Yield over Energy: Task-based Intermittent Computing with Themis

Andrea Maioli[+‡], Matteo Visotto[+‡], Francesco Bertani[+], and Luca Mottola[+*†]
[+]Politecnico di Milano (Italy), [*]RI.SE Computer Science (Sweden), [†]Uppsala University (Sweden)

ABSTRACT

We present Themis: a task-based intermittent programming system able to steer task executions based on programmer-provided directives on the tasks' relative data yield. Unlike existing solutions that solely seek energy conservation, Themis allows programmers to slice the application logic into *task groups* with attached data yield directives. Themis takes these information as an input and regulates the execution of task groups against energy failures and periods of energy surplus. We compare Themis against state-of-the-art task-based intermittent programming systems using five real-world energy traces, measuring how closely each system can match the programmer directives. We show that Themis is up to 31% closer to the programmer directives compared to the best performing baselines and, depending on available energy, can at times even match those requirements perfectly. This performance is achieved while responding to wild energy fluctuations and without requiring manual fine-tuning on a per-scenario basis.

CCS CONCEPTS

• **Computer systems organization → Embedded software**.

KEYWORDS

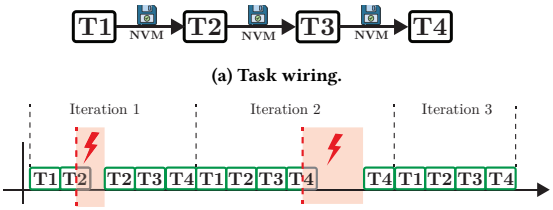
Battery-less Devices, Intermittent Computing, Task Scheduling, Data Yield

1 INTRODUCTION

To abate maintenance costs, ambient energy harvesting replaces traditional batteries to power embedded sensing [2, 7, 15, 21]. Energy from the environment is erratic, causing frequent and unanticipated energy failures leading to an *intermittent computing pattern* [3]. Systems alternate periods of active operation with periods of recharging energy buffers.

Problem. Intermittent computing applications are often encoded as sequences of computational units called *tasks*, which execute with transactional semantics. Fig. 1a shows an example. To cross energy failures, tasks persist their output on non-volatile memory (NVM). Most existing systems [14, 20, 22, 24] execute tasks opportunistically, as energy is available. Fig. 1b illustrates a possible execution of the tasks in Fig. 1a, subject to two energy failures.

In existing systems, as described in Sec. 2, energy conservation is both the primary design goal and sole performance metric [3], *irrespective of application requirements*. Users, however, typically value different types of sensor data differently [2, 6]; for example, sensing of environmental quantities is often crucial compared to monitoring of system parameters [2, 6]. As energy coming from the ambient fluctuates unpredictably, users wish to express the relative importance of the different data the system may produce, and let the system distributed available energy accordingly.



(a) Task wiring.

(b) Example schedule.

Figure 1: Task example.

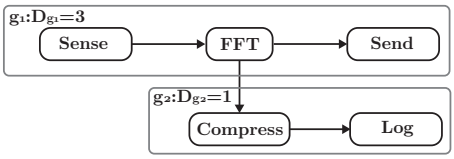


Figure 2: Example application in Themis.

Existing systems generally fail to capture how different slices of an application logic may bear a different value to the users. In cases of excess energy, for example, existing designs would equally (and unnecessarily) increase the execution rate of all tasks, regardless of how valuable they are.

Yield over energy. We argue for a fundamental shift in the design of intermittent computing systems. Rather than merely aiming at conserving energy, we seek to let programmers *provide directives about the relative data yield of different slices of the application logic*, and design the underlying run-time system to take those into account. *Data is what ultimately matters*, as it enables the analysis and control of real-world phenomena [2, 9, 10, 21]. However, *not all data is equal*: letting users express the relative ratios of the amount of different data the system should produce provides a way to differentiate the application functionality.

We make this argument concrete with Themis, a task-based programming model and run-time support for intermittent computing. Using a notion of *task group*, Themis allows programmers to slice the application logic and use this as the basis to attach directives on their relative data yield compared to other groups. Consider Fig. 2: the application defines two task groups. Task group g_1 implements the primary data processing pipeline for the application. The `sense` task samples a sensor, the `FFT` task computes the FFT signature of the readings over a sliding window, and the `transmit` task sends the output over the radio. Task group g_2 implements additional functionality to `compress` the FFT values and `log` them, to be used as a backup. In this example, users are primarily interested in data

[‡]Co-primary authors.

from the wireless channel: they define the relative data yield ratios as $D_{g_1} = 3$ and $D_{g_2} = 1$. Themis accordingly regulates task executions: in the long run, if g_1 executes n times, g_2 runs $\frac{n}{3}$ times.

A group's data yield ratio generally represents the relative number of desired iterations compared to other task groups. Domain experts set these values based on the relative importance of the data each task group generates. These decisions may rely on deployment-specific considerations [2, 16, 20, 22, 35] or may be based on the energy profile of task implementations [18]. Note that data yield directives do not map to temporal deadlines, as deadline awareness is not a design objective in Themis. Due to the energy dynamics, the concept of real-time deadlines is challenging to cater for in intermittent computing, yet prior work exists [22, 25, 35] that provides deadline-driven run-time designs. Differently, we place data yield at the core of the run-time logic, prioritizing the amount of data produced over timely execution.

To ensure that data yield directives are matched during execution, the system should carefully select when to execute what task group. The Themis run-time system, illustrated in Sec. 4, takes as input tasks, task groups, their interconnections, and their data yield ratios. It measures the dynamic execution of every task group and selectively executes tasks within groups to match the required data yield ratios.

Benefits and performance. We compare Themis with two task-based programming systems for intermittent computing: InK [35] and the Energy-Aware (EA) [33] task scheduler. We use three benchmark applications representing real-world deployments using publicly-available implementations [2, 5, 36], and five vastly-different energy traces gathered from a variety of sources [16, 30]. Our benchmarks are not deadline-driven, modeling those real-world scenarios where strict temporal deadlines are not required, differently from scenarios where timely execution is key instead [25, 35].

In Sec. 5, we show that Themis outperforms both baselines across all applications, in that it is up 31% closer to the required data yield ratios compared to the best performing baselines. Although perfectly matching the data yield requirements is generally difficult due to unpredictable energy patterns, Themis can sometimes even match those precisely. In contrast, while InK is essentially bound to static task execution schedules that are oblivious to programmer requirements and energy fluctuations, EA often misallocates energy, prioritizing less relevant task groups.

