execution, leaving the developer with few ways to strongly and safely express time constraints of data.

A New Approach: Due to the limitations mentioned in this section, we and others have struggled to build interesting batteryless applications. In answer, the next section describes Mayfly, a task based language that does away with checkpointing and integrates time management as a first order concern, designed specifically to help designers create sophisticated batteryless sensing applications. With Mayfly, developers can coordinate time-sensitive data and tasks without getting lost in checkpoints, memory fences, and timekeeping. Mayfly automatically takes care of common issues like data expiration and time sensitive data collection. Additionally, Mayfly handles computational dependencies and ensures they are co-located in time despite interruption by power failures.

3 MAYFLY LANGUAGE

We developed the Mayfly language and runtime to enable developers to easily reason about the relationship between time and sensor data in intermittently-powered, batteryless sensing applications. Mayfly is designed to simplify or eliminate the time management and intermittent programming challenges described in Section 2 in three key ways:

(1) **Simplify Time Management:** Developers should not concern themselves with recording and managing timestamps and dealing with timing uncertainty at a low level. In Mayfly programs, the developer can focus on the high level application sensing goals instead of low level, error prone, timekeeping drivers.
(2) **Enable Dataflow Control:** Gathering, computing, and transmitting accurate and timely data is a core function of all sensors. Despite intermittent execution, Mayfly must provide the ability to coordinate dataflow in spite of power failures and unpredictable energy environments.
(3) **Provide a Usable Programming Model:** Language and tools are useless if no one can use them. Mayfly focuses on (and evaluates) providing a high level of usability, reducing distractions and enhancing developer understanding of program execution and purpose.

The Mayfly *language* simplifies the development of batteryless sensing applications—enabling programmers to confidently write intermittent programs. The Mayfly *runtime* efficiently schedules programmer defined tasks in an unstable energy environment, carefully timekeeping, managing relevant data and checkpointing progress. Together, the Mayfly language and runtime support emerging batteryless applications for the Internet-of-Things and beyond.

Listing 1: A Mayfly program for greenhouse monitoring

```
// Optional global policies
{: scheduling_policy(FINISH_FLOW); :}

// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int light, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge constraints
light -> wet {expires(10s)}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10)}
```

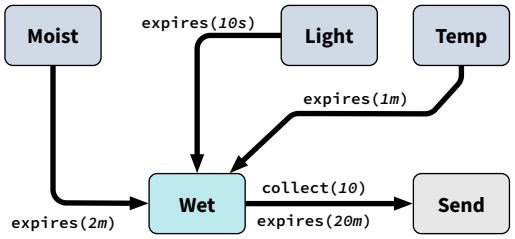

Figure 5: This figure shows the three constructs Mayfly needs in a program; task definitions, flow definitions, and constraints. Also shown are optional policy information.

3.1 Language Overview

A Mayfly program is a directed data-flow graph; where nodes are tasks and edges define the flow of data and their associated temporal constraints. *Tasks* are connected by *Flows*, with data flowing through the graph as tasks execute. Edges are annotated with *Constraints* that describe how to treat data as it flows through the task graph. Each set of connected tasks is a *Task Graph*. The Task Graph is executed by the runtime opportunistically, one task at a time, from the top-most tasks (the tasks with no inputs) to the bottom-most tasks (the tasks with no outputs). Representing a program as a task graph allows developers to quickly and clearly see the structure and purpose of an intermittent program without having to worry about the effects of intermittent execution. From these logical constructs each program is composed of three mandatory syntactic constructs, 1) task definitions, 2) flows, and 3) constraints, and optional policy information, all detailed below and shown in Figure 5 and Listing 1.

Listing 2: Mayfly syntax examples

```
// (1) Task definition
task_name (TYPE input, ...) -> (TYPE output, ...)

// (2) Flow for activity recognition
sample -> compute -> send

// (3) Predicate
// compute -> (int error, int activity)
sample -> compute[_,RUN] -> send
compute[_,WALK] -> log

// (4) Program policies
{: scheduling_policy(FINISH_FLOW); :}
```

3.2 Tasks and Flows

Tasks encapsulate a single purpose; they are a logical grouping of Embedded-C code that accomplishes a single objective—for example, collecting temperature readings, processing a buffer of data, or interfacing with a radio to send a packet. These tasks correspond to C functions written by the programmer (possibly taken from an existing code base), and can involve computation, use of external sensors, or communication. A task's inputs are defined by its incoming edges, and a task's outputs are defined by its outgoing edges.

Tasks are treated as atomic units of computation, and must be executed without being interrupted by a power failure. If a tasks execution is interrupted, the runtime will rollback any intermediate results, and try to execute the task again at the next available opportunity. After a task completes, its results are stored in non-volatile memory, where they become available as inputs to other tasks. In this way, Mayfly guarantees that forward progress will be preserved, with the caveat that forward progress is only useful *as long as timing requirements on data (specified by the programmer) are met.* Figure 4 shows how a wearable activity recognition program (in C) could be divided into the tasks `sample, compute` and `send`. These tasks could be further subdivided if developers expect low energy availability in deployment causing many interruptions. For instance `sample`, could be broken into `transform`, `featurize`, `classify`, and `stats`. Tasks are specified in a Mayfly program using the syntax shown in the first example of Listing 2. These task definitions precisely define the data types and sizes each task expects as input, and the data the task generates.

Flows give structure to the task graph, defining the relationship between tasks. Developers connect the tasks to each other, giving explicit dependency information to the runtime. This also has the effect of making program execution explicit, despite the frequent power failures batteryless devices go through. This is shown in the second example of Listing 2. Flows enable conditional execution between tasks, through the use of predicates between tasks. With predicates, a programmer can adjust program behavior in response to intermediate results, environmental input, or even user actions. The third example of Listing 2 shows how a predicate flow can be used to divert information to either of two actions in a wearable activity recognition program. The `compute` task has two outputs, a measure of the classification `error`, and the identifier of the `activity` recognized (in this example, `WALK` or `RUN`). The predicate states that if the compute task generates a `RUN` output, then route data to the `send` task for broadcasting the information to a basestation or the users phone. Otherwise route to the `log` task, to record the data locally.

