Adaptable Runtime Monitoring for Intermittent Systems

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

Khakim Akhunov
University of Trento
Trento, Italy
khakim.akhunov@unitn.it

Lorenzo Antonio Riva
University of Trento
Trento, Italy
lorenzoantonio.riva@studenti.unitn.it

Arda Goknil
SINTEF Digital
Oslo, Norway
arda.goknil@sintef.no

Ivan Kurtev
Eindhoven University of Technology
Eindhoven, The Netherlands
i.kurtev@tue.nl
CapGemini Engineering
The Netherlands
ivan.kurtev@capgemini.com

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

Abstract

Batteryless energy harvesting devices compute intermittently due to power failures that frequently interrupt the computational activity and lead to charging delays. To ensure functional correctness in intermittent computing, applications must exhibit several unique properties, such as guarantees for computational progress despite power failures and prevention of stale operations caused by charging delays. We observe that current software support for intermittent computing allows for checking only a fixed set of properties and leads to tightly coupled application and property-checking, thus hampering modularity, scalability, and maintainability.

In this paper, we present ARTEMIS, the first framework designed to facilitate flexible property checking of intermittent programs at runtime. ARTEMIS is developed based on techniques from the area of runtime monitoring, offers a specification language for specifying an open set of properties, and provides automatic generation of monitors responsible for checking the properties. Our evaluation showed that ARTEMIS achieves comparable efficiency to state-of-the-art solutions while significantly preventing failure scenarios through its monitoring capabilities.

CCS Concepts: **• Computer systems organization → Embedded software**; **• Software and its engineering → Domain specific languages**.

Keywords: Intermittent Computing, Runtime Monitoring, Batteryless Internet of Things

ACM Reference Format:
Eren Yıldız, Khakim Akhunov, Lorenzo Antonio Riva, Arda Goknil, Ivan Kurtev, and Kasım Sinan Yıldırım. 2024. Adaptable Runtime Monitoring for Intermittent Systems. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3650070

1 Introduction

Providing a constant and wired power source for IoT devices, especially those in remote areas, is often infeasible [3, 32], leading to limitations in system design, operation, and maintenance [2]. In contrast, batteryless IoT devices (e.g., SuperSensor [7] and Camaroptera [24]) powered by ambient energy sources like radio waves, sunlight, or heat offer sustainability, reduced maintenance, longevity, and cost-effectiveness. These devices convert ambient energy into electrical power using harvesters and store it in capacitors. However, due to limited capacitor capacity and dynamic ambient energy, they experience frequent power failures, resulting in the loss of computational state (e.g., registers and memory contents). Consequently, they perform *intermittent computations* [29], periodically backing up their state when power failure is imminent and restoring it when sufficient energy is available to resume. Usually, a specialized software called *intermittent runtime* is deployed on these devices to handle the tasks of backing up and restoring the computational state. Various runtimes (e.g., [38, 40, 50]) have been proposed to hide the intermittent execution complexity while enabling power failure-resilient execution of applications.

Intermittent programs face challenges due to unpredictable and uncontrollable energy sources [26]. Variability in energy availability leads to charging delays, power outages, and repeated code execution [49]. These issues necessitate

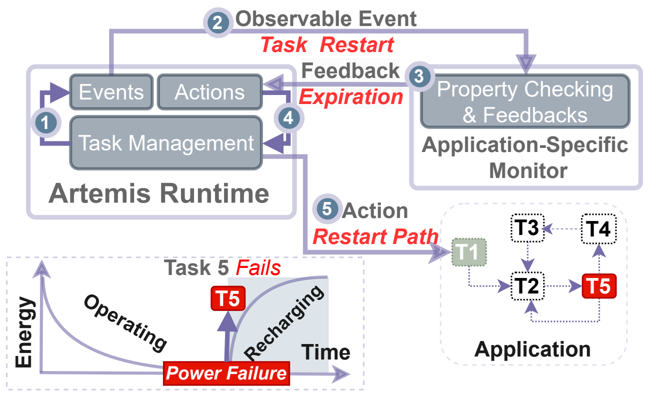


Figure 1. The ARTEMIS framework. The runtime executes tasks and delivers observable events (Task 5 restart due to power failure, steps ① and ②) to the application-specific monitor. The monitor checks the programmer-specified properties and returns actions to be taken if they are violated (data expiration, step ③). The runtime checks the actions and executes them (restarts the path, steps ④ and ⑤).

specific *properties* for intermittent programs' functional correctness. An essential property of intermittent programs is *computational progress* in case of power failures. This property requires checking if the energy capacity of the device is sufficient to execute the code between two consecutive backup points. Another property is *timely execution* [31, 35], which prevents the use of data outdated due to lengthy charging delays. If such properties are not checked, intermittent programs might fail, e.g., due to non-termination (i.e., no computational progress) and stale outputs.

Recent works proposed compile-time solutions (e.g., ETAP [26]) for assessing specific properties of intermittent programs without running them on the target platform. However, these compile-time solutions rely on prior information about the target environment, necessitating extensive profiling efforts and lacking consideration for unpredictable runtime dynamics. In contrast, some studies introduced runtime solutions, such as TICS [35] and Mayfly [31], to check timely execution property during program execution. These runtime solutions have limitations due to their rigid design, resulting in several associated problems listed below:

(P1) Tightly-coupled Application and Property Checking. In some approaches (e.g., TICS [35], where time constraints are directly given in the source code), the property checking is the responsibility of the developer (e.g., ensuring freshness of data), resulting in code intertwined with the application code. This tight coupling adds complexity to intermittent applications, making it hard to adapt responses to property violations or changing property requirements without modifying the application code.

(P2) Tightly-coupled Runtime and Property Checking. Some approaches support the separation of property specification and application logic, delegating the property checks to the intermittent runtime. For example, Mayfly's runtime integrates timeliness property checking and power failure-resilient application execution [31]. However, as a consequence, adjustments to property-checking logic might require alterations to the entire core runtime.

(P3) Unscalable Property Checking. Existing intermittent computing runtimes often have limited property-checking capabilities restricted to a fixed set of supported properties (e.g., Mayfly [16] can detect timeliness violations but not non-termination). As the runtime evolves with new features, incorporating new properties to be checked or adapting existing ones becomes more challenging.

Our Contributions. In this paper, we introduce and assess ARTEMIS, the first framework designed to facilitate flexible property checking of intermittent programs at runtime (see Figure 1). ARTEMIS is developed based on techniques derived from the domain of ***runtime monitoring*** *(aka runtime verification)* [10], which involves monitoring the execution of a system to ensure the satisfaction of its desired properties. ARTEMIS targets task-based intermittent applications [15, 38, 48] in which the programmer decomposes the computation into atomic tasks with a control flow. The ARTEMIS intermittent computing runtime executes these tasks in a power-failure resilient manner, meanwhile feeding one or more application-specific monitors with a sequence of events indicating the start and end of task execution at specific timestamps. On top of these primitive events, ARTEMIS enables defining an open set of properties, e.g., the maximum allowed duration for executing a task and task periodicity. It proposes a property specification language for expressing such properties and provides automatic generation of monitors responsible for checking the properties. The generated monitors evaluate properties and recommend corrective actions to the ARTEMIS runtime, such as skipping or restarting a task. We assessed ARTEMIS against Mayfly [31], demonstrating its efficiency and superiority, particularly in preventing non-termination scenarios. ARTEMIS is open-source [6]. In summary, our contributions include:

(1) Improved separation of concerns between property specification and application logic that supports seamless integration of diverse property checking approaches without any modification to the application code;

(2) Modular runtime architecture that decouples the generic intermittent runtime functionality from the property checking logic (implemented in monitors), retaining their ability to evolve independently;

(3) Scalable property checking based on monitoring an open set of properties with minimal programming effort, facilitated by an expressive property language and automated generation of monitor code.

2 Background on Intermittent Programs

Batteryless sensing applications (e.g., [3, 23]) face intermittent power supply constraints since they rely exclusively on ambient power sources. To achieve power failure-resilient execution in such contexts, various software solutions have been proposed [13, 15, 35, 43, 48, 50]. *Checkpointing systems* [4, 5, 9, 11, 14, 34, 37, 43, 47] capture volatile state snapshots, including registers, stack, and global variables, in non-volatile memory at programmer-defined points, facilitating computation state recovery after power failures. In *task-based systems* [8, 15, 38–41, 44, 48], programs are decomposed into tasks with task-based control flow. The runtime tracks active tasks, restarts them after power failures, ensures atomic completion, and progresses to the next task. This study focuses on task-based systems.