2 BACKGROUND AND RELATED WORK

Task-based programming is prevalent in intermittent computing [2, 14, 16, 20, 22, 24, 33, 35]. Due to resource constraints [3], most existing systems determine task schedules at compile time. The few existing run-time schedulers [33, 35] rely on time constraints to perform scheduling decision. In contrast, Themis does not take into account any task deadline or time constraints, and determines scheduling decisions only based on current data yield of task groups.

InK [35] focuses on task execution in response to runtime events. Programmers group tasks into threads, similar to task groups. InK executes threads in response to events, for example, interrupts or timers; the execution remains sequential once triggered within a thread. Unlike Themis, InK cannot adapt task iterations in response

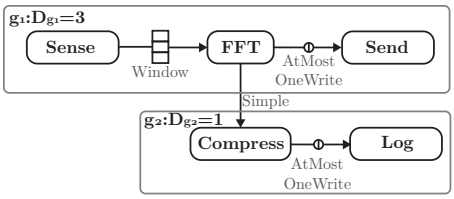


Figure 3: Data pipes in an example Themis application.

to fluctuations in energy availability. Scheduling decisions are confined to the moment a thread is triggered; thus, even with increased energy input, the system would not exploit the surplus.

The Energy-Aware Task Scheduler (EA) [33] organizes tasks into applications, similar to task groups, which are possibly given a priority and a deadline. The scheduler selects the highest-priority application and checks whether available energy is sufficient to complete its execution. That occurs only if energy is sufficient; otherwise, the system enters a low power mode for a fixed period. Accumulated delays due to prolonged periods with insufficient energy may cause the task to miss its deadline. Compared to Themis, we have no notion of task deadline and use energy as soon as it is available, while matching the desired data yield ratios.

MayFly [20] lets programmers define task sequences based on a notion of data lifetime. Compile-time scheduling is used as a basis; at runtime, task execution may be skipped if the required data is not valid or not yet available. In contrast, Themis takes task scheduling decisions entirely at run-time, matching desired per-group data yield ratios.

Celebi [22] lets programmers associate tasks with a real-time deadline. The scheduler, using earliest deadline first [34], inserts recharging periods between task executions to ensure they meet their deadlines. Unlike our work that seeks to target data yield ratios, Celebi targets the timely completion of individual task iterations. By focusing on per-task deadlines, Celebi enforces a fixed-rate data delivery regime, even under conditions of energy surplus.

3 PROGRAMMING WITH THEMIS

We describe the key abstractions in Themis and the corresponding compiler pipeline.

3.1 Groups and Pipes

Tasks groups are subsets of tasks representing units of execution. Programmers use task groups to structure the application logic, similar to modularization abstractions [23]. Based on application requirements [2, 16, 22, 35] and possibly with the help of energy profiling [18], domain experts set the relative data yield requirements for each task group.

As data exchanges across tasks are seldom encoded as a simple handover of single data items, we also provide re-usable *data pipes* enabling data processing between tasks.

Example. Consider again the example of Fig. 2. Programmers define two task groups to separate primary application processing from non-critical functionality. Within the primary group, the **sense** task must provide multiple outputs for the **FFT** task to execute, as the latter should compute over a sliding window of sensor readings. In principle, the sliding window logic belongs neither to

the **sense** nor to the **FFT** tasks, as it bridges the two. Existing systems force programmers to work around the problem. One option is to embed the sliding window logic within either task, essentially polluting either implementation and limiting re-use. A different option would be to create an "adapter" task sitting in between the **sense** and **FFT** tasks, whose only goal is to handle the logic of the sliding window, increasing overhead.

Unlike systems where task wiring cannot encode any data processing or is limited to persisting data [14, 24, 35], we give programmers a catalog of re-usable *data pipes*. Fig. 3 depicts the complete Themis task configuration for the example of Fig. 2. Programmers insert a properly parameterized Window pipe between the **sense** and **FFT** tasks. Besides providing data persistency, the role of pipes in Themis is twofold: they facilitate task coordination and synchronization, and provide additional inputs to the Themis scheduler. Their semantics, in fact, must be taken into account to determine what task can execute, as illustrated in Sec. 4.

Design. Fig. 4 shows the pipes we design based on existing deployments of intermittent computing systems [2, 16, 22, 24, 33, 35]. Each pipe only connects two tasks: one upstream producing data, and one downstream consuming it.

The Simple pipe behaves as a queue of size one and provides the same semantics as existing task-based programming systems [14, 20, 22, 24, 35]. Whenever the task upstream produces a new data item, the one in the pipe is overwritten, if any. The task downstream reads the data item in the pipe, if any, *without* emptying the pipe. Tasks downstream of a Simple dependency cannot execute unless data is in the pipe, even if energy is available.

The AtMostOneRead pipe ensures that the data produced by the task upstream *cannot* be read multiple times, unlike the Simple pipe. Each execution of the downstream task must receive fresh data. Multiple consecutive executions of the producer are allowed, each overwriting the data possibly in the pipe already. When scheduling task executions, provided enough energy is available, this pipe forces the execution of at least one iteration of the task upstream between two consecutive executions of the task downstream.

The AtMostOneWrite pipe is dual compared to the AtMostOneRead. Each data item output by the task upstream must be consumed by one execution of the task downstream, that is, the data possibly in the pipe cannot be overwritten by a new execution of the upstream task. When scheduling tasks, this yields a perfect interleaving of the two tasks.

The Fifo pipe implements the semantics of a FIFO queue of a given size. The task upstream pushes data items in the queue; the one downstream reads *and removes* them from the same queue. Any interleaving of executions of the task upstream or downstream is feasible, as long as the number of executions of the task upstream is equal or greater than the downstream's one. If not, the task downstream cannot run since the queue is necessarily empty.

The Window pipe implements the semantics of a sliding window. To fill the window, we force the execution of a given number of iterations of the task upstream before enabling the one downstream. Each access to the pipe by the task downstream then occurs with the same semantics as in the AtMostOneRead pipe.

It may also be necessary for a task not to wait for data from a specific pipe, for example, when a task processes data available from either of two tasks upstream. To support this, programmers may tag pipes as *optional* for the task downstream, which overrides all cases above.

3.2 Compiler Pipeline

Fig. 5 illustrates the compiler pipeline in Themis. Programmers encode tasks in C and define their wiring through a separate configuration file, along with the definition of task groups, their data yield ratios, and the configuration parameters we explain in Sec. 4. They also provide a *platform specification*, which includes information on the on-board capacitor(s), the target MCU and connected peripherals along with their energy figures, for example, as found in datasheets.