Policy information (example four in Listing 2) for the program is optionally given at the top of the program. Policy settings allow developers to specify global attributes of the program itself, select runtime heuristics, and pass on hints to the runtime about what the developer expects of the environment and the application. This policy could define the scheduling method to use, choosing between a scheduler that prioritizes moving new data through a graph, or one that focuses on finishing the processing of older data through the graph. Policy can also change which metric to use for determining priority, or which flow to prioritize.

Tasks, and Flows (with some help from policy) guarantee that developers will make forward progress in a program (assuming energy is available to complete individual tasks). Additionally this structure lets developers quickly and easily understand the logical execution and purpose of a Mayfly program, while providing a structure which can be annotated to explicitly define timing related constraints for data generation and processing.

3.3 Timely Data Constraints

Constraints describe how data is treated as it flows through program tasks (the Task Graph). Constraints acknowledge that not all data is equal, and in fact the value often depends on the time the data was gathered, or how much repetitive data is available in the same timespan. Constraints are provided by annotating the edges between tasks with three possible constraints: *expires*, *minimum-inter-sample-delay (MISD)*, and *collect*. By using these three constraints, developers can specify what valued data means to their final application.

`Expires` tells the runtime how long data can sit on an edge before it loses value to the application. In programs with a stable power, processing is usually predictable, and developers take appropriate design-time measures to ensure that data is delivered while it is fresh, instead of explicitly checking timestamps.

In intermittent devices, as start-to-end processing times vary more dramatically, timestamp checking becomes both essential to efficient operation and tedious, requiring time checks at least after every reboot. In checkpointing systems, this timestamp gathering is also prone to failure as it is easily interrupted, meaning that data may appear to be timely, but in fact has an errant timestamp.

`MISD`, or minimum-inter-sample-delay throttles the data rate so that the runtime does not generate more data than needed by the application. When power is reliable, developers use timers to regulate sampling rates—an approach ill suited to intermittent operation (reboots reset timers). Instead, Mayfly developers explicitly tell the runtime how long after generation before more data is useful. MISD is a runtime hint from the developer that makes data collection more efficient. MISD acknowledges that gathering new data that has been specified by the developer as useless, is counterproductive and wastes energy. The best thing to do is wait (put the processor

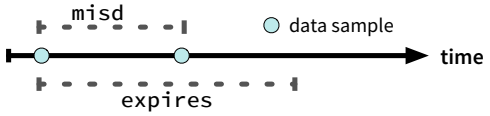

Figure 6: This shows how the constraints `expires` and `minimum-inter-sample-delay` (or `misd`) work together, letting the runtime know exactly which data is most valuable to gather and when.

in sleep mode) till new data is useful again (according to the developer), or enough energy has been gathered to execute a different task. This mechanism is shown alongside `expires` in Figure 6.

`Collect` takes the tediousness of gathering a set of data off the developer, and onto the runtime. Rather than look at a single data point, many sensing applications collect multiple data and perform operations on them. With reliably-powered sensors, getting a buffer of data for processing, that is all fresh, and was gathered at a *consistent* sample rate is trivial. When execution becomes fragmented creating these buffers gets difficult, as now the programmer must replace expired samples in addition to all other sensing tasks. Memory is limited. So, buffers should not contain redundant data (data that violates a `MISD` constraint) or data points gathered too frequently or too sparsely. Both scenarios provide data of little value to the application. The `collect` constraint allows developers to simply gather useful windows of data, coming out of a task. The `collect` constraint is compatible with the `expires` and `MISD` constraints, so that developers can specify the spacing and expiration of data in a `collect` buffer. Collect also serves as a replacement for a traditional loop construct. Loops can be embedded in the task code, alternatively, the loop body can be broken up into tasks, and then represented as a task graph that will be continually executed based on its priority, using a `collect` constraint going into the sink node of the task graph.

These three constrains provide the core ways developers interact with sensing data inside their programs. These constraints provide a way for developers to declare what data is most valuable in a structured way, that masks the intermittent operation of batteryless sensors. We anticipate there will be more constraints added (or current constraints extended) as the language develops and new needs arise.

3.4 Ancillary Language Details

3.4.1 Multiple Task Graphs. Mayfly programs can be made up of multiple task graphs, each of which comprises multiple dependent tasks. These can all be defined in the same program. With multiple task graphs, developers can describe different sequences of actions to take during operation, under differing parameters. Each task graph has an implicit **priority** assigned to it, based on the ordering of the task graphs flows in the input file containing the source code. This means that the highest priority task graph will be checked first for any possible tasks to execute, if none are found, the next highest priority task graph is checked. Low priority tasks are only executed if there is nothing else to execute: Mayfly greedily execute tasks based on priority. In some cases, this could potentially lead to task starvation. To ensure that low priority task graphs are executed, developers need to apply `misd` constraints to the highest priority task graph, to leave time and energy for the other tasks graphs to execute.

3.4.2 Memory Model. Mayfly prescribes a task-local memory model (similar to Chain[9]) Mayfly programs have no global memory that is accessible, or writable, from individual tasks. Tasks are only allowed to use volatile memory (local variables) internal to the task itself, as well as the read only inputs, which are explicitly defined by the program. Tasks generate output, which is accessible on the edges only once the task has completed. Since tasks cannot alter system or non-volatile memory, tasks will avoid consistency issues associated with mixed memory volatility systems. This task-local memory model also simplifies the programmer interface, as programmers only need concern themselves with the input and outputs of the task.

3.4.3 Hardware Interactions. The task-local memory model removes the possibility of memory inconsistency for the computational device (usually a microcontroller). However the memory and initialization state of the connected hardware peripherals such as sensors, radios, and storage devices, can cause problems. For example, any interactions a task has with external components will change those components in a non-deterministic way from the perspective of the next task to execute. To mitigate this issue, Mayfly developers must write tasks that build up, and break down hardware state, or simply reset hardware peripherals before use to reduce consistency errors.