2.1 Properties of Intermittent Programs

Intermittent programs face unpredictability due to unstable ambient energy sources [26], resulting in frequent power failures, varying charging delays, and repeated code execution. These factors introduce essential properties for intermittent program correctness. In this paper, we primarily focus on the following fundamental intermittent program properties (in the context of task-based systems), which can be extended and combined.

(1) Maximum Inter Task Delay (MITD) defines the allowed maximum delay between two tasks. For instance, the sensed data expires when it cannot be consumed within a specific timeframe due to longer charging times [31, 35, 49].

(2) Maximum Task Re-execution Count defines the maximum number of successive re-execution attempts for a specific task. Limited ambient energy, wrong capacitor size selection [17], or peripheral operations [12, 49] might lead to successive power failures during the execution of the same task, leading to a lack of progress and non-termination.

(3) Total Execution Time defines the maximum duration allocated for completing a computation or generating a response. It ensures timely decision-making and prevents unnecessary computation by terminating tasks that exceed the allocated time frame [40].

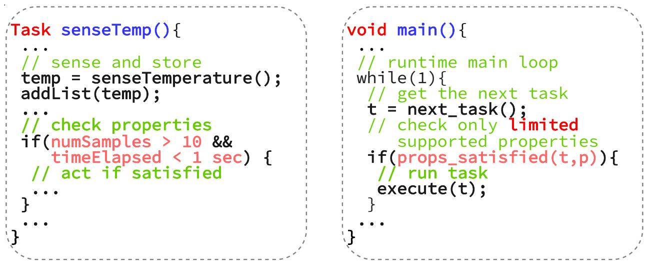
(4) Number of Samples defines the required number of samples to execute a particular task. A sensing application may necessitate varying sample counts from each sensor (e.g., temperature) to determine the need for action.

(5) Periodicity defines the desired frequency of task execution. Programs may experience power failures and charging periods, leading to unsuccessful sampling attempts. In such cases, they may need to restart the sampling process anew.

2.2 Problems with Checking Program Properties

Current intermittent programming solutions lack clear separation between the application and property checking codes, resulting in several associated problems.

2.2.1 Tightly-coupled Application and Property Checking. In Figure 2(a), the senseTemp task in Chain [15] is responsible for temperature sensing and storage. However, it also includes checking the number of collected samples (*numSamples*) and elapsed time (*timeElapsed*). This integration of property checking logic within the application code requires modifications to the application code for any property-related changes or updates, which results in increased code complexity. This intertwining of concerns within the same codebase makes code comprehension, maintenance, and debugging challenging as both application and property checking logic complexity grows, raising the risk of errors.



```
Task senseTemp(){
 ...
 // sense and store
 temp = senseTemperature();
 addList(temp);
 ...
 // check properties
 if(numSamples > 10 &&
    timeElapsed < 1 sec) {
  // act if satisfied
  ...
 }
 ...
}
```

(a) Application checks properties

```
void main(){
 ...
 // runtime main loop
 while(1){
  // get the next task
  t = next_task();
  // check only limited
     supported properties
  if(props_satisfied(t,p)){
   // run task
   execute(t);
  }
 ...
}
```

(b) Runtime checks properties

Figure 2. Integrating property checking in the application code (e.g., Chain [15]) is error-prone and requires application modifications when altering property checking logic. The same drawbacks arise when the runtime checks properties (e.g., Mayfly [31]), requiring modifications to the runtime.

2.2.2 Tightly-coupled Runtime and Property Checking. Figure 2(b) portrays a representation of the main loop of Mayfly runtime [31], wherein the monitoring is tightly coupled with the runtime logic. `props_satisfied(t,p)` verifies the satisfaction of properties associated with task `t` during its execution. This tightly coupled arrangement restricts the adaptability of the runtime system, as any adjustments or enhancements to the property checking logic or the properties examined necessitate modifications to the runtime code.

2.2.3 Unscalable Property Checking. Integrating property checking logic directly into the runtime loop, as in Figure 2(b), limits property-checking capabilities. The code snippet implies that only a fixed set of properties is checked using the `props_satisfied(t,p)` function, as these properties must be hardcoded in the main runtime loop. This approach hampers the ability to accommodate more complex or domain-specific properties that require additional context or data. It introduces overhead and complexity into the runtime system, as computational resources and memory are allocated for property checking logic unutilized. This complexity can result in inefficiencies and potential performance issues as the main runtime loop becomes burdened with extraneous code.

3 The ARTEMIS Framework

ARTEMIS addresses the problems above through design principles based on *runtime monitoring*, a dynamic analysis method for checking system behavior against correctness properties [10]. Runtime monitoring involves a software

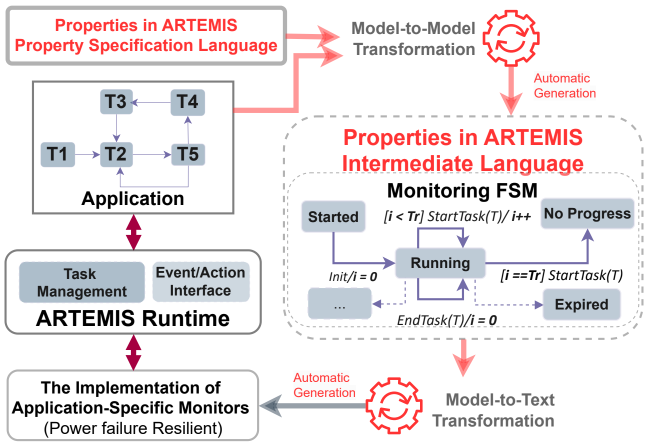


Figure 3. Application development flow in ARTEMIS framework, which includes (i) the specification of application properties in property specification language, (ii) the generation of the properties in intermediate language, and (iii) the generation of application-specific monitors.

component, the *monitor*, which checks properties using observations (input/output data or system state information) obtained during system execution. Properties are typically expressed in formal languages like temporal logic, regular expressions, state machines, or grammars [28]. These languages are versatile but generic. Alternatively, a problem-specific language can be designed and translated into a generic language. ARTEMIS adopts this approach, introducing a consise *property specification language* and an *intermediate language* representing properties in a generic way as finite-state machines.

ARTEMIS offers a modular architecture for scalable intermittent programs, separating property specification from application code. It supports dynamic adaptation of intermittent applications based on runtime conditions, improving system robustness during energy disruptions and changing operating conditions. Figure 3 shows the system overview of ARTEMIS, comprising a runtime, property specification language, and a transformation pipeline for creating application-specific monitors.

In ARTEMIS, developers identify properties to monitor in the intermittent systems, which entails considering energy availability, timing requirements, and other pertinent factors that significantly influence the system's behavior and performance. They use the ARTEMIS property specification language to express these properties independently from the application code. This step involves setting property thresholds, defining interactions between properties, and specifying runtime actions performed in case of property violations.

ARTEMIS uses a model-to-model transformation [45] to create monitors in an intermediate language based on state machine concepts. This language bridges the gap between the ARTEMIS property specification language and the target

```
1  // Task definitions
2  Task bodyTemp() {
3      ...
4      temp = convert(sampleADC());
5      ...
6  }
7  Task accel() {...}
8  Task micSense() {...}
9  Task send() {...}
10 Task calcAvg(depData,&avgTemp) {...}
11 Task heartRate() {...}
12 Task filter() {...}
13 Task classify() {...}
14 //Execution paths of tasks
15 (Path: 1, bodyTemp, calcAvg, heartRate, send)
16 (Path: 2, accel, classify, send)
17 (Path: 3, micSense, filter, send)
```

Figure 4. Health monitoring application code.

C code, allowing for higher-level, domain-oriented property representation. Employing a model-to-text transformation [42], ARTEMIS generates monitoring C code from the intermediate language, seamlessly integrating it with the application code and the ARTEMIS runtime. The monitoring code captures runtime information (such as timing events and task executions), evaluates it against specified properties, and determines runtime actions for property compliance.

3.1 ARTEMIS Monitorable Applications

In ARTEMIS, developers employ a task-based programming model [15, 31, 38, 48] to implement their applications. This model enables the expression of desired properties at the task level. Tasks are atomic units with all-or-nothing semantics; any power interruption leads to the runtime rolling back task modifications on nonvolatile memory before the task restart. Successful task execution stores outputs in nonvolatile memory, and the next task in the control flow is executed. This model permits ARTEMIS to check properties upon task restart or completion.