In step 1 of Fig. 5, we measure the worst-case energy consumption of every task. We obtain those values from real hardware or via a static analysis in a simulated environment [13, 27]. This quantity, combined with information on capacitor sizes, determines the *capacitor activation threshold* for each task t, called $V_{on}(t)$. This is the minimum voltage reading of a capacitor to complete task t in the worst case. We use this value to update the MCU activation threshold at runtime, depending on scheduler status.

In step 2 of Fig. 5, the Themis *builder* takes as input the task code and group definitions as determined in the configuration file and generates a single C program as output, including the Themis runtime system, explained next. Inlining the whole code also means that the compiler may operate on the entire code at once, unlocking further compile-time optimizations. The last step triggers a regular C compiler to obtain the binary for deployment.

4 RUN-TIME SUPPORT

Central to the run-time operation of Themis is handling the execution of task groups while matching the programmer-defined data yield ratios.

4.1 Mitigating Energy Fluctuations

Determining at run-time whether task group executions match the required data yield ratios is non-trivial. Energy fluctuations and failures happen unpredictably, while recharging times are not quantifiable a priori, making it difficult to forecast how often task groups may execute. Providing absolute guarantees is thus generally impossible.

Observation window. The system should observe how task group executions unfold over time as a function of the actual energy patterns, and dynamically take scheduling decisions to push the system operation as close as possible to the required data yield ratios. The key is thus to decide what time boundaries to consider to observe executions, evaluate how many iterations each task group completed, and accordingly compute the data yield ratios until that time.

Time boundaries do not determine when a task group should execute, they define a quantifiable interval where the system observes its execution over time. This information may form the basis for future scheduling decisions.

In Themis, we define these time boundaries as a function of the time to execute the minimum number of iterations of all task groups that match the required data yield ratios *in the absence of energy failures*. We call the latter quantity $T_{\min}$. We can compute $T_{\min}$ at compile-time with the information input to the Themis compiler

	Pipe	Representation	General pattern	Execution example	Sample use
Between any tasks	AtMostOneWrite	T_1 —⊕→ T_2	$(T_1 T_2)^*$	$T_{1,W}(1), T_{2,R}(1), T_{1,W}(2), T_{2,R}(2)$	Packet transmission over radio
	Window	T_1 —▭→ T_2	$(T_1^n T_2)^*$	$T_{1,W}(1), T_{1,W}(2), T_{1,W}(3), T_{2,R}(1,2,3)$	FFT processing
Between task groups only	Simple	T_1 —→ T_2	$(T_1^n T_2^m)^*$	$T_{1,W}(1), T_{1,W}(2), T_{2,R}(2), T_{2,R}(2)$	Process a value, if at all available
	AtMostOneRead	T_1 —//→ T_2	$(T_1^+ T_2?)^*$	$T_{1,W}(1), T_{1,W}(2), T_{2,R}(2), T_{1,W}(3), T_{2,R}(3)$	Process only latest sensed value
	FIFO	T_1 —▭▭→ T_2	$T_1^n T_2^m \quad n \geq m$	$T_{1,W}(1), T_{1,W}(2), T_{2,R}(1), T_{2,R}(2)$	Sensed data buffer

Figure 4: Data pipes in Themis. *We use regular expressions to represent the general pattern. In the execution example, $T_{x,W}(y)$ indicates that task x writes data item y, and $T_{x,R}(y)$ means task x reads data item y.*

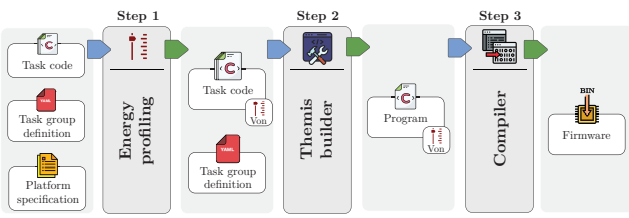
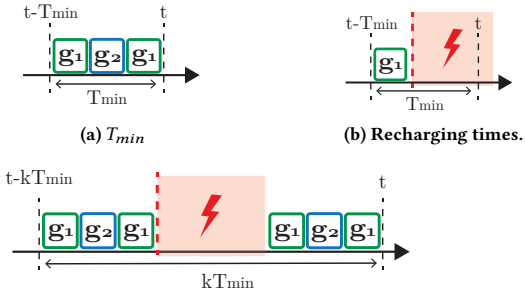


Figure 5: Themis compiler pipeline



(a) T_{min} **(b) Recharging times.**

(c) $k \cdot T_{min}$ **across rechargint times.**

Figure 6: Scheduling window.

pipeline, seen in Sec. 3.2. This time quantity matches exactly the scheduling unit in Themis.

Consider two task groups g_1 and g_2 with $D_{g_1} = 2$, $D_{g_2} = 1$, in the example of Fig. 6. In Fig. 6a, T_{min} is the time needed to perform two iterations of g_1 and one iteration of g_2. Note that any time boundary smaller than T_{min} to monitor executions would not be informative, as the scheduling unit is always the task group. At the same time, monitoring executions only within $T_{\min}$ may be insufficient as well in the general case: if an energy failure occurs as in Fig. 6b and the device spends time recharging while not processing, the system may not observe even a single complete iteration of all task groups.

To amortize the effect of recharging times, we introduce a scaling factor $k \in \mathbb{N}$ and define an *observation window* as $T = k \cdot T_{\min}$. The larger window allows Themis to collect additional execution samples and to mitigate the effect of recharging times on scheduling decisions. For instance, in Fig. 6c the fraction of time affected by recharging times within the observation window is limited, leaving a sufficient number of complete iterations of all task groups visible to the Themis run-time system to evaluate data yield ratios.

In general, larger values for k makes the system more robust against the effect of recharging times, as a longer execution history smooths short-term fluctuations. This comes at the cost of increased memory usage to retain additional bookkeeping data. On the other hand, as $k \longrightarrow 1$, the observation window shrinks, reducing memory overhead but also decreasing robustness to recharging times due to the fewer samples available for evaluating current data yield ratios.

Avoiding starvation. When energy is scarce, Themis may repeatedly schedule groups with higher required data yield in an attempt to match their target ratios. With frequent energy failures that constantly prevent these groups from achieving the target within a valid observation window, this may lead to starvation of groups with lower required data yield, which may be under executed or not executed at all.