4 IMPLEMENTATION

This section describes the Mayfly Runtime and implementation. We also describe the software and hardware used to evaluate, and deploy Mayfly, as well as applications developed with Mayfly. All software and hardware designs are freely available at our website.

4.1 Code Generation

Writing a new language from scratch is unnecessary, and potentially hurts adoption by the community. The Mayfly language is instead a coordination language built on Embedded-C, meaning developers can reuse common libraries and functions.

The Mayfly compiler goes through multiple phases to create a device specific firmware from a Mayfly program and user libraries. The architecture of the compiler is shown in Figure 7. The compiler is written in Java, the Java CUP library is used as a parser generator, with JFlex as scanner generator. A templating [17] is used to construct the Mayfly runtime instance for each program. Using templates makes porting to other processors and platforms easier.

First, Mayfly program code is parsed, and checked for syntax errors. The compiler checks that inputs of tasks match edge data, and that constraints, tasks, and policies are all defined. After validation, graphs of tasks are constructed. Next, task graphs are annotated with the constraints and policies given by the developer, and graph structure is checked that no cycles exist. These constraints are analyzed for logic errors or potential problems: for example, if the expiration of a source node is too short, or if an edge is trying to gather a prohibitive amount of data for the time period.. Once

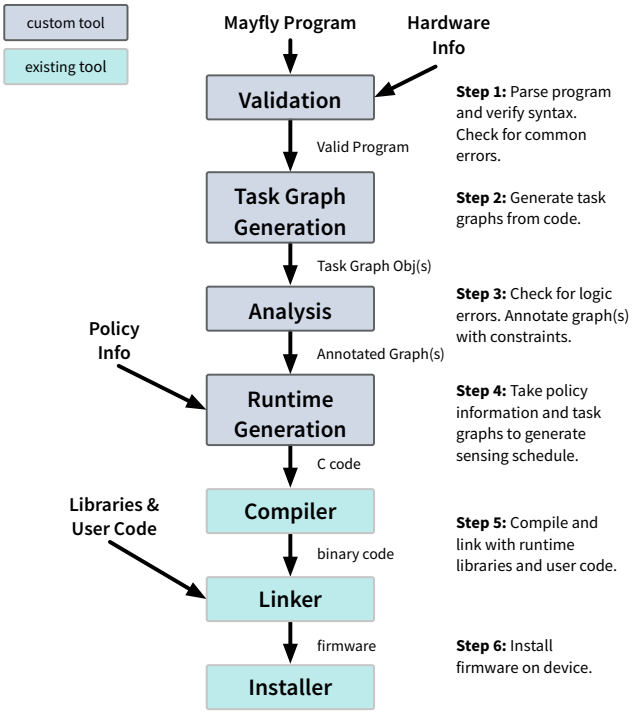

Figure 7: Architecture of the Mayfly compiler, showing the steps in producing the firmware image for a given Mayfly enabled, batteryless sensing device.

task graphs are validated, annotated, and analyzed, code generation can begin. Task and edge data structures are created from the Task Graph(s), and written to the runtime templates. This approach allows for flexibility in scheduling algorithms. At the end of this stage, a complete Embedded-C program is generated. Finally, the Embedded-C runtime is compiled, along with user code, hardware headers, and runtime libraries. This is all linked into a binary, and installed onto the batteryless sensing device.

4.2 Mayfly Runtime

The Mayfly Runtime is generated from the developers program and the language specification by the Mayfly compiler. The generated Embedded-C runtime is a statically defined schedule of tasks, with energy management, timekeeping, and checkpointing built-in. The schedule has to be static because of the extremely constrained resources of these devices. Energy is limited, so any energy used to execute runtime functions is taking away from potential user and sensing tasks. The Mayfly Runtime keeps track of three things through power failures: 1) local time, 2) the execution state as in the progress thought the task graph(s), and 3) the data in each edge of each task graph. Using these three things, Mayfly's Runtime can execute tasks across power failures, while respecting temporal properties of sensor data. The runtime relies on hardware support for timekeeping, byte addressable NVRAM technology (FRAM) for checkpointing, coupled with software techniques to persist tasks through failures.

Listing 3: Mayfly runtime operation pseudocode

```
1  main() {
2   if(timekeeper_reset())
3    rollback_full()  // Rollback all data
4
5   if(!state.atomic_lock)
6    rollback_one()  // Rollback last task
7
8   time = get_time() // from RTC
9   while(1)
10   t = next_task()
11   if(constraints_satisfied(t))
12     state.atomic_lock = FALSE
13     execute(t, input, output)
14     state.atomic_lock = TRUE
15     // User task ran
16     // Changes committed to task graph
17
18   if(t==NULL)
19     reset(t)
20     sleep() // Nothing to do
21 }
```

Runtime Operation: Pseudocode for the runtime operation is shown in Listing 3. After rebooting from a power failure (Line 1), the Mayfly runtime does three things (Lines 2-7), get the time, check if the external remanence timekeeper was reset, and check if the last task was completed. First, the Mayfly runtime updates the *local* system time using the Remanence Timekeeper[13] which is external to the microcontroller, on the same printed circuit board. This is the **only** time the external timekeeper is polled. This timekeeper is an external capacitor or real-time-clock (RTC) with its own dedicated energy store—a 10 µF capacitor. The timekeeper draws orders of magnitude less current than the microcontroller (MCU) while maintaining the clock, drawing less than 20 nA. This timekeeper might have reset if the time between a power outage and reboot was too long, in which case the runtime will rollback all time sensitive data as now it has no guarantees on the age of any previous data collected. This is preferable to continuing to process on useless (according to the developer constraints) data. The final check the runtime performs before executing tasks is determining if the last task executed was able to complete, and set the lock. If the lock is not set, then the tasks outgoing edges are rolled back, and the tasks is available to re-execute.