Figure 4 illustrates code for a health monitoring application in a wearable device, tracking health parameters. The application employs sensors for data collection, including body temperature, acceleration, and other vital signs. It continuously monitors the user and provides insights into her health status, enabling proactive healthcare management and early detection of abnormalities. The code snippet defines tasks, e.g., *bodyTemp* (Line 2), which processes temperature measurements, and others like *accel*, *micSense*, *send*, *calcAvg*, *heartRate*, *filter*, and *classify* (Lines 7-13), handling various functions within the application. The *bodyTemp* task involves converting the output of the *sampleADC()* function into a temperature value and storing it in the temp variable. In addition, the *calcAvg* task states that a dependent variable (*avgTemp*) needs to be monitored. This variable address is kept in the task context and transmitted to the monitor when calling the monitor. Specific task operations are not detailed in this snippet but are represented by the ellipsis

```
1  micSense: {
2   maxTries: 10 onFail: skipPath;
3  }
4
5  send: {
6   MITD: 5min dpTask: accel onFail: restartPath maxAttempt:
        3 onFail: skipPath Path: 2;
7   maxDuration: 100ms onFail:skipTask;
8   collect: 1 dpTask:accel onFail: restartPath Path: 2;
9   collect: 1 dpTask:micSense onFail: restartPath Path: 3;
10 }
11
12 calcAvg {
13  collect: 10 dpTask:bodyTemp onFail: restartPath;
14  dpData: avgTemp Range: [36, 38] onFail: completePath;
15 }
16
17 accel {
18  maxTries: 10 onFail: skipPath;
19 }
```

Figure 5. An example property specification in ARTEMIS for the example intermittent application code in Figure 4.

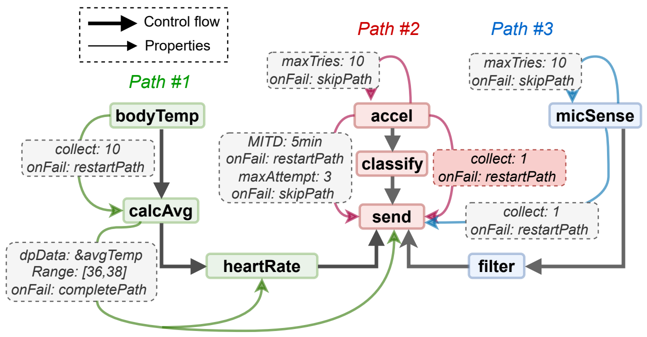


Figure 6. Paths, tasks, and properties from Figure 5.

("..."). Path execution is specified in Lines 15-17, showing the task sequence for the intermittent application.

3.2 ARTEMIS Property Specification Language

The ARTEMIS property language is declarative. In declarative programming, developers specify what they want to achieve without detailing how. In ARTEMIS, developers use this language to define desired properties of the intermittent system, focusing on system properties rather than implementation specifics. Table 1 explains ARTEMIS property language constructs. Figure 5 shows an example property specification for the application in Figure 4. Figure 6 visualizes task paths and properties from Figure 5 for illustration purposes, though not required by ARTEMIS.

The *micSense* task allows a maximum of ten execution attempts before bypassing its path (Lines 1-3 in Figure 5). The *send* task has a Maximum Inter-Task Delay (MITD) of five minutes (Line 6) for receiving data from the *accel* task; noncompliance results in path restart with up to three attempts before bypassing its entire path. The *send* task also has a maximum execution duration of 100 milliseconds (Line 7), leading to the task skipping on failure. Additionally, the *send* task requires one data item from the accel task and one from the *micSense* task (Lines 8 and 9); failure triggers path restart. Explicit path specification for skipping or restarting is only needed for the *send* task (e.g., Path 2: *accel*, *classify*, *send* for the *MITD* property in Line 6 and the *collect* property in Line 8) due to path merging; other tasks (*micSense*, *calcAvg*, and *accel*) do not require such specification as they do not involve path merging. In Line 13, we specify that the *calcAvg* task must collect ten data items from the *bodyTemp* task. In Line 14, we also specify that the *calcAvg* task has a data dependency with its final result (*avgTemp* - defined during the Task declaration in Figure 4) to immediately complete the current path without executing any other path. Thus, if the average body temperature exceeds the given range (36 - 38 C°), the current path is directly completed by executing *heartRate* and *send* task without property checking to inform the emergency case. Additionally, for the *accel* task (Lines 17-19), we set a maximum of ten execution attempts before bypassing its path.

3.3 ARTEMIS Application-Specific Monitors

The ARTEMIS intermediate language supports the specification of monitors as state machines. Each monitor is a single-state machine, usually derived from a single property. The triggers of the state transitions are the events that the runtime sends: the start and end of tasks with a timestamp of the event. State machines can define variables of commonly used types (integer, boolean, and others), transitions may have boolean expressions as guards, and transition bodies contain statements like assignment and if-then-else construct. Furthermore, each transition may signal property failure with possible further actions to be taken by the runtime, for example, skipping a task or path.

In most cases, developers interact with the intermediate language for property specification indirectly, as the system automatically generates it. While the primary interaction occurs via the more concise property specification language, there might be situations where this language lacks the necessary expressiveness. In such cases, developers can engage directly with the intermediate language.

Each property expressed in the ARTEMIS property specification language is translated into a single state machine in the ARTEMIS intermediate language. Figure 7 presents state machines for four properties.

The first property addresses the maximum number of attempts to initiate task A, considering the system's intermittent nature where power failures may interrupt task execution. The initial state (*NotStarted*) indicates that task A has not yet begun. A transition occurs when the event *startTask*, carrying the task name and timestamp (stored as variable *t*), is triggered - but only if the task name is A. The transition body (marked after "/") contains an assignment action for variable *i*, which tracks the number of attempts to start task A. Within the state *Started*, two transitions are triggered

Table 1. Summary of the ARTEMIS Property Specification Language Constructs.

Language Construct	Type	Explanation
period	Property	Specifies the time interval between consecutive executions of a task, defining its periodicity. It assumes a jitter. If the time interval is exceeded, the specified onFail action is taken.
maxTries	Property	Defines the maximum number of attempts allowed for executing a task. If the task exceeds this limit, the specified action will be taken.
maxDuration	Property	Defines the maximum duration or time limit for the execution of a task. If the task exceeds this limit, the specified action will be taken.
MITD	Property	Stands for "Maximum Inter-Task Delay" and defines the maximum duration within which the current task should receive data from another task.
collect	Property	Specifies the number of data items the current task requires to obtain from another task.
dpData	Property	Specifies the inter-task data dependencies (when the task has a dependency on its result or another task).
dpTask	Variable	Used within the context of a property to specify the dependency of a task on the execution of another task.
Path	Variable	Used within the context of a property to identify a specific path within the application code.
Range	Variable	Used within the context of a property to specify the range of the dependent data to decide the action of the dependent task. If the dependent data is out of the given range, the onFail action is applied.
maxAttempt	Control Flow	Used within the context of a property to specify the maximum number of attempts allowed for executing a task. It is used with time-related properties (period and MITD) to avoid non-termination.
onFail	Control Flow	Used within the context of a property to specify the case when a property fails to be satisfied. It is used with other control flow constructs that specify the action (restarting a path, skipping a path, restarting a task, or skipping a task) when the property fails.
onFail: restartPath	Control Flow	Represents an action to be taken when a property fails to be satisfied, causing the current path to be restarted.
onFail: skipPath	Control Flow	Represents an action to be taken when a property fails to be satisfied, causing the current path to be skipped.
onFail: restartTask	Control Flow	Represents an action to be taken when a property fails to be satisfied, causing the current task to be restarted.
onFail: skipTask	Control Flow	Represents an action to be taken when a property fails to be satisfied, causing the current task to be skipped.
onFail: completePath	Control Flow	Represents an action to be taken when a property fails to be satisfied, causing the immediate termination of the current path without executing any further paths. This action preserves the next task, halts the monitoring of other properties, and executes subsequent tasks within the same path until its full completion. Upon path completion, execution resumes by monitoring properties starting from the preserved next task.

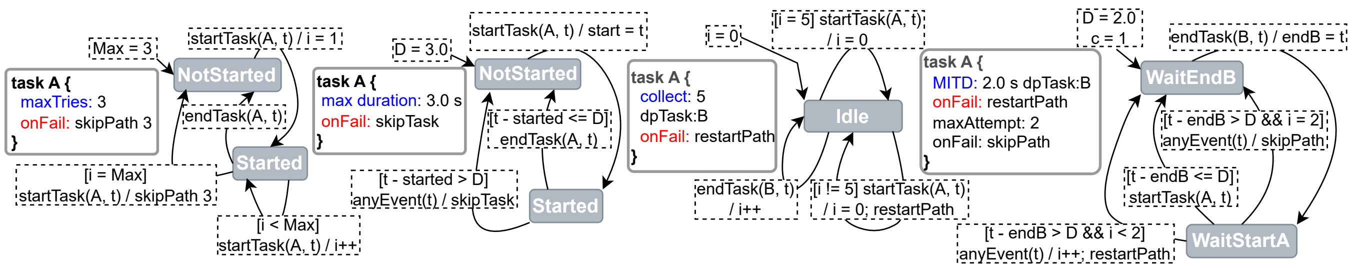


Figure 7. Implementation of the ARTEMIS properties as finite state machines in the ARTEMIS intermediate language.

by the start event of task A, each with guards specified in square brackets before the event. The counter is incremented if the number of attempts is less than the maximum allowed. Upon reaching the maximum attempts, the monitor signals a failure and sends the corresponding action (in this case, *skipPath*) to the runtime. Finally, upon observing the end of task A, the property is considered satisfied, and a transition back to the initial state occurs without reporting any failure.