To mitigate this effect, we introduce a second parameter $r \in [0, 1]$. It specifies the minimum fraction of the target data yield that a group must achieve before Themis schedules executions of the next lower-yield group. In essence, r controls how tenaciously Themis tries to match a target data yield before switching to groups with lower required data yield. Larger values of r require a task group to make substantial progress toward its target before execution is granted to other groups with lower required data yield. In contrast, smaller values of r enable earlier execution of groups with lower required data yield, avoiding their starvation.

In Sec. 5, we analyze the combined impact of k and r on runtime behavior under different energy sources.

4.2 Scheduling

Scheduling decisions are taken at group- or task-level.

Task group selection. We use data yield as the primary criterion to ensure proportional execution across task groups. Alternative schemes may result in executions that yield a non-proportional allocation of scheduling decisions, for example, when considering a group's net energy demand compared to available energy. In this case, for example, task groups requiring lower energy to complete would be favored independent of the required data yield ratio. This happens precisely in one of the baselines we discuss in Sec. 5.

To match the programmer-provided data yield ratios, we order task groups in decreasing order of D_g, meaning that the task groups

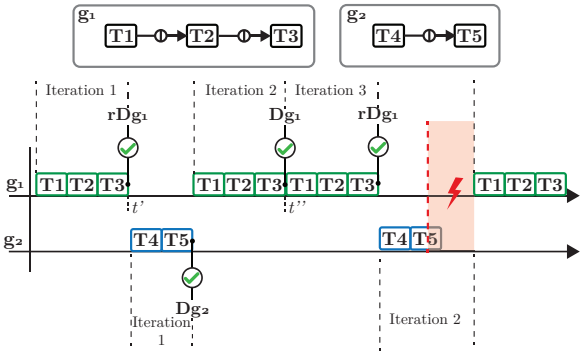


Figure 7: Themis execution example.

with the highest data yield ratios are given higher priority. Consider the example in Fig. 7. The system includes two task groups g_1, g_2 with D_{g_1}=2 and D_{g_2}=1, in addition k is set to 10 and r is set to 0.5. The system starts by selecting g_1 in Fig. 7 and executes that as soon as energy is available. The execution of g_1 eventually completes.

When a task group completes one iteration or when resuming from an energy failure, Themis computes *data yield satisfaction* S_i at time t as

$$S_i(t) = \min \frac{N_i^{(t-kT_{min},t)}}{rD_{g_i}} = \min \frac{N_i^{(t-T,t)}}{rD_{g_i}} \tag{1}$$

that is, the number of executions N_i performed by a group i within the observation window; this value is divided by the required group data yield D_{g_i} and scaled by the factor r. The quantity rD_{g_i} represents the minimum data yield group i must achieve before another group with lower required data yield is allowed to execute.

At time t, Themis evaluates S_i for all task groups and executes the group with the lowest satisfaction value. In the example of Fig. 7 at time $t = t'$, group $S_1(t') = 1$, whereas $S_2(t') = 0$: Themis selects group 2 for execution. As soon as g_2 completes its execution, we evaluate Eq. 1 again and Themis selects g_1. Sufficient energy is available to complete the second iteration of g_1, which now fulfills its data yield ratio requirement. Since no other task group is defined and at time $t = t''$ all groups match the required data yield ratios, Themis restarts by selecting g_1 again. Group g_1 completes and, according to Eq. 1 with $r = 0.5$, Themis selects g_2 for its second execution. During the execution of T_5, an energy failure occurs. When resuming from this failure, the time interval advances, reducing the data yield satisfaction of g_1; as a result, Themis selects g_1 for execution.

Task execution. Whenever a group is selected, task execution happens in a depth-first manner, as in existing systems [33, 35]. Exceptions occur with specific combinations of data pipes, which may introduce dependencies across tasks or task groups.

In principle, the Simple, AtMostOneRead, and FIFO pipes allow repeated executions of the upstream task without ever triggering the downstream task. To prevent starvation of the latter, Themis allows programmers to use these pipes only between tasks of different task groups, decoupling the scheduling of the downstream task from the upstream one. Within the same task group, only AtMostOneWrite and Window pipes are allowed. AtMostOneWrite preserves depth-first execution. Window modifies the execution order by allowing the upstream task to run multiple times.

4.3 Activation Threshold

To facilitate matching the data yield ratios, using hooks provided by intermittent computing platforms [2, 12, 28], we programmatically change the activation threshold for the MCU to resume after an energy failure. This determines both the amount of energy available and the duration of recharging periods. A higher threshold results in longer recharge periods, allowing the system to harvest more energy and potentially reducing energy failures. A lower threshold leads to faster system restarts but with less energy available.

We generally prioritize faster system restarts. This approach allows us to use harvested energy as soon as possible. Available energy after an energy failure, however, should suffice to complete the execution of the selected task(s); otherwise, the system may uselessly consume energy without completing any task executions, ending up in a livelock [13].

Identifying what task executes after an energy failure, however, is difficult: it depends on various factors, including the semantics of pipes. We adopt a *conservative* approach and set the system's activation threshold to the specific $V_{on}(t)$ that grants sufficient energy to execute the most energy-intensive task t in any of the task groups. By construction, this guarantees that sufficient energy is available to complete at least one iteration of *any* task after an energy failure. This value is determined off-line in step 1 of the compiler pipeline.

Overall, Sec. 5 shows that our design decisions yield a run-time overhead that is arguably negligible, and yet we obtain better overall performance than the baselines we consider.

5 EVALUATION

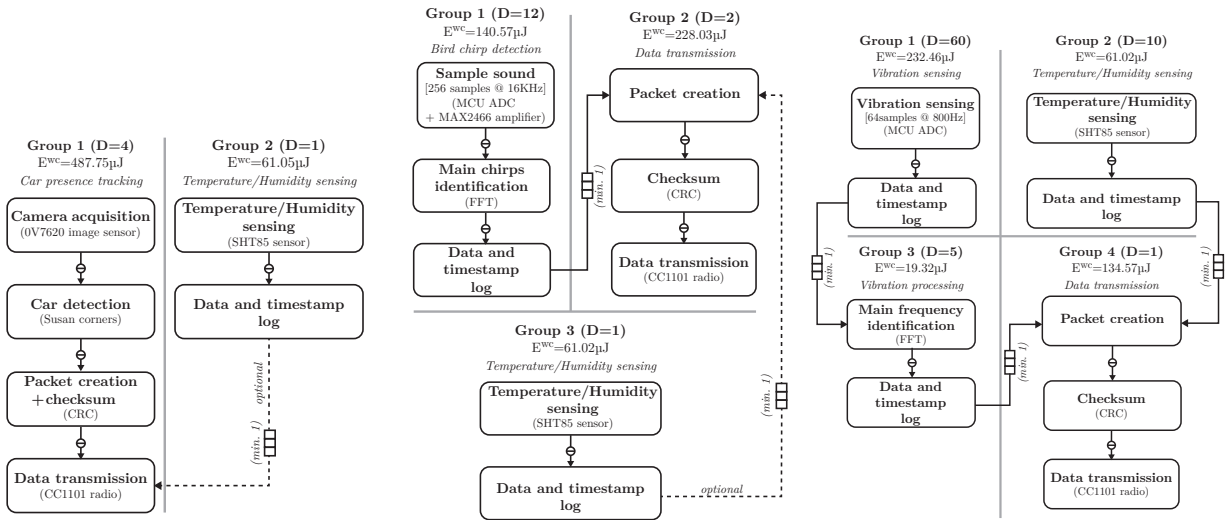
We implement a prototype of the Themis pipeline, depicted in Fig. 5, and run-time support, described in Sec. 4. We implement the run-time support in *C*. We use a Python script along with a series of macros and preprocessor directives to assemble the application code with the run-time support implementation. We compile the code using *clang* v8.0.1.