After successful recovery from a power failure, Mayfly begins looking for something to do. Tasks are checked in priority order, held in a static list defined at compile time. Task constraints are checked and if the constraints are satisfied, the task is executed. These constraints include all those listed in Section 4.2. Before the task is executed, the lock is released, and the output values from the incoming edges are placed into the task. Pointers to a temporary output buffer are also put into the task to receive any generated data. After a task is executed, the changes are committed to the task graph state, if the task is interrupted or the commit is interrupted, then the task is re executed when it becomes available again. In this way, program state will not be corrupted. This continues until either a power failure or a low voltage scenario, where nothing can be done, at which point the runtime puts the device to sleep

until more energy becomes available (or the device dies), or a task becomes ready to be executed.

Data Management: Mayfly keeps all edge data in FRAM. FRAM is a low power non-volatile memory (NVRAM) with write speeds that allow it to be treated as RAM, enabling very fast checkpoints. Edge data is double buffered, so that old data is not overwritten by new data until the lock is set. This ensures that at any point, the execution can rollback to the previous task safely. Each edge also stores timestamps for each piece of data, array information for the `collect` constraint, and the last time the task was used to generate data for servicing the `misd` constraint. The runtime checks data against all constraints defined on the edge to keep `collect` buffers valid, only adding to the buffer if the data point is unexpired (according to the developer constraints), and the last sample is older than the developer defined `misd`. If no expiration or `misd` is specified, the data is considered always useful, always fresh, and will be gathered as fast as possible and never removed from the buffer.

Timekeeping: The runtime keeps track of time across power failures by using Remanence Timekeepers[13]. When energy runs out, the microcontroller, volatile RAM, and all clocks are reset. This means that any previous timestamps must be updated when power is regained. On each power-up, the runtime reads the timekeeper using either an ADC or the SPI bus (depending on the platform). After a read, the runtime charges a dedicated external small capacitor that maintains the Remanence Timekeeper. The key insight is that the remanence timekeeper will draw an order of magnitude less power than the the main sensing platform, and the draw is not dependent on sensor behavior. This means that it can maintain a granular sense of time throughout power failures.

4.3 Applications

We implemented two complete applications with Mayfly. These applications demonstrate real problems that batteryless, energy harvesting systems can solve. Each application has time sensitive data generated or transformed by tasks that vary in their time and energy requirements. Each application encompasses multiple task nodes, and uses multiple Mayfly constraints to specify the program. These applications are variants of those presented in Chain [9], developed for the WISP [28] Computational RFID platform. We rewrite these applications using Mayfly, with timing information specified from our own observations.

Cold-Chain Equipment Monitoring (CEM): CEM systems continuously monitor temperature controlled environments, such as vaccine and biological sample storage. Temperature logging also has application in smart home technology and HVAC monitoring for commercial and industrial buildings. These logs could be read off the device at a later time using an RFID reader, physical access, or by broadcasting to a basestation. The CEM system implemented in Mayfly can safely assume that temperature will not change rapidly, meaning that the data generation rate can be throttled using the MISD constraint. Since the temperature is being logged, each data value has no expiration, but is tagged with a timestamp that persists through reboots.

Exercise Recognition: The health and wellness of a large aging population is a major concern in the USA. Exercises for elderly people, especially overweight elderly people, are often prescribed by doctors. Doctors often prescribe hip exercises for elderly patients, including the sit to stand exercise, which helps prevent disability. Providing doctors with information on exercise completion would help treatment. Using a wrist-worn, batteryless wearable device equipped with an accelerometer is one way that these exercises could be tracked. By discarding the batteries, the wearable is easier to wear, and does not have to be taken off to charge, which presents opportunities for losing the device. Activity Recognition (AR) can use the on-board accelerometer to determine sitting and standing states to tally exercise completion based on prior training. AR samples a sliding window, filters out noisy values, then extracts features and classifies as standing up or sitting down. Mayfly can take advantage of programmer timing knowledge to discern when old accelerometer data has expired, and is therefore not worth processing, or if it is too early after a standing or sitting action to gather new data (since it is physically impossible to sit or stand in a few milliseconds). It is assumed that energy is available such that power failures are frequent, but the failures themselves are short, as is the case in environments where RFID readers are deployed throughout a space.

Activity recognition with intermittent devices has been undertaken successfully [6, 27], and provides an interesting application space for energy harvesting. Mayfly allows activity recognition applications to handle the lossy data problem caused by intermittent execution by providing accurate timekeeping. Mayfly does not use old acceleration data to classify activities, this means that elderly patients will not be penalized if the application missed activities, and that false positives are less likely to occur. Multiple batteryless devices could also be used to reduce data loss, at lower cost than a single battery powered system, potentially enabling a richer dataset.

5 EVALUATION

We evaluate Mayfly by examining the benefits of timekeeping when faced with intermittent power, in comparison to untimely languages. We make comparisons to other intermittent languages in terms of memory overhead, data utility, and usability, for a variety of real world applications outlined in Section 4. We introduce our experimental setup and metrics in Section 5.1, then outline the results of our experiments. We present execution overheads of the scheduler, as well as initialization costs of the runtime in Section 5.5. Finally, we investigate the usability of Mayfly and traditional Embedded-C for programming intermittent devices in a user study in Section 5.6.

5.1 Experimental Setup

Designing experiments for runtime systems for intermittent devices must be done with consideration of energy harvesting environment, leakage, measurement overhead, and available platforms. Because of low energy storage, measurement techniques must be non-invasive and energy free. In this section we describe the experimental design we use to compare each state-of-the-art runtime system with Mayfly.

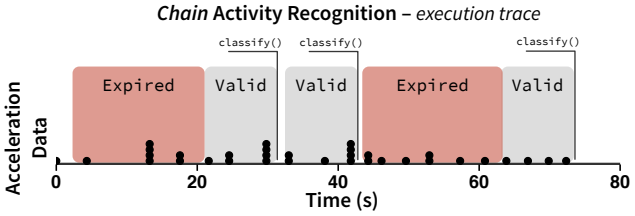

Figure 8: This shows the times at which the Chain app gathers data during its execution. Once it gathers enough data, it `classifies` that data into an activity. The taller the stack of black dots, the more data gathered at that time. With Chain, expired acceleration samples (blocked in red) are incorrectly used to `classify` activities.