The second property (second state machine in Figure 7) addresses the maximum execution duration of task A, exemplified as three seconds in this example. The transition from the initial state to the state *Started* occurs upon initiating task A, with the start time stored in the variable *start*. Two transitions are present within the state *Started*. The first transition signifies the satisfaction of the property: the end of task A falling within the allowed time interval (represented by variable *D*) after the start. If any event occurring beyond this time interval is observed (indicating a failure to satisfy the property), the failure is reported with the action *skipTask*. The trigger *anyEvent* encompasses both the start and end events of tasks. Events that do not have a specified transition are always accepted, resulting in no actions taken and no change in state (implicit self-transition). In our second example, this scenario could correspond to the event *startTask* that still falls within the permissible time interval.

The third property (third state machine in Figure 7) addresses the number of data items task A needs to obtain from another task (task B). The property is structured around a single state, incorporating a transition that delineates the successful execution of task B by incrementing the counter, represented by variable *i*. To facilitate the start of task A, two distinct transitions are delineated, each governed by mutually exclusive conditional guards. In the example, if

i is not 5, a failure is signaled through action *restartPath*, concurrently resetting the counter to zero.

The fourth property (fourth state machine in Figure 7) addresses the Maximum Inter-Task Delay (MITD) requirement, exemplified as task A should commence within a two-second timeframe following the completion of task B. Failure to meet this condition triggers a path restart. If the constraint is violated twice, the entire path is skipped (with *maxAttempt* set to 2). The state machine logic involves recording the completion timestamp of task B in variable *endB* and monitoring the elapsed time until the initiation of task A. The guards of transitions from state *WaitStartA* to state *WaitEndB* capture the possible cases.

Multiple properties may fail concurrently for a given event, such as both maximum duration and maximum start attempts for a task. In such cases, multiple monitors will report failures to the runtime. The runtime determines the appropriate course of action in response to the suggested ones. If a proposed action involves path restart, monitors linked to already initiated tasks within that path must be re-initialized.

Using an intermediate language provides several advantages in translating property specifications from the ARTEMIS property language to the final C code. It closes the abstraction gap between the domain-specific compact property descriptions and their C implementation. In Figure 7, each property has a fairly simple state machine representation. State machines' translation to C is straightforward (multiple solutions in the literature). The ARTEMIS architecture also enables the creation of different property specification languages on top of the intermediate language, enhancing reusability.

The shown state machines are abstract and do not account for the intermittent nature of the generated C code, which will run in tandem with the monitored application. Preserving the state machine variable values and machine current state information throughout power failures is of paramount importance. Section 4 elaborates on this.

3.4 ARTEMIS Runtime

The ARTEMIS runtime deploys monitors with the application to evaluate properties during execution. Monitors receive *startTask* and *endTask* events from the runtime, indicating task initiation and completion. The *startTask* event precedes task execution, while the *endTask* event follows task completion, enabling post-task scheduling. This event-based communication establishes a feedback loop between the runtime and monitors, facilitating dynamic control during execution.

4 ARTEMIS Implementation

We implemented ARTEMIS runtime in C language. Our target platform was an MSP430FR [46] series microcontroller, the de facto standard computational platform for intermittent systems with an internal FRAM. Like in many intermittent computing solutions, e.g., TICS [35], InK [48], and

```
1  // Data structure to maintain observable monitor event
2  typedef struct _MonitorEvent {
3      eventkind_t kind; // StartTask, EndTask
4      float timestamp; // timestamp of the event
5      void* taskAddr; // current task pointer
6  } MonitorEvent_t;
7
8  // persistent variable to hold the last monitor event
9  MonitorEvent_t event;
10
11 int main {
12     ...
13     // Initialize constraints to be monitored
14     resetMonitor();
15     // Progress interrupted monitor operation
16     monitorFinalize();
17     ...
18     while(1){ // start running application tasks
19         // check current task properties
20         if (curtask == checkTask(curtask)){
21             run(curtask); // run the selected task
22             // commit current task and get the next task
23             curtask = taskFinish(curtask);
24         }
25     }
26     ...
27 }
```

Figure 8. ARTEMIS runtime main loop.

Mayfly [31], ARTEMIS requires keeping track of timestamps, which implies persistent timekeeping [22, 31, 35, 51] helping not to lose the notion of time due to power failures.

The ARTEMIS property and intermediate specification languages are implemented in the Xtext language workbench [27] that uses the Eclipse Modeling Framework (EMF) [45] for model manipulation. EMF supports tools for model-to-model and model-to-text transformations used to implement the ARTEMIS generator pipeline. When generating monitors, we used ImmortalThreads [50] library to enable power failure-resilient execution of the generated monitors.

4.1 ARTEMIS Observable Runtime

Figure 8 presents the main entry of the ARTEMIS runtime.
Initial Hard Reset. When the system boots for the *first time*, the monitor is initialized to reset its internal variables (`resetMonitor`, Line 14), performed only once during the application life cycle. This hard reset is necessary not only for monitoring properties but also for task handling (not shown in the code listing) since there are internal variables that should be initialized for the consistent start of the system.
Reboot and Monitor Progress. Another issue is keeping application-specific monitors always in a consistent state since a power failure might interrupt monitoring. At each reboot, ARTEMIS progresses the monitor to finalize its interrupted task (`monitorFinalize`, Line 16).
Monitor Event Structure. ARTEMIS maintains a persistent variable `event` of type `MonitorEvent_t`, which is a structure that holds the event type (task start or task end), the event

```
1  task_t *checkTask(task_t *curtask){
2      ...
3      // check current task status
4      switch(curtask->status){
5          case TASK_READY:
6              event.kind = StartTask;
7              event.timestamp = GetTime();
8              break;
9          case TASK_FINISHED:
10             event.kind = EndTask;
11             event.depData = curtask->depData;
12             break;
13     }
14     // monitor checks properties and returns the action
15     result = callMonitor(taskEvent);
16     // decide the next task using the monitor action
17     return getNextTask(result);
18 }
19
20 task_t * taskFinish(task_t *curtask){
21     // update finish time
22     curtask->finish = event.timestamp = GetTime();
23     // set the status of the task
24     curtask->status = TASK_FINISHED;
25     ...
26     return checkTask(curTask);
27 }
```

Figure 9. Calling monitor and determining next task.

timestamp, and a task pointer. This variable is used to pass the event type and corresponding data to the monitor for checking the system properties specified by the developer.

4.1.1 Task Execution

ARTEMIS employs a task-based model [15, 31, 38, 48], where tasks are atomic, and their re-execution is idempotent. It executes a task and commits its changes only after the task execution is finalized. ARTEMIS maintains a structure to hold information to manage the tasks, such as task status (TASK_READY and TASK_FINISHED), task pointer, and timestamps regarding task start and finish.
ARTEMIS Main Loop. ARTEMIS executes tasks by executing the loop in Lines 18–25 in Figure 8. The runtime calls checkTask (Line 20) to check the properties of the current task. If the properties are satisfied (indicated by the equality of the current task pointer and the returned value from the checkTask), the task is run (Line 21). Otherwise, checkTask returns the next task whose properties is checked in the next iteration. If the task is run without a power failure, taskFinish is called (Lines 23) to finalize the task execution and get the next task to execute. If a power failure occurs, the loop is called again without any inconsistency issue.