5.1 Setting

We describe benchmarks, baselines and metrics for comparison, and the measurement setup.

Benchmarks. We consider three staple *sensing applications*, depicted in Fig. 8, widely used in existing literature [4, 8, 14, 24] and also deployed in real-world settings [2, 9, 10, 21]. We build portions of each application using MiBench2 [1], a benchmarking suite commonly used for intermittent computing systems. Tasks in the same task group are connected with a AtMostOneWrite pipe.

The *smart parking* application, shown in Fig. 8a, consists in two task groups, g_1 and g_2 [5]. Group g_1 encodes the main functionality, using the Susan corner recognition algorithm. Next, it creates a data packet with its CRC and transmits it through a radio. Group g_2 contains secondary functionality to sense the environment temperature and humidity and log them. We prioritize data yield for car detection over environment sensing, setting $D_{g_1} = 4$ and $D_{g_2} = 1$. The last tasks of g_1 and g_2 are connected using an *optional* Window



(a) Smart parking (b) Bird tracking (c) Archaeological site monitoring

Figure 8: Benchmark applications.

pipe, which allows the last task in g_1 to opportunistically transmit also temperature and humidity data.

The *bird habit tracking* application, shown in Fig. 8b, includes three task groups, g_1, g_2, and g_3 [36]. Group g_1 identifies the presence of birds by collecting 256 sound samples at $16KHz$ and using FFT to identify the main frequencies [31, 36]. Group g_2 transmits data and g_3 senses environment quantities. We favor bird detection over data transmission and environment sensing, setting $D_{g_1} = 12$, $D_{g_2} = 2$, and $D_{g_3} = 1$. The last task of group g_1 is connected to group g_2 using a Window pipe. Similarly, the last task of group g_3 is connected to group g_2 with an *optional* Window pipe. As before, this allows group g_2 to opportunistically transmit temperature and humidity data when available.

Fig. 8c shows the *archaeological site monitoring* application [2] including four task groups, g_1, g_2, g_3, and g_4. Group g_1 records vibrations data at $800Hz$. Group g_2 measures environment temperature and humidity. Group g_3 extracts the oscillating frequency of previously-collected vibration data using the FFT algorithm. Group g_4 transmits the FFT output, as well as temperature and humidity information. We set $D_{g_1} = 60$, $D_{g_2} = 10$, $D_{g_3} = 5$, and $D_{g_4} = 1$, favoring the collection of vibration data over temperature and humidity sensing [2]. To prioritize collection of vibration data over processing, the last task of group g_1 is connected to the first task of group g_3 using a Window pipe.

For all three application, we run experiments through all possible combination of parameters $k \in \{5; 6; 7; 8; 9; 10; 20\}$ and $r \in \{0.25; 0.50; 0.75; 1.0\}$ and, in addition, we also test $r = 0.33$ for the bird tracking application.

Baselines. We compare Themis with InK [35] and the Energy-Aware (EA) task scheduler [33]. Both are arguably closer to our work than any other system, as seen in Sec. 2.

Using InK, we set timers to trigger thread execution proportionally to the required data yield ratios. Setting the exact timer value, however, is difficult. Short periods may cause InK to execute task threads sequentially, eventually overlooking the requested data yield ratio, or may prevent lower-priority threads from running due to frequent preemption. Longer period may leave the system inactive even if energy is available. Finding a compromise between the two must, anyway, reckon with unpredictable energy patterns.

For the *smart parking* application, we set the timer for g_1 to $15s$, reflecting a reasonable interval for detecting parked cars and transmitting occupancy data. In the other two applications, we set the timer for g_1 to $5s$, as it is meant to collect data from sensors attached to ADCs, requiring more frequent sampling. We set all other timers based on the data yield ratios. Note that this configuration is thus entirely the result of *domain expertise*, if and when available [3].

The EA task scheduler [33], instead, executes tasks that fit available energy and can complete within their deadline, as described in Sec. 2. This baseline allows us to compare Themis against scheduling approaches that primarily target energy conservation.

We use the open-source release of InK [35] and we re-implement the EA scheduler by accurately following the algorithm description [33] due to the lack of an open-source release. Unlike InK and Themis, EA does not support the semantics of Window pipes. We extend its implementation as required in the applications of Fig. 8.

Metrics. Our primary goal is to measure the degree to which the executions match the required data yield ratios. To quantify this, we define *compliance* as a measure of the distance from ideal executions that perfectly match the required data yield ratios. We express compliance in percentage: a 100% value indicates that the system matches the required data yield ratios perfectly. Whenever compliance is lower than 100% but still positive, the system maintains the relative ordering of task group executions, that is, groups with higher D_g still execute more often than those with lower D_g, albeit not exactly matching the requirement. Compliance is negative in the inverse situation, that is, group with higher D_g executes less often than a group with lower D_g.

For a given system execution with M groups, we compute compliance as

$$C = \frac{\sum_{j=1}^{M}(100 - |100 - D_{g_{actual_j}}|) \cdot D_{g_j}}{\sum_{j=1}^{M} D_{g_j}} \quad (2)$$

where $\left(100 - |100 - D_{g_{actual_j}}|\right)$ represents how far, in percentage, is the actual execution of group j from an execution that perfectly matches the required data yield ratio.