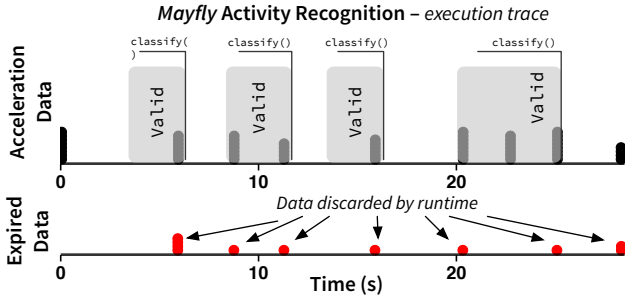

Figure 9: This shows the times at which Mayfly gathers data (gray dot stacks) and discards old data (red dot stacks) for the activity recognition app. This data replacement is specified by the developer flow constraints. With Mayfly, only valid data is used to `classify` activities.

Test Devices: The WISP [28] and Moo [32] are, to our knowledge, the only hardware platforms available for batteryless sensing. For evaluating Mayfly, we use a WISP running at 8MHz, with 64KB of main memory (FRAM) and 2KB of RAM (SRAM), augmented with a custom printed circuit board (PCB) that attaches to the connector, providing an RTC as remanence timekeeper which can time more than 17 minutes of failure. We note this is longer than in previous work [13] because we used a larger capacitor and larger trickle resistor.

Runtime Systems: We compare implementations programs in Mayfly to implementations of the same or similar applications for DINO[20], and Chain[9]. The Chain artifact provided implementations of the CEM and activity recognition (exercise recognition) application in DINO, and Chain. In our experiments, we compare results for each application on every runtime system.

Measurement Setup: We used a number of tools to gather application success metrics, sensor data, and execution statistics without interfering with the execution of the devices under test. A Saleae logic analyzer and an Energy-Interference-Free-Debugger (EDB) [8] were used to record application success metrics by snooping a communication bus on the test platforms. Python scripts were used to coordinate the data collection, and start or stop test runs. An Impinj RF2500 Speedway RFID Reader was used as the energy source.

5.2 Data Utility

Mayfly takes advantage of developer application knowledge and insights to deliver the same or better quality of service and data utility while doing less work. The exercise recognition application described in Section 4 was run multiple times on the RFID reader, with the Chain and Mayfly intermittent runtime systems. The WISP that the exercise recognition program was running on was placed 25 cm away from the mini guardrail antenna connected to the Impinj RFID reader. At that distance, harvestable energy is scarce and longer outages on the order of seconds are frequent. Figure 8 shows a representative trace of the *Chain* Exercise recognition app running through this experiment. Figure 9 shows a representative trace of the *Mayfly* Exercise recognition app running through this experiment. These figures show the relative amount of data gathered at specific times, taller stacks of black dots mean more data was gathered.

The Chain app gathers a predetermined number of samples which are then used to classify the current activity. In Figure 8, samples are represented by stacks of black dots, with the red highlight showing expired samples. Figure 8, shows that Chain (and any other untimely runtime) will use expired data to classify an activity, giving a potentially incorrect classification and wasting cycles and energy. The bottom part of Figure 9, shows times when Mayfly discards old data (red dot stacks) using its external timekeeper; only using unexpired data for activity classification. These figures show that not all data are equal in determining the Quality of Service (QoS) of an application. These figures show that preserving forward progress without regard for elapsed time of power failures can hurt the quality of service. Throughput does not linearly relate to QoS, which means that throughput can be traded off for energy without reducing quality of service for applications that have temporal data constraints.

5.3 Memory Usage

Memory usage is important in embedded devices in general, and batteryless sensors especially. These ultra constrained devices can't hold much data (the MSP430FR5969 on the WISP has 64KB of FRAM, 2KB of SRAM) so must be intelligent in their data management. We characterized the amount of memory used by each application implemented on each runtime. The memory usage of each application and its runtime implementation is shown in Table 1. Mayfly benefits from the absence of checkpointing, since developers specify which data is important, Mayfly only needs to store those data, not the entire stack. A mirrored memory space is not required, meaning that Mayfly will always have comparable or better memory usage to the Mementos and DINO approach. None of the approaches use a significant amount of memory in relation to the total memory on the WISP device. Mayfly benefits beyond other approaches in some cases since timing constraints control how much data is gathered and stored. Mayfly can avoid storing excess data (that have expired or that don't benefit the application) based on user-defined constraints like `expires` and `MISD`.

A significant portion of the memory footprint of the Mayfly runtime is from libraries and platform initialization routines. Table 2

Table 1: Non-volatile memory usage (KB) of each app and system.

App.	Mayfly	Chain	DINO
ER	2.9KB	2.5KB	4.2KB
CEM	3.1KB	4.1KB	5.8KB

Table 2: Mayfly memory breakdown per app in bytes.

App.	Task data	Scheduler	Constraints
ER	406 B	884 B	476 B
CEM	270 B	1128 B	636 B

shows the breakdown of Mayfly runtime memory. Most of the memory footprint comes from the generated scheduler. Constraints, and task data are directly created from user specified constructs. The scheduler trades a higher memory footprint for a more efficient runtime. We anticipate that further improvements to the scheduler will reduce the memory footprint.