4.1.2 Property Checking and Executing Tasks.

Function checkTask in Lines 1–18 in Figure 9 checks the properties of the given task and returns the task to be executed.
Path and Task Order. The path order in the task graph is followed during task execution. Once a path is selected, its tasks are executed sequentially in the specified order until the path is completed or prematurely halted due the violation of associated task/path properties. The subsequent path and its tasks are executed similarly, ensuring application progress.
Property Checking. Upon its initial launch, ARTEMIS selects the first task of the first path and sets the task status as TASK_READY. As the main loop commences execution, it invokes checkTask, which fills the event structure (event) with the event information StartTask (Figure 9, Lines 5–8). It calls the monitor via callMonitor to retrieve the properties to be checked by the monitor (Line 15). If a power failure occurs after the first task is selected, ARTEMIS re-invokes checkTask from the case TASK_READY (Line 5) and re-engages the monitor with the StartTask event.
Property Violation. After calling callMonitor, the monitor returns the property violation status and corresponding action (Line 15). ARTEMIS checks the result and decides the next task utilizing this information (getNextTask, Line 17). If all properties are satisfied, getNextTask returns the current task to be executed by ARTEMIS (Figure 8, Line 21). If there is a property violation, getNextTask returns a task in the application graph based on the property specification.
Task Finish. Upon task completion, taskFinish is called (Figure 9, Lines 20–27) to update the end-time fields of both task structure and event data (Line 22) and set the current task status to TASK_FINISHED. checkTask is invoked to send the EndTask event to monitor, to let the monitor check the properties, and to return the next task to execute (Line 26).

4.1.3 Timestamps Consistency.

As depicted in Figure 9, the assignment of timestamps to task start and end events occurs at distinct sections of the code. In the event of a power failure after the task status has been set to TASK_FINISHED, ARTEMIS refrains from updating the timestamp of the EndTask event, ensuring that callMonitor consistently receives the accurate finalization time of the task. Conversely, during the initial triggering of callMonitor by the StartTask event, the monitor does not internally record the start time. Consequently, although the timestamp of the StartTask event is updated with each restart prompted by a power failure (Lines 5–8), the monitor disregards these values and retains the initial timestamp. These considerations bear significance in checking time-related properties accurately.

4.2 Intermittently-Executable Monitors in C

The ARTEMIS specification and intermediate languages are implemented using the Xtext language workbench [27], which offers text-based syntax defined by an Xtext grammar. This grammar enables automatic editor and parser generation. After parsing, specifications become EMF models, allowing the use of Model Driven Engineering (MDE) techniques like model-to-model and model-to-text transformations. The Xtext-generated editors provide advanced features such as syntax highlighting, auto-completion, and type checking, enhancing the development experience.

Monitor Code Generation. Generating C monitoring code from ARTEMIS property specifications involves a two-step process. First, the specifications are transformed into state machines in the ARTEMIS intermediate language through a model-to-model transformation. This transformation contains syntax-directed mapping rules, where each property in the specification language corresponds to a template encoding the associated state machine. Furthermore, there are transformation templates for the simpler syntactical constructs such as expressions, constants, and function calls. The mapping rules are encoded in a Java-like language integrated into the Xtext workbench.

Subsequently, the state machines are translated into C code via a model-to-text transformation. This translation is relatively straightforward, with various possible implementations documented in existing literature [25, 30]. The transformation process navigates the state machine in a top-down fashion (processing the states, the transitions, and the statements in their bodies). The main challenge in designing the resulting C code lies in ensuring its suitability for intermittent execution, although this aspect does not pose a challenge for the transformation process itself. To overcome this challenge, the state machines are translated into C code using the ImmortalThreads library [50], equipped with a compiler frontend facilitating source-to-source transformations to generate intermittently executable binaries. This library furnishes C macros for constructing power failure-resilient monitors, employing a local continuation approach to resume execution post-power failures. To advance the monitor and conclude event handling after reboots, ARTEMIS invokes the `monitorFinalize` function (Figure 8, Line 16).

4.2.1 Code Example.

Figure 10 presents a simplified rendition of the monitor code generated by the ARTEMIS framework. This generated monitor code implements the interface `callMonitor`, which functions to receive event notifications and provide result feedback. `callMonitor` takes an `event` parameter comprising the event type, timestamp, and task pointer. It returns a result structure containing the action type and path information.

Data Structures. The monitor maintains the `property_t` data structure (Lines 1–9) to check the property violation and decide on the action. This data structure encompasses the properties and corresponding actions for each task in the system. For instance, the `MITD_t` structure includes the time limit, dependent task pointer, and the action type to be applied in the event of a property violation.

MITD Example. The monitor checks the elapsed time by using the finish time of the dependent task (Line 14), the current time sent by the runtime via event `e`, and the time limit field (Line 13) in `MITD_t`. If there is time a violation, the action in the `action` field is returned to the runtime.

Maximum Attempt Example. The `MITD_t` structure employs the `max` (Line 16) and `maxAction` (Line 17) fields to

```
 1  // Properties data structure for tasks
 2  typedef struct {
 3      MITD_t mitd[MAX_TASK];
 4      Collect_t col[MAX_TASK];
 5      ReExe_t reExecution;
 6      ExeTime_t executionTime;
 7      Periodic_t periodic;
 8      ...
 9  } property_t;
10
11  // Data structure of each constraint
12  typedef struct {
13      uint64_t timeLimit;
14      task_t *dependentTask;
15      type_action action;
16      uint64_t max; // maximum attempt
17      type_action maxAction; // maximum attempt action
18      ...
19  }MITD_t;
20
21  // monitor interface
22  void callMonitor (MonitorEvent e){
23      _begin // for ImmortalThreads
24          if(e.kind == startTask){
25              check_MITD_const(e.task);
26              ...
27              update_Exetime_const(e.task);
28          }else if (e.kind == EndTask){
29              check_Exetime_const(e.task);
30              update_MITD_const(e.task);
31              ...
32          }
33      _end // for ImmortalThreads
34  }
```

Figure 10. Generated monitor (simplified for readability).

check the MITD property, considering a number of attempts determined by the user (*maxAttempt* in Table 1). As depicted in Figure 5, the monitor monitors the consecutive attempts for the MITD property. If the MITD property remains unsatisfied after three attempts, the monitor sends the `skipPath` action (specified in the `maxAction` field, Line 17) to the runtime to prevent non-termination.

Other Properties. Similarly, the monitor keeps required parameters in the `property_t` data structure.

4.2.2 Extending Properties.

We present a scenario wherein the property specification language undergoes expansion to accommodate a new property type, analyzing its implications on the framework's key components. In intermittent systems, energy awareness is vital for estimating a task's likelihood of uninterrupted completion based on the current capacitor energy level. This awareness can be integrated into ARTEMIS as a novel property, enabling pre-task execution energy level checks and potential task skipping if energy levels are insufficient. Implementation of this extension involves the following steps. Firstly, the ARTEMIS specification language is updated to include the new property type and a built-in primitive for retrieving current energy levels, integrated as new grammar rules. Secondly, the runtime is augmented with a function to access capacitor energy levels, contingent upon suitable hardware support. Subsequently, the property-to-intermediate-language generator is expanded to include a template for the new property. Modifications in the application-specific monitor generator involve translating calls to the new energy awareness primitive into calls to the corresponding runtime function. These changes are straightforward, additive, and non-disruptive, leaving existing application code unchanged. Adopting the Xtext language workbench in ARTEMIS facilitates these modifications with minimal effort.

This hypothetical scenario underscores the importance of the design objectives of our work, which aims to establish a clear separation of responsibilities between application code, runtime, and monitoring code. The proposed extension impacts multiple components, yet the modifications are minor and confined to specific areas. While specialized language engineering expertise is necessary, this constitutes a predominantly one-time endeavor. Moreover, the entire process remains transparent to application developers.

4.2.3 Atomicity and Forward Progress of the Monitor We leveraged the ImmortalThreads library and its C macros to construct monitors that can withstand power failures. These monitors employ a local continuation strategy, enabling them to resume operation from their previous state following a power interruption. This resilience is achieved by storing all monitor variables in nonvolatile memory, ensuring their persistence across power cycles. Consequently, our ImmortalThreads-based monitors can facilitate forward progress without necessitating runtime updates to monitoring variables. In ARTEMIS, task boundaries signify points at which task statuses are updated, ensuring synchronization of all monitoring variables at these junctures. Therefore, the ARTEMIS monitor seamlessly resumes the finalization of invoked monitors without encountering data inconsistencies, even if it is interrupted by a power failure. We illustrate this data consistency assurance using a timestamping example in Section 4.1.3.

5 Evaluation

We compared ARTEMIS with Mayfly [31], a state-of-the-art task-based intermittent computing solution that checks data freshness and collection properties.

Benchmark Application. To assess the performance of ARTEMIS and Mayfly, we used a *wearable health monitoring* application (see Figure 6) tracking human body conditions with the help of several wearable sensors.

Experimental Setup. Figure 11 presents our experimental testbed. We used an MSP430FR5994 [46] microcontroller (MCU) with 256KB of FRAM (Ferroelectric RAM) and 4KB of SRAM. We set the operating frequency of the MCU to 1MHz. We used the Thunderboard EFR32BG22 board [1] as a sensor node comprising all the necessary sensors for the wearable health monitoring application: the temperature sensor for measuring body temperature, the accelerometer and the microphone for breath rate and cough detection, and the BLE 5.0 module for sending data. We used a Powercast TX91501-3W [19] transmitter as a source of RF (radio frequency) energy and a P2110 [20] receiver to harvest RF energy and power our boards.