The term $D_{g_{actual_i}}$ measures the fraction of executions for group i out of the number of executions for the same group in a situation where all required data yield ratios are perfectly matched. Therefore, intuitively, this represents a measure of how close is the execution of group i to a perfect execution.

We compute $D_{g_{actual_i}}$ as the ratio between the number of times group i executed, over the total number of executions of all groups, and its required data yield ratio, weighted over the data yields required for all groups

$$D_{g_{actual_i}} = \frac{N_i}{\sum_{j=1}^{M} N_j} \cdot \frac{\sum_{j=1}^{M} D_{g_j}}{D_{g_i}} \cdot 100 \quad (3)$$

To compute these quantities, we monitor execution at runtime. For each task group g_i, we record the number of times N_i it executes within the duration of an experiment. This measurement captures the amount of data produced by the system and forms the basis for all the metrics above.

We compute additional metrics to draw a complete picture of Themis performance and to explain relative trends in compliance of Themis against the baselines. Specifically, we count *i)* how many times a task group starts execute but does not complete due to energy failures, *ii)* the number of energy failures recorded within the experiment time, and *iii)* the energy balance in suboptimal task groups executions, that is, how much energy the system uses to execute additional iterations of task groups beyond the required data yield, or the energy the system is lacking to match the requirement.

Tools and traces. Monitoring intermittent computing devices requires specialized hardware and software [11], which likely interferes with the measurements. Further, reproducing energy harvesting sources using real hardware is challenging, as their behavior is non-deterministic [16, 18].

We use ScEpTIC [26, 27], an open-source emulator for intermittent computing, based on its extensive use in literature [28, 29, 32]. ScEpTIC emulates executions at cycle level, along with ambient energy sources and energy consumption. We emulate a TI MSP430-FR5969 MCU, a popular MCU for intermittent computing [4, 24, 30], with a $100\mu F$ capacitor. We model an existing power-on circuit [2, 28] to programmatically set the MCU activation threshold. We model a MAX20361 charge booster to regulate the source voltage to $3.65V$, ensuring capacitor's recharge and MCU activation even in presence of low input voltage. We emulate a time keeper architecture [19] to track time during energy failures.

We also implement models of various peripherals into ScEpTIC, using datasheet information. We consider a Sensirion SHT85 temperature and humidity sensor, a Omnivision OV7620 color image sensor, a MAX4466 microphone pre-amplifier, and a TI CC1101 radio. The sampling of sound and vibration signals relies on the MCU's internal ADC, which ScEpTIC also emulates [26]. Tab. 1 summarizes the energy and time profiles of all task groups.

		Smart parking	Bird tracking	Archaeological site monitoring
g_1	**Operation**	Car tracking	Chirp detection	Vibration sensing
	Energy	$487.75\mu J$	$140.57\mu J$	$232.46\mu J$
	Time	$667.77ms$	$37.86ms$	$78.79ms$
g_2	**Operation**	T/H sensing	Data TX	T/H sensing
	Energy	$61.05\mu J$	$228.03\mu J$	$61.02\mu J$
	Time	$18.53ms$	$5.05ms$	$18.53ms$
g_3	**Operation**	-	T/H sensing	FFT
	Energy	-	$61.02\mu J$	$19.32\mu J$
	Time	-	$18.53ms$	$4.84ms$
g_4	**Operation**	-	-	Data TX
	Energy	-	-	$134.57\mu J$
	Time	-	-	$4.22ms$

Table 1: Energy and time profiles of task groups.

Energy source	Mean power	Duty factor	Gini index
Jogging	35.110 μW	0.011	0.544
Washer	42.383 μW	0.249	0.727
Car	7.099 μW	0.127	0.646
RF	90.189 μW	0.676	0.673
Office	168.982 μW	0.994	0.164

Table 2: Characterization of energy traces.

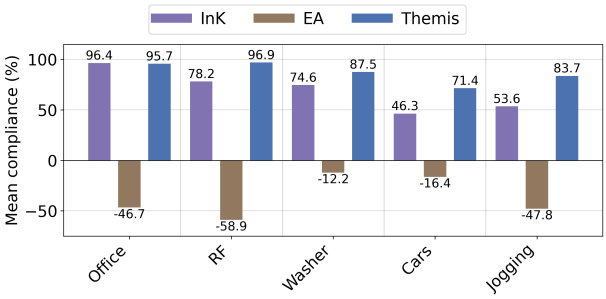


Figure 9: Average compliance.

To evaluate Themis under various energy dynamics, we consider five energy traces. We use the RF trace from Mementos [30] and three kinetic energy traces from Bonito [16], specifically Jogging, Washer, and Car, along with the solar indoor trace from Bonito [16], called Office. These traces expose the systems to a variety of energy dynamics, both in time and energy content, helping generalize results.

As summarized in Tab. 2, we characterize each power trace by computing the Duty Factor and the Gini Index. The Duty Factor DF captures the temporal distribution of power above a threshold θ. It quantifies whether the harvested energy is spread over time or concentrated in a small number of high-power events. A value of $DF \approx 1$ indicates that power is evenly distributed along the trace, whereas $DF \ll 1$ denotes energy concentrated in few short-lived spikes. The Gini Index G [17] quantifies the temporal distribution of power, capturing whether high-power events are uniformly spread over the energy trace or clustered in limited intervals. Values of $G \to 0$ denote an even distribution of power over time, whereas $G \to 1$ indicate that most of the energy is concentrated in a small fraction of the trace.

Note that the Office trace is both rich of energy and rather stable. It produces executions with few energy failures, leading to a

mostly continuous rather than intermittent behavior. We include it in our analysis as a reference point to non-intermittent executions. The RF trace, although producing intermittent behaviors, is shorter than one minute. To prolong the experiments, we re-play the trace over longer periods, which artificially introduces a periodic pattern.

5.2 Results

We run a total of 300 distinct experiments, each emulating 3600s of execution, and collect more than 16M data points that we use to compute the results we discuss next.

Overall performance. Fig. 9 shows the average compliance obtained by the three systems across the three applications, broken down by energy trace. For Themis, we use $k = 10$ and r equal to the inverse of the number of groups; we return later to this setting and discuss the impact of parameters.

Whenever the energy patterns produce an intermittent behavior, Fig. 9 demonstrates how Themis outperforms the other two baselines, steering executions in ways that closely match the required data yield ratios. This is achieved by continuously adapting scheduling decisions at runtime, reacting to energy failures and rescheduling task groups accordingly. This makes group executions align to the required data yield ratios even under highly fluctuating energy sources.