5.4 Developer Effort and Usability

For many applications, Mayfly is easier to program with than other systems, because of the (1) reduced number of language constructs that must be hand coded, and (2) the top-down, simple to visualize development approach. Table 3 shows the difference in lines of code required to develop an app in Chain and Mayfly. Mayfly was specifically designed to make the job of writing time aware applications for intermittently powered sensors easier. As we demonstrate in our user study (Section 5.6), developers have a hard time with understanding intermittent programming when using Embedded-C, even with a reliable, external timekeeper. Mayfly programs are designed top-down. Developers define the input and outputs of tasks, then make connections between tasks, then finally assign constraints to the tasks or edges. This is assisted by a visualization tool that shows a graphical representation of the Mayfly program on compilation that looks similar to Figure 5. Once the Mayfly program has been successfully created, developers only need to include a separate source file with the task definitions implemented. These function definitions can be ported from existing code, use existing libraries, and are written in Embedded-C like every other runtime system we evaluated.

Mayfly has the advantage over other runtime systems by separating the global goals (the Mayfly program defining task graphs) from the actual implementation of tasks in Embedded-C. Other systems join the two; Chain for example, requires programmers to explicitly specify memory channels and then specify control flow inside the task definitions themselves, using new C language constructs. While this approach can greatly reduce the memory footprint, it obscures control flow, requiring users to search through a program to find where tasks lead to. This is mainly because of the approach; Chain is designed as a C library, Mayfly is a compiler, allowing for much greater freedom and flexibility in the input language, and compiled output. Compiling gives greater control, allowing Mayfly to generate intelligible error messages, and capture the most common errors in the validation stages.

Table 3: LOC for language constructs in Mayfly and Chain.

	Mayfly			Chain			
App	*tasks*	*flow*	*constraints*	*tasks*	*ch*	*flow*	*decl*
ER	5	1	13	11	49	19	61
CEM	9	3	17	12	63	19	82

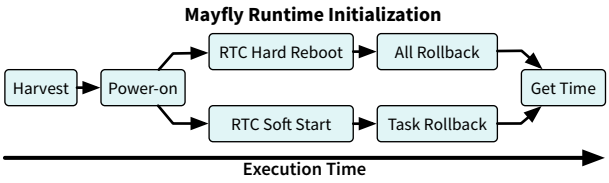

Figure 10: Runtime initialization flowchart. Each function has specific timing overhead. If a hard reboot happens where the Real-Time-Clock (RTC) is reset because of a long power failure, initialization cost increases.

Table 4: Initialization and scheduler runtime costs.

Init function	Time
Power-on from brown-out	1.0 ms
RTC hard reboot	708.5 ms
RTC soft start	144.6 µs
Rollback all Data	23.6 µs
Single task rollback	4.6 µs
Get time (seconds, minutes, hours)	246.4 µs
Get time (seconds)	80.6 µs
Scheduler function	
Process constraints for single task	4.5 µs
Scheduler cant find task to execute	56.3 µs
Task finished, commit results	7.0 µs

5.5 Overhead

Certain implementation details are pertinent to the overall evaluation. We detail their effect on performance, note potential areas of improvement, and provide practical implementation costs in this section. Specifically we look at the runtime initialization costs (above the execution costs of user defined tasks), and the scheduling costs for determining which user defined task to run. The platform initialization scheme is shown in Figure 10, along with execution costs in Table 4.

In Figure 10, two paths are shown for initialization, the hard reset path, and the soft reset path. The soft reset path ("RTC Soft Start" in figure) simply polls the RTC, rolls back the last task if it was not completed and then gets the time. This happens fairly quickly. However, when the external timekeeper loses power completely (in addition to the MCU), the timekeeper resets its clock back to zero on next boot. This reboot ("RTC Hard Reboot" in figure) takes longer to initialize the timekeeper properly, than if it had not expired. Additionally, all timestamps ("All Rollback" in figure) must be set back to zero for each task output. In our current implementation for the MSP430FR5969 this hard reboot could take up to one second, but on average takes 0.71 seconds. This length of time is

significant, and difficult to overcome with current RTC components available off-the-shelf, however, in our experiments the timer rarely loses power completely, as we used a conservatively sized storage capacitor of 10 µF for the timekeeper, which can time more than 17 minutes of of failure. This 17 minute timing ability is more than adequate for our applications, however, for applications with longer timing requirements, new advances in zero power timekeeping are required.

Table 4 shows the costs of the scheduler functions. Before a task can be executed, its constraints must be satisfied, this check happens many times during the scheduling cycle and must be very fast. On average, this check happens in 4.5 µs, quick enough to not be a burden, and making the cost of not finding anything to do low. The other important function is committing data from a task to non-volatile memory so it can be preserved through a power failure. This function only takes 7.0 µs.

Energy and Cost: To use Mayfly with the WISP, it must be augmented with the custom PCB. The energy cost of maintaining the timekeeper, the price increase per unit, and the firmware are all overhead items for Mayfly. The initial charging of the timekeeper requires 28.8 µJ, then a constant trickle current of 54 nA. Adding the custom PCB, or designing a new PCB with required hardware only increases the price point by $1.05 per unit. Additionally, the memory overhead of supporting the hardware timekeeper is constrained to a small library that requires 1364 bytes.

5.6 User Study

We evaluated the usability of Mayfly on eleven participant drawn from a junior-level, university Computer Operating Systems course[2] Our findings suggest 1) Mayfly reduces the time needed to write intermittent programs, 2) Mayfly helps developers reason about intermittent behavior, and 3) the benefit of using Mayfly is high enough for encourage C developers to migrate.

Methodology: Participants were provided documentation describing methods for writing intermittent programs in embedded-C, relevant syntax, and function definitions provided by the research team for sensing and timekeeping. They were given 20 minutes to familiarize themselves, prior to the experiment. Participants were then provided three unique programming challenges, with 20 minutes to complete each. The research team manually compiled participants' code, and reported errors in syntax and timing related bugs. Participants repeated the above process for the same challenges, but this time using the Mayfly language. Participants received documentation describing methods for writing intermittent programs in Mayfly, relevant syntax, and function definitions, as above. Participants used Mayfly to complete the same three programming challenges, under the same conditions. These programming challenges asked participants to 1) sample a sensor and send only data that is less than one minute old, 2) sample two sensors, ensuring the first is not sampled more than once per minute, and the second is sampled no more than once every five minutes, and 3) sample a single sensor ten times in one minute, but no more than once every two seconds, and send an average that is calculated using only data that is no older than one minute. These challenges were designed to model the three key Mayfly timing constraints, `expires`, `misd`, and `collect`, to illustrate the difference in handling these scenarios in C and Mayfly. After each set of three challenges, participants completed a survey rating the ease of the language used.