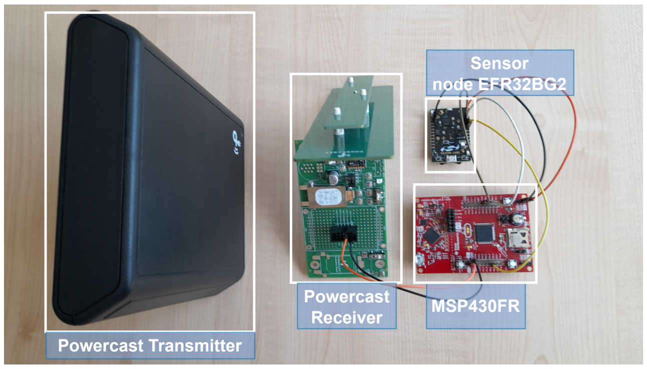


Figure 11. Evaluation setup.

Evaluation Metrics. We considered *five* evaluation metrics: (1) *Runtime and Monitor Overhead* is the overhead load to the system in both time and energy metrics; (2) *Execution Time* represents the time to finish the application while considering the given properties; (3) *Energy Consumption* is the energy consumed to complete a single execution of the application; (4) *Non-termination* refers to conditions that could lead to the non-termination of a task, explicitly concerning energy and time properties; and (5) *Memory Overhead* refers to the extra memory the runtime and monitor occupy.

5.1 Benchmark Application Paths and Properties

Our benchmark consists of three paths for three health indicators (see Figure 6).

Path #1 collects ten body temperature readings and transmits the average. Hence, the `calAvg` task is characterized by the `collect` property. Thus, ARTEMIS restarts the first path until enough samples are collected.

Path #2 is responsible for calculating the respiration rate. According to our measurements, the `accel` task is the highest power-consuming among other tasks. Therefore, we use the `maxTries` property to prevent non-termination. If the power failures occur consecutively ten times within the `accel` task, ARTEMIS skips the path and proceeds to the next one. Thus, ARTEMIS allows the application to complete and transmit the remaining data, even if some data is missing. We have the `MITD` property also in the `accel` and `send` tasks in this path: *the acceleration data must have been collected within the last five minutes when the* `send` *task starts sending data.* If this property is not met, ARTEMIS restarts the respective path. However, since the `send` and `accel` tasks have high energy consumption, it is likely to experience power failures during these tasks. In scenarios where the duration of a

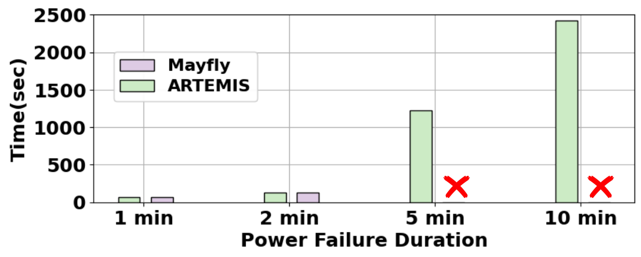


Figure 12. ARTEMIS prevents non-termination when charging time increases.

power failure surpasses the given time constraint, the real-time nature of acceleration data cannot be preserved. Thanks to the `maxAttempt` construct, the path is skipped to ensure the completion of the `send` task and to avoid non-termination if up-to-date data cannot be obtained after three attempts.

Path #3 includes the `maxTries` property for the `micSense` task to prevent non-termination. The `collect` property is also defined between `micSense` and `send` to guarantee the transmission of at least one sample.

5.1.1 Mayfly Version.

Mayfly does not support `maxTries` and `maxAttempt`. Therefore, the Mayfly version of our application solely utilizes the `collect` and `MITD` properties.

5.2 Results on Intermittent Power

Figure 12 depicts the total execution time during intermittent execution with power failure durations (i.e., charging times) ranging from 1 to 10 minutes in our testbed. Under intermittent operation, tasks with peripheral operations, due to their high energy requirements, might be interrupted by power failures frequently. These frequent interruptions might lead to the violation of timely data semantics due to long charging times, leading to application non-termination. For instance, Mayfly restarts path #2 when the time between the `accel` and `send` tasks exceeds five minutes. In this case, the application cannot progress and, in turn, finalize the execution of paths, as the `MITD` property required by the `send` task is never satisfied. Mayfly continues its re-execution attempts of task `accel` to meet the property after each reboot, resulting in a prolonged non-termination state. Therefore, the `send` task will not be executed as long as the `MITD` is not satisfied. As seen in Figure 12, charging durations exceeding 5 minutes led to non-termination since Mayfly could never complete the application execution and deliver beneficial results.

Figure 13 depicts how ARTEMIS prevents non-termination in the benchmark application. The `maxAttempt` construct allows the ARTEMIS runtime to skip paths repeatedly failing. These path skips enable the completion of the application by taking alternative paths given by the programmer. After attempt #3, ARTEMIS skips the path and executes the `send` task, ensuring progress toward completing the application. Hence, ARTEMIS responds to changing conditions and adapts application execution.

5.3 Results on Time Overheads

To assess the time overheads incurred by ARTEMIS and Mayfly during the execution of our benchmark application, we used a continuously powered setup. With continuous power, all timing properties of our benchmark are met during task transitions in both Mayfly and ARTEMIS. Thus, the task execution flow is identical without property violations, leading to a fair and repeatable comparison.

Figure 14 provides a comprehensive overview of the execution time of our benchmark application and the associated overheads. The comparison focuses on the execution time of the application logic and the minimal, acceptable overheads imposed by both ARTEMIS and Mayfly.

ARTEMIS and Mayfly adopt a graph-based programming model for property checking during task transitions. Mayfly focuses on time-related properties and data collection counts between tasks; ARTEMIS checks a broader range of properties through separate monitors and takes actions specified by these monitors in cases of property violations. This design choice of ARTEMIS introduces a slightly higher overhead due to the additional property checking and communication between the runtime and monitors. However, despite this difference, the overall execution times of both systems remain nearly identical.

Figure 15 presents a more detailed breakdown of the overheads. The x-axis is scaled in milliseconds for a finer resolution than the seconds scale in Figure 14. In this detailed view, ARTEMIS incurs additional overhead compared to Mayfly, primarily due to its thorough property checking and the separation of monitoring logic from the application. However, these overheads are still deemed negligible. The distinction in runtime and monitoring overheads reflects the design of ARTEMIS, which enhances reactivity and flexibility. This design choice empowers programmers to tackle power failures with a diverse set of solutions, making ARTEMIS an adaptable and robust system for various scenarios.

5.4 Results on Energy Consumption

Figure 16 depicts the energy consumption in ARTEMIS and Mayfly to complete the application for a single run in continuous and intermittent execution scenarios with different charging delays. In conditions of continuous power and intermittent execution with one and two minutes of charging delay, ARTEMIS and Mayfly demonstrate similar energy consumption in completing the application. This parity is as anticipated, given that the benchmark's timing requirements are satisfied, and the task sequence remains unchanged.

During intermittent execution with longer charging delays, Mayfly exhausts its stored energy by repeatedly executing the `accel` task to meet the `MITD` property between the `accel` and `send` tasks. Consequently, the application's energy demands become effectively unbounded as it continuously consumes energy without completing. Conversely,

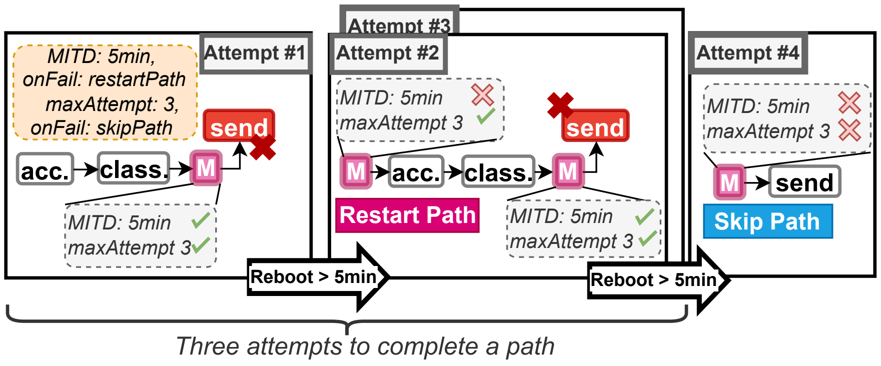


Figure 13. ARTEMIS prevents non-termination by using `maxAttempt` construct.