In contrast, InK schedules task threads solely when timers expire. This only allows InK to execute a thread if energy is available whenever the timer fires. When energy is abundant, as in the Office trace, InK matches the required data yield ratios. Performance degrades in situations with higher variability, such as Jogging, Washer, Car, and RF. As indicated in Tab. 2, these traces feature low mean power and few energy spikes ($DF \rightarrow 0$) concentrated in short time intervals ($G \rightarrow 1$). Under such unstable conditions, InK frequently delays or skips the execution of entire groups.

EA behaves the opposite. It schedules tasks based on energy availability, systematically selecting the task with the lowest energy demand when energy is scarce. The ordering of execution is thus often reversed relative to the one dictated by data yield ratios. Across the three applications, the group with the lowest required data yield is also the one that consumes the least energy, as in Tab. 1.

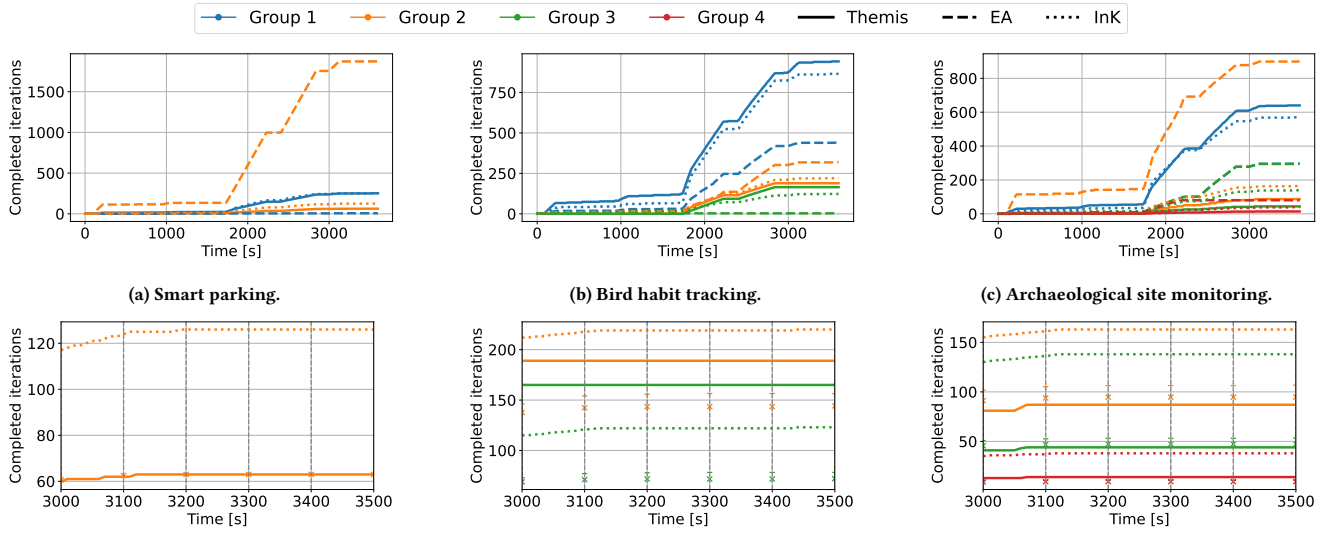
Group iterations over time. Fig. 10 provides a complementary perspective compared to Fig. 9, reporting the number of completed iterations for every group over time. Fig. 10a to Fig. 10c demonstrate that, regardless of the application, EA performs a large number of executions for groups with lower required data yield, often resulting in low or even negative compliance. Instead, Themis and InK show similar patterns. The differences arise from how Themis and InK schedule the execution of groups with lower required data yield.

Fig. 10d to Fig. 10f focus on the last 500s of the experiment, not including the group with highest required data yield. Markers indicate the expected number of group executions in Themis and InK. In the smart parking application, shown in Fig. 10d, we note how the second group in Themis follows the marker; InK executes the same group twice as Themis, exceeding the required data yield ratio. This reflects with the 100% compliance Themis obtains.

In the bird habit tracking application, shown in Fig. 10e, Themis and InK achieve similar performance, since both execute the groups with lower required data yield more frequently than required. A different behavior emerges in the archaeological site monitoring application, shown in Fig. 10f. Themis obtains higher compliance than InK, as it stays closer to the required data yield throughout the experiments also for groups with lower required data yield.

Managing energy. We analyze next how Themis distributes available energy across the existing task groups.

Fig. 11 reports the number of times a task starts execute but that execution is ultimately aborted due to energy failures. This leads to energy waste as tasks run with transactional semantics and restart from the beginning after the failure. The EA scheduler exhibits a high number of interrupted executions, despite its fundamentally energy-conservative design. This stems from EA's tendency to



(a) Smart parking.

(b) Bird habit tracking.

(c) Archaeological site monitoring.

(d) Smart parking - last 500s.

(e) Bird habit tracking - last 500s.

(f) Archaeological site monitoring - last 500s.

Figure 10: Per-group iterations over time.

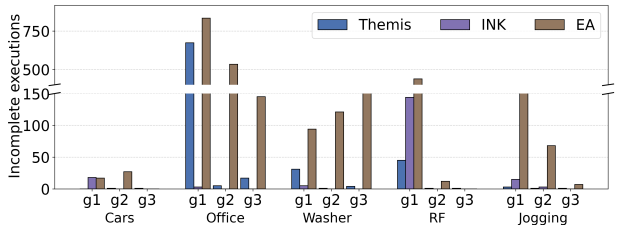


Figure 11: Incomplete executions.

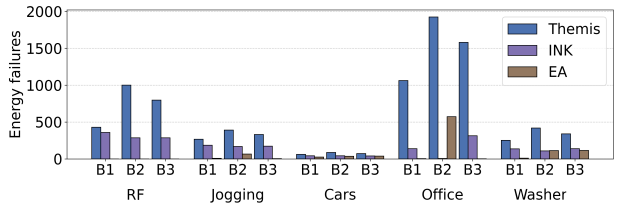


Figure 12: Energy failures.

preempt energy-intensive tasks when energy is scarce and the execution requirements exceed the available energy budget. This leads to group starvation, forcing tasks to restart after extended idle periods or upon subsequent energy failures.

On the other hand, Themis generally exhibits a higher number of interrupted executions than InK. Fig. 11 shows that these interruptions predominantly affect group g_1. We attribute this to Themis 's opportunistic scheduling: upon partially fulfilling the target data yield ratios across groups, it attempts to launch a new iteration of the first group to increase the overall data yield. This strategy may cause a subsequent energy failure and a corresponding group restart, and is intrinsically linked to how Themis dynamically modulates the V_{on} threshold at runtime. The system sets the wake-up voltage to match the maximum energy required for the subsequent task group under worst-case conditions. It may then initiate execution with a reduced energy margin, leading to higher chances of energy failures.