Sample: Our eleven participants ranged in class standing from junior to senior, with three to seven years of formal computing education, and three to eight years of total programming experience. Participants self-rated their overall programming abilities, comfort using C, and knowledge of computer architecture, as compared to other students, and average application developers. Overall, our participants reported their abilities were average, but slightly above other students.

Results: Mayfly reduced the time needed by our participants to write intermittent programs. Participants using Mayfly unsuccessfully compiled an average of 0.64 fewer times per task, than they did when using C. On average, participants successfully completed 1.54 more challenges within the set time limit, in Mayfly than in C. Mayfly made overcoming the complexity of intermittent programs easier, for our sample. None of our participants successfully completed our `expires` coding challenge in C, while all eleven completed the challenge in Mayfly. Five participants completed our `misd` challenge in C, while seven did so in Mayfly. Four participants successfully completed our `collect` challenge in C, while eight did so in Mayfly.

Mayfly was reported to be more usable than C, to accomplish these tasks. Using a series of 5-point Likert-type scales, participants rated the usability of each language. The items in this survey were verified using confirmatory factor analysis, to ensure validity. We summed and divided these ratings by the total possible, to create a percent language usability score for both C and Mayfly. Nine of our eleven participants reported C was their preferred programming language, prior to this study, and seven participants indicated their C programming abilities are above average compared to other students at their university. However, working with Mayfly, ten of eleven eleven users rated its usability higher than C, for completing the assigned challenges. Additionally, Mayfly received a 23% higher mean score than C, on our usability survey.

While we recognize several limitations to this study, including sample size, and, consequentially, order effects, we believe our results indicate that the benefit of migrating to Mayfly from C is high enough to propose using Mayfly for temporally-constrained, batteryless applications.

6 RELATED WORK

Disruption tolerance is a core area of computer science research. However, researchers have traditionally either ignored disruptions due to power failures or treated them as rare catastrophic events. In this section, we describe the Mayfly language and runtime's place in the literature, and demonstrate the novelty of our approach.

Preserving Forward Progress: Checkpointing systems like Mementos [26], Hibernus [1], QuickRecall [15], and others[3, 4, 14, 23, 31] have emerged to keep forward progress on intermittently-powered systems. Other solutions, like DINO [19], show that even with checkpointing, memory consistency is not guaranteed. None of these solutions consider how the loss of timekeeping affects

[2]This study was approved by our Institutional Review Board.

the duty cycle, and how the utility of sensed and computed data changes over time. The proposed Mayfly runtime builds off of DINO and previous checkpointing and scheduling libraries to preserve forward progress *and* manage the temporal aspect of sensing tasks.

Operating Systems & Runtimes Most operating systems for wireless sensor networks have assumed a stable power supply, and were not built for intermittent programs. However, recent advances with computational RFID have pushed for batteryless task management. Dewdrop[5], an energy-aware runtime for Computational RFID tags like the WISP, delays a tag's computation in order to increase the likelihood that the task will complete. DEOS[34] schedules tasks in response to changing energy-harvesting conditions in order to avoid power failures. QuarkOS[33] is a low overhead operating system that divides every communication, sensing, and computation task into tiny fragments. It then sleeps between execution of these fragments to recharge. BY doing this, QuarkOS allows tasks to be executed on extremely small energy budgets. EnOS[30] is a kernel for energy-neutral systems. Energy-neutral systems rely on battery backups but function almost exclusively off harvested energy. EnOS allows for tasks to be organized into different criticality levels, helping manage blackouts. However, EnOS assumes a mostly stable supply, and does not consider the effects of time. In fact, none of the operating systems and runtimes mentioned consider the temporal aspects of sensor data.

Languages Mayfly is the first language designed for batteryless sensors that captures the temporal constraints associated with sensor data. Numerous languages have been adoptedfor wireless embedded sensors [10, 18, 21, 24], most are closely related to or built on top of TinyOS and NeSC [11], which assumes a stable power supply. Synchronous programming languages like Lustre [12], Esterel [2], and Signal [16], are generally used for embedded control systems. This class of languages share Mayfly's consideration of time as critically important to proper function of a program. Mayfly's focus on energy harvesting, intermittent operation, and tracking of data across power failures, separates Mayfly from synchronous languages. However, the well studied mathematical foundations of synchronous languages could greatly inform modeling, specifying, and validating Mayfly programs.

Eon[29] was the first programming language for sensors that was built to be energy-aware. Eon is based on Flux [7], and Mayfly draws inspiration in syntax and purpose from both. Eon programmers used a declarative coordination graph to sequence and categorize tasks in terms of energy states. Tasks are executed based on available energy and dependencies of the task, the goal of avoiding power failures while maximizing performance. Mayfly's design is inspired by Eon—both use declarative coordination graphs and both deal with harvested energy—but Eon assumes a sufficiently large battery and near-term reliable power. Eon is designed to avoid power failures. Mayfly is designed to get work done in spite of power failures. Eon ignores the relationship between data and time, while that relationship is central to Mayfly's design.

7 DISCUSSION AND FUTURE WORK

Mayfly is a first step towards making batteryless sensing mainstream. We envision future work investigating more intelligent, and dynamic task scheduling with time sensitive data streams, generalized hardware platforms for many applications, and more sophisticated tooling to aid the amateur and expert developers of batteryless applications. In this section we discuss the limitations of the Mayfly language and runtime, and outline potential for future work.