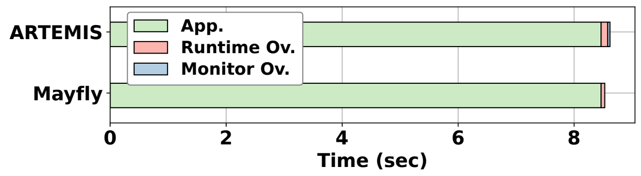


Figure 14. Overheads introduced by ARTEMIS and Mayfly during application execution on continuous power.

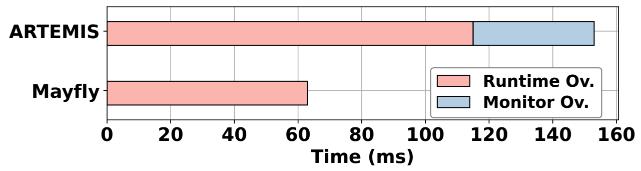


Figure 15. A detailed view of overheads in Figure 14. Note that the x-axis scale of this figure is in **milliseconds**.

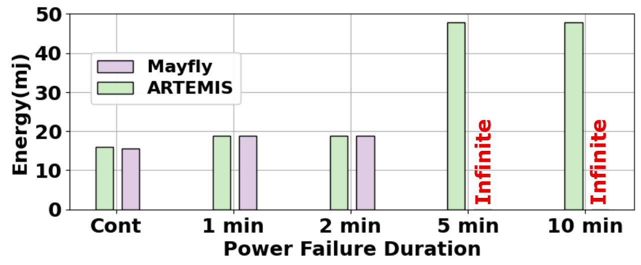


Figure 16. Energy consumption of ARTEMIS and Mayfly for continuously and intermittently powered setups.

Table 2. Memory Requirements (in Bytes).

Mayfly			ARTEMIS					
Runtime			Runtime			Monitor		
.text	*RAM*	*FRAM*	*.text*	*RAM*	*FRAM*	*.text*	*RAM*	*FRAM*
1152	2	6354	1512	2	4756	4644	0	15856

ARTEMIS utilizes the `maxAttempt` construct to bypass the path after three failed attempts. This results in an energy consumption three times higher compared to continuous execution to finish the application.

5.5 Results on Memory Requirements

Table 2 shows the code size (`.text` segment) and RAM and FRAM requirements of the benchmark application. Compared to Mayfly runtime, ARTEMIS runtime requires less FRAM since it has separate runtime and monitor components. Additional memory overhead was incurred due to the monitors generated for the benchmark application. Note that monitors include data structures and variables stored in FRAM to track properties and change the task flow to prevent property violations, which are application-specific. In our current implementation, this memory cost remains within an acceptable range without performing any optimization.

6 Prior Works and Our Differences

Table 3 compares key characteristics of our approach with the most relevant studies to our work, notably TICS [35], InK [48], Mayfly [31], and ImmortalThreads [50]. These studies mainly focus on the timing properties of intermittent programs. Mayfly [31], for instance, focuses on task-based systems with a data freshness consideration. It features a language to express data expiration during task transitions and a runtime that checks if the data consumed by the current task is still fresh. InK [48] also provides data freshness checks at runtime. TICS [35] is a checkpoint-based system that uses source-code annotations to enforce time consistency during intermittent execution, inserting code to check data expiration and ensure timely operations. ImmortalThreads [50] follows a similar approach in this context.

Our Novelty. To be brief, all mentioned solutions propose intermittent computing runtimes that keep track of only a specific property of intermittent systems and detect violations regarding this property, e.g., mainly time consistency violations. They do not provide a modular architecture; they support limited runtime adaptation. Contrary to existing approaches, ARTEMIS enables the specification of complex properties based on a fixed set of primitives reflecting the execution of a task-based application. It separates monitoring logic from the runtime dealing with intermittent execution.

Table 3. Comparison of the main features of ARTEMIS against state-of-the-art solutions.

Prior Art	Property Specification	Property Checking	Runtime Adaptation
DINO [37], Chain [15], Alpaca [38], HarvOS [11], Chinchilla [39], Coati [44],	No language constructs	Explicitly by programmer	Explicitly by programmer
Capybara [16]	No language constructs	*Non-termination* checked by *compiler*	Compile-time solution (NA)
Etap [26]	No language constructs	*Timeliness* checked by *analysis tool*	Compile-time solution (NA)
Mayfly [31]	Limited support for temporal properties	*Runtime* checks (only *expiration*)	*Runtime* restarts task-graph
InK [48]	Limited support for temporal properties	*Runtime* checks (only *expiration*)	*Runtime* evicts thread upon expiration
TICS [35]	Limited support for temporal properties	*Runtime* checks (only *expiration*)	*Runtime* executes programmer-specified code upon expiration
ImmortalThreads [50]	Limited support for temporal properties	*Runtime* checks (only *expiration*)	*Runtime* evicts thread upon expiration
ARTEMIS	*Seperation of property specification from application:* **Extendable** and **separate** language supports **several properties**. Other property specification languages can also be mapped to the intermediate language.	*Seperation of monitoring module from runtime:* **Automatically generated application-specific** monitors check properties. These monitors are generated as a separate module interacting with the runtime using generic interfaces.	*Seperation of monitoring module from runtime:* **Monitors** interact with runtime to trigger **programmer-specified** actions. These actions enable intermittent systems to respond effectively to energy disruptions and varying operating conditions.

Moreover, it automatically generates monitor code interacting with the runtime module using a generic interface, which allows scalability with minimal programmer effort.

Runtime monitoring is an established field with diverse approaches, languages, and tools [10, 28]. While it primarily focuses on evaluating properties based on system observations, our work goes further by defining specific actions for failures. ARTEMIS addresses challenges specific to ambient-powered devices with unique energy collection methods. To our knowledge, no other approaches apply runtime monitoring techniques in intermittent computing. ARTEMIS utilizes state-based specifications as an intermediate language, similar to Larva [18] and Comma [36]. Unlike traditional runtime monitoring approaches well-suited to continuously powered systems, ARTEMIS addresses the inherent challenges of intermittent computing by considering the power-failure resilience of monitors. Furthermore, ARTEMIS explores possible actions to be taken when monitoring reveals property violations. This adaptability is crucial in intermittent computing, where energy disruptions can occur at any moment.

Our approach aligns with the principles of Aspect-Oriented Programming (AOP) [33]. AOP identifies code entangled across multiple modules within the main program logic, referred to as crosscutting concerns, and extracts them into separate modules called aspects. These aspects are then woven into the main code using specialized tools. In our approach, the properties related to the intermittent behavior are treated as aspects and require a separate specification. Unlike AOP, however, ARTEMIS does not merge the generated monitor code with the application code. Nevertheless, in theory, this approach can also be applied to integrate the two. Such integration could lead to a larger memory footprint, which is problematic for resource-constrained edge devices where ARTEMIS is intended to operate efficiently. For instance, when properties need to be monitored at multiple points in the application, duplicating the monitoring code in various places can result in larger memory usage (the same code for monitoring properties may need to be repeated in multiple parts of the application). Separating the monitoring code from the application code helps manage these memory concerns efficiently, ensuring that ARTEMIS remains practical for its intended use cases.

7 Discussion and Future Work

Power Failure-Resilient Runtime. We discussed the generation of power failure-resilient monitors in Section 4.2. The ARTEMIS runtime can be a subject of the same consideration: in case of power failure during the runtime operation, do we have guarantees that the execution of the runtime, application tasks, and monitors will continue correctly after the system restart? Rigorously demonstrating these guarantees and possibly formalizing the interaction among runtime, tasks, and monitors is a point for future work.

Enhanced Modularity. ARTEMIS enables the separation of the property specification from the application code. This approach allows for a more organized and clear code structure, facilitating easier management and maintenance of intermittent systems. By decoupling property descriptions, ARTEMIS promotes code reusability and facilitates the addition, modification, or removal of properties without impacting the underlying application logic. This modularity simplifies system development and enhances code maintainability.

Usability of Property Specification Language. The support for modularity frees the developer from handling the complex task of integrating the application logic and monitoring logic. However, it comes with the price of learning the ARTEMIS property language. We believe that this is not a serious obstacle. The examples in the paper demonstrate that the property specifications are simple, expressed in compact syntax, and very close to annotation-based approaches.

Adaptive System Execution. One advantage of ARTEMIS is its support for adaptive runtime monitoring. The property specification language enables dynamic adaptation of intermittent systems' execution based on properties monitored, such as timing behavior or task executions. This adaptive execution capability enables intermittent systems to respond effectively to energy disruptions and varying operating conditions. ARTEMIS empowers developers to create systems that can autonomously adjust their execution strategies, ensuring reliable and efficient execution in the face of changing energy availability or other runtime factors.

Abstraction and Simplification. The intermediate language provides an abstraction layer between the ARTEMIS property specification language and the target C code. This abstraction enables the representation of the desired properties in a higher-level language closer to the problem domain. It simplifies the translation process by capturing the essential semantics and concepts of the property specifications in a more concise and manageable form.

Support for Other Languages. The intermediate language can play a pivotal role in supporting other property specification languages (e.g., Mayfly [31]) in ARTEMIS. By leveraging model-to-model transformations [21], we can map the constructs and semantics of diverse specification languages to the common intermediate language. This mapping enables the utilization of multiple specification languages, each tailored to specific properties, while still benefiting from the unified representation provided by the intermediate language. Developers can choose the most suitable language for their needs while maintaining compatibility with ARTEMIS and its code-generation capabilities.

Property Consistency Checking. Generally, we admit the possibility that the simultaneous use of time-related properties such as periodicity, maximum duration, and inter-task delays may lead to inconsistent specification. Here, inconsistency means that there is no sequence of task executions that satisfies all constraints. Consistency checking of property specifications is a non-trivial future work. We envisage translating constraints to time-aware models that allow model checking with state-of-the-art tools. This translation can ensure that specifications accurately capture the desired properties of the intermittent system to be monitored.

Implementation Alternatives. Several different implementation alternatives merit examination as future work. One approach that brings performance gains is compiler instrumentation that automatically inlines the monitoring code within the application and runtime. Inlining enables a more tightly integrated system by eliminating calls between different modules for property checking. However, this implementation comes at the cost of an increased memory footprint, as mentioned in Section 6. Another approach can be deploying monitors onto external devices that communicate wirelessly with the device that executes the application to be monitored. Wireless communication is way more energy-hungry compared to computation, which can result in significant overheads when observing events and providing feedback. However, employing external monitors offers modularity and facilitates the deployment of new monitors and adapting existing ones directly on external devices without recompiling and deploying the application and runtime. Evaluating these alternatives (and probably others) involves a trade-off between resource utilization, code maintainability, and performance efficiency, emphasizing the need for a carefully tailored solution that aligns with unique constraints and requirements of intermittent computing scenarios.

8 Conclusion

We introduced ARTEMIS, a novel runtime and monitoring framework for intermittent systems. Utilizing a dedicated property specification notation and having separate runtime and monitor components in ARTEMIS enhance its flexibility and scalability. It offers a robust and adaptable solution that addresses non-termination conditions and incorporates specific actions for property violations. In our experiments, ARTEMIS demonstrated comparable efficiency to a state-of-the-art solution, while its unique features provided distinct advantages in preventing non-termination scenarios. Future work could focus on minimizing further the runtime and monitoring overhead while exploring its additional applications where intermittent computing is prevalent.

Acknowledgments

Funded by the European Union

This work is co-funded by the European Commission's HEU and H2020 Programmes under grant agreements 101070537 (CROSSCON), 101095634 (ENTRUST), 101135576 (INTEND), and 101120657 (ENFIELD). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Efr32bg22 thunderboard kit. https://www.silabs.com/development-tools/thunderboard/thunderboard-bg22-kit?tab=overview, June 2023. Last accessed: June. 29, 2023.

[2] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. The signpost platform for city-scale sensing. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 188–199. IEEE, 2018.

[3] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, et al. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 368–381, 2020.

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

[5] 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.

[6] Anonymous. ARTEMIS Repository. https://, 2023.

[7] 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.

[8] 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.

[9] 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.

[10] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.

[11] 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.

[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] Jongouk Choi, Hyunwoo Joe, and Changhee Jung. Capos: Capacitor error resilience for energy harvesting systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4539–4550, 2022.

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

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

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

[17] 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.

[18] Christian Colombo and Gordon J. Pace. Runtime verification using LARVA. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, volume 3 of *Kalpa Publications in Computing*, pages 55–63. EasyChair, 2017.

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

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

[21] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[22] 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.

[23] Brad Denby, Emily Ruppel, Vaibhav Singh, Shize Che, Chad Taylor, Fayyaz Zaidi, Swarun Kumar, Zac Manchester, and Brandon Lucia. Tartan artibeus: A batteryless, computational satellite research platform. 2022.

[24] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. Camaroptera: A long-range image sensor with local inference for remote sensing applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(3):1–25, 2022.

[25] Eladio Domı, Beatriz Pérez, Ángel L Rubio, et al. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54(10):1045–1066, 2012.

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

[27] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.

[28] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.

[29] Arda Goknil and Kasim Sinan Yildirim. Toward sustainable iot applications: Unique challenges for programming the batteryless edge. *IEEE Software*, 39(5):92–100, 2022.

[30] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 246–257. IEEE, 1996.

[31] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[32] Natsuki Ikeda, Ryo Shigeta, Junichiro Shiomi, and Yoshihiro Kawahara. Soil-monitoring sensor powered by temperature difference between air and shallow underground soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(1):1–22, 2020.

[33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[34] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. Wario: efficient code generation for intermittent computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 777–791, 2022.

[35] 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.

[36] Ivan Kurtev, Jozef Hooman, and Mathijs Schuts. Runtime monitoring based on interface specifications. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 335–356.

Springer, 2017.

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

[38] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *arXiv preprint arXiv:1909.06951*, 2019.

[39] 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.

[40] 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.

[41] 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.

[42] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J Berre. Toward standardised model to text transformations. In *Model Driven Architecture–Foundations and Applications: First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005. Proceedings 1*, pages 239–253. Springer, 2005.

[43] 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.

[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] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[46] 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.

[47] 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.

[48] 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.

[49] Eren Yildiz, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasim Sinan Yildirim. Efficient and safe i/o operations for intermittent systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 63–78, 2023.

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

[51] Eren Yıldız and Kasım Sinan Yıldırım. Persistent timekeeping using harvested power measurements. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 572–574, 2021.

A Artifact Appendix

A.1 Abstract

ARTEMIS is the first framework designed to facilitate flexible property checking of intermittent programs at runtime. ARTEMIS is developed based on techniques from the area of runtime monitoring, offers a specification language for specifying an open set of properties, and provides automatic generation of monitors responsible for checking the properties. In ARTEMIS, developers use a language to define desired properties of the intermittent system, focusing on system properties rather than implementation specifics.

A.2 Description & Requirements

A.2.1 How to access All the project source codes are available in the following Github repository: https://github.com/tinysystems/ARTEMIS. The artifact evaluation materials are reachable from d87e02906c79c6e04c84e0d6c25af5f59d7559e6 commit.

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

A.2.3 Software dependencies ARTEMIS Languages and Code Generator can be loaded and used in Eclipse, distribution Eclipse IDE for Java and DSL Developers. Once the projects are loaded in the Eclipse workspace they can be built as any regular Xtext project and used in a runtime Eclipse instance. The provided example project can be loaded in this runtime instance.

A.3 Set-up

ARTEMIS consists of two main components ARTEMIS Code Generator ("src" folder) and ARTEMIS runtime/monitor (runtime -monitor folder). The repository also contains the example holder which includes a Heath Monitoring App.

A.3.1 ARTEMIS Languages and Code Generator The implementation of the ARTEMIS languages is located in the "src/dsl" folder. There are three languages: Base (provides commonly found constructs), Spec - the ARTEMIS property specification language, and SM - the language for expressing monitors as state machines. All languages are implemented in the Xtext language workbench. The grammars can be found in the "src" folders of projects org.artemis.base, org.artemis.spec and org.artemis.sm. The code generator from property specifications to state machines can be found in project org.artemis.spec, package org.artemis.spec.generator.

A.3.2 ARTEMIS Runtime/Monitor The ARTEMIS runtime and monitor are located in the "runtime-monitor" folder as a static C library. This library includes hardware configuration for MSP430FR5994, as well as ARTEMIS source codes, such as memory configuration MACROS for non-volatile variables (mem.h), and a timekeeping simulator (clock.h). The ARTEMIS runtime's source code and headers can be found in the "libartemis" folder, while the ARTEMIS monitor's source code and headers are located in the "monitor" folder. Additionally, the Artemis C library includes a small version of Immortal Threads in the "ImmortalLib" folder, since the monitor code utilizes Immortal Threads to ensure monitoring against intermittent execution constraints.

A.4 Evaluation workflow

A.4.1 ARTEMIS Example The example is located in folder "examples/ArtemisExample". It contains a model expressed in the ARTEMIS language and is an implementation of the example shown in Figure 5.