Applicability to Energy-Neutral Systems: Mayfly is focused on enabling long running, timely applications for tiny, batteryless, energy harvesting systems, which can lose power many times a second. However, the language is easily applied to other larger classes of sensing systems that deal with less frequent, or longer-term interruptions, such as energy-neutral sensing systems [22]. These sensors may have much more compute resources, larger energy harvesters, and be carefully profiled so as to fail rarely. Mayfly describes *time*—specifically regarding data collection—as a first-order concern. If supported by external remanence timekeepers that can time longer failures (on the order of hours and days) larger energy harvesting sensing systems could more easily manage their timely sensor data collection reliably with Mayfly.

Task Atomicity: A limitation of all languages for intermittently powered sensors stems from task size mismatch to energy availability. If a user task takes a long time and conducts multiple operations, it has a higher chance of failing to complete. Re executing wastes energy, and completion is not guaranteed. Determining if a task is too large is difficult, as externally connected components can influence task time. Exploring compiler and tooling techniques that leverage knowledge about the deployment environment, the predicted available energy, and profiles of the device itself to bound task times and is an interesting and necessary area of future work.

Triggered Tasks: Tasks execute if they have no unsatisfied task dependencies. Providing a more expressive way to trigger tasks is an interesting area of future work. Tasks could, for example, be triggered by changes in environmental conditions (such as sunlight), average energy availability, or even a user interface item like a touch sensor or button.

Dynamic Scheduling: Currently the runtime schedule of tasks is generated at compile time. While this achieves a low overhead and quick runtime functions, it is imagined that the scheduling could integrate contextual information about the energy state of the sensor, the previous energy costs of the current task, and other information to make better scheduling decisions on the fly.

8 CONCLUSIONS

The Mayfly language and runtime was created to provide the developer with a intuitive mechanism to create timely, sophisticated batteryless sensing application. With a graph based declarative language, and a scheduler that maintains time across power failures, Mayfly makes programming these tiny devices more straightforward, without sacrificing flexibility. We validated Mayfly through multiple applications and a user study. Batteryless sensors are an indispensable part of the future of the Internet-of-Things. These devices promise to revolutionize sensing, and even computing. We view Mayfly as a positive step towards the manifestation of this vision.

ACKNOWLEDGMENTS

The authors would like to thank: Chris Datko and Siara Fabbri for initial work on Mayfly; Alexei Colin and Brandon Lucia for providing resources for comparison with Chain; our shepherd, Gian Pietro Picco, and our anonymous reviewers, for their helpful comments. This research is based upon work supported by the National Science Foundation under grant CNS-1453607. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation

REFERENCES

[1] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embedded Systems Letters, IEEE* 7, 1 (2015), 15–18.

[2] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

[3] Naveed Bhatti and Luca Mottola. 2016. Efficient State Retention for Transiently-powered Embedded Sensing. In *Proceedings of the 13th ACM International Conference on Embedded Wireless Systems and Networks (EWSN) Graz (Austria)*.

[4] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17)*. ACM, New York, NY, USA, 209–219. https://doi.org/10.1145/3055031.3055082

[5] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-Aware Task Scheduler for Computational RFID. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*.

[6] Michael Buettner, Richa Prasad, Matthai Philipose, and David Wetherall. 2009. Recognizing daily activities with RFID-based sensors. In *Proceedings of the 11th international conference on Ubiquitous computing*. ACM, 51–60.

[7] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D Berger, and Mark D Corner. 2006. Flux: A Language for Programming High-Performance Servers. In *In Proceedings of USENIX Annual Technical Conference*.

[8] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 577–589. https://doi.org/10.1145/2872362.2872409

[9] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

[10] Roland Flury and Roger Wattenhofer. 2010. Slotted Programming for Sensor Networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '10)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/1791212.1791216

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI'03)*. ACM, San Diego, CA, USA, 1–11.

[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[13] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. 2016. Persistent Clocks for Batteryless Sensing Devices. *ACM Trans. Embed. Comput. Syst.* 15, 4, Article 77 (Aug. 2016), 28 pages. https://doi.org/10.1145/2903140

[14] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 228–240.

[15] Harishankar Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 330–335.

[16] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE* 79, 9 (1991), 1321–1336.

[17] Logic-less templates. 2017. Mustache Logic-less templates. http://mustache.github.io/. (2017). Last Viewed March 22, 2017.

[18] Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh. 2008. Resource Aware Programming in the Pixie OS. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*. ACM, New York, NY, USA, 211–224. https://doi.org/10.1145/1460412.1460434

[19] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[20] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[21] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, Vol. 43. ACM, 335–346.

[22] Geoff V Merrett and Bashir M Al-Hashimi. 2017. Energy-driven computing: Rethinking the design of energy harvesting systems. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 960–965.

[23] Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. 2013. Automated checkpointing for enabling intensive applications on energy harvesting devices. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. IEEE Press, 27–32.

[24] Luca Mottola and Gian Pietro Picco. 2011. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Comput. Surv.* 43, 3, Article 19 (April 2011), 51 pages. https://doi.org/10.1145/1922649.1922656

[25] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. 2012. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 221–236. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati

[26] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices.. In *Proceedings of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] A. Rodriguez, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddel. 2017. Intermittently-powered energy harvesting step counter for fitness tracking. In *2017 IEEE Sensors Applications Symposium (SAS)*. 1–6. https://doi.org/10.1109/SAS.2017.7894114

[28] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Trans. Instrumentation and Measurement* 57, 11 (Nov. 2008), 2608–2615.

[29] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[30] Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, and Volkmar Sieh. 2016. A Kernel for Energy-Neutral Real-Time Systems with Mixed Criticalities. (2016).

[31] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 17–32. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

[32] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011. *Moo: A Batteryless Computational RFID and Sensing Platform*. Technical Report UM-CS-2011-020. UMass Amherst Department of Computer Science.

[33] Pengyu Zhang, Deepak Ganesan, and Boyan Lu. 2013. QuarkOS: Pushing the Operating Limits of Micro-powered Sensors. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=2490483.2490490

[34] Ting Zhu, Abedelaziz Mohaisen, Yi Ping, and Don Towsley. 2012. DEOS: Dynamic energy-oriented scheduling for sustainable wireless sensor networks. In *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2363–2371.