Fig. 12 quantifies the latter effect, reporting the total number of energy failures during the experiment. As expected because of the observations above, Themis consistently records the highest number of energy failures. However, this is not a source of inefficiency; it is a deliberate design trade-off intended to minimize recharging latency by triggering execution when the minimum energy requirement is met.

Despite the higher number of energy failures, this behavior is not indicative of degraded performance. Rather, the higher number of energy failures is merely a a side-effect of Themis successfully driving executions to achieve a data yield significantly closer to the optimal application requirements, as Fig. 9 illustrates.

Fig. 13 offers a complementary perspective on energy distribution, showing the total energy used on task executions that do not directly contribute to the target data yield ratio. We use the archaeological site monitoring application as an example, yet we draw identical conclusions with the other applications. A positive value in Fig. 13 indicates energy wasted on executing more task group iterations than strictly required. A negative value implies an energy deficit, denoting the additional energy the system would need to satisfy the data yield requirements. The plot does not include the

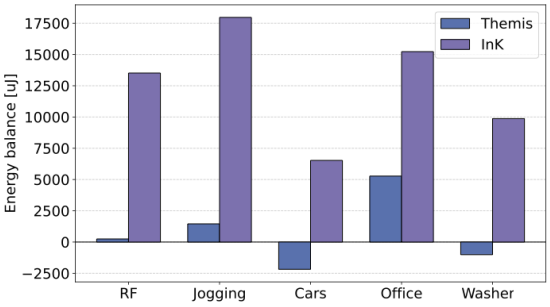
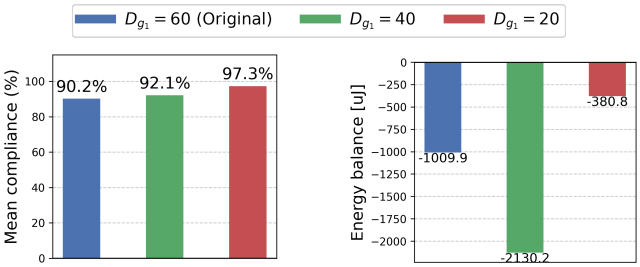


Figure 13: Energy balance.



(a) Compliance. **(b) Energy balance.**

Figure 14: Workload variation.

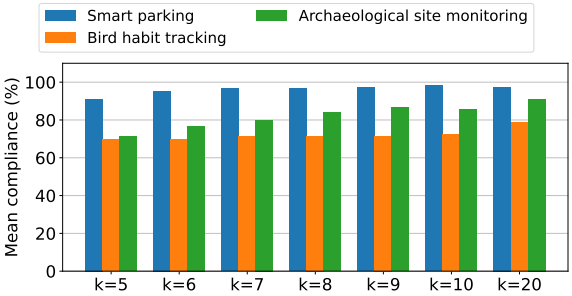


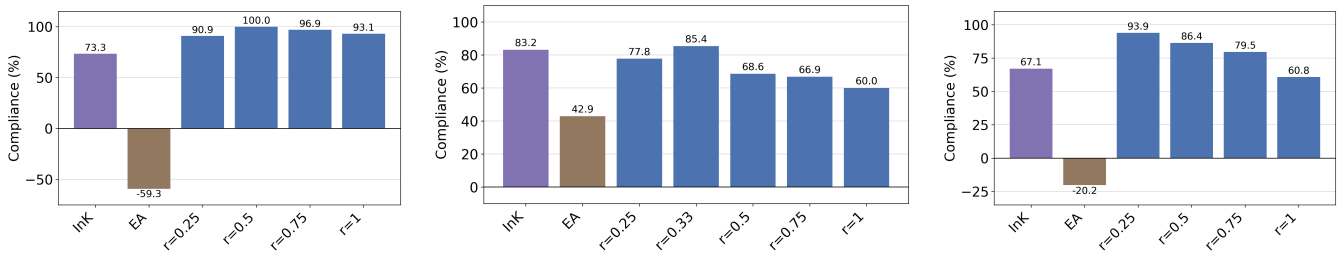
Figure 15: Impact of parameter k.

EA baseline, as its values are orders of magnitude higher compared to Themis and InK.

Despite causing a higher number of energy failures, as discussed earlier, Fig. 13 demonstrates that Themis wastes much less energy compared to InK. This stems from the ability of Themis to enforce strictly proportional execution across distinct task groups. It actively prevents redundant executions that deviate from the specified data yield requirements. This is precisely what steers Themis executions towards higher compliance compared to InK and EA.

Workload variation. We check Themis performance against varying workloads by modulating the data yield requirement of group g_1 in the archaeological site monitoring application. This value defines the mutual proportion across all task groups. Adjusting D_{g_1} thus effectively re-shapes the cumulative workload profile of the application.

The compliance performance is shown in Fig. 14a. Reducing the required data yield of the most demanding group grants Themis greater scheduling flexibility; with fewer iterations required for g_1, the system can better accommodate the remaining tasks, thereby increasing overall compliance. Note that the transition from D_{g_1} = 60 to D_{g_1} = 20 only results in a marginal 7% compliance gain. This



(a) Smart parking **(b) Bird habit tracking** **(c) Archaeological site monitoring**

Figure 16: Performance with varying r.

confirms that Themis performance remains robust across different workloads.

The energy dynamics of the three variants are captured in Fig. 14b, similar to Fig. 13. The configuration with $D_{g_1} = 20$ reaches equilibrium with the lowest energy overhead, whereas $D_{g_1} = 40$ corresponds to the highest energy deficit. This occurs despite the latter achieving higher compliance than the original workload. This seemingly counter-intuitive behavior is a direct consequence of the energy profile of g_4 in this application, which Tab. 1 indicates as the second most energy-intensive group. By relaxing the data yield requirement of g_1, Themis gains room to increase iterations for g_2 and g_3, but that is still insufficient to push g_4 too. Because g_4 has a high per-iteration energy cost but a relatively low impact on overall compliance, its inability to complete even a few executions creates a significant energy imbalance, without significantly penalizing overall compliance.

Parameter settings. Parameter k sets the size of the observation window for group executions, as described in Sec. 4.1. With a small k, Themis requires limited memory for bookkeeping data, but fewer samples complicates monitoring group executions against their data yield, especially when an energy failure occupies part of the observation window. Larger values of k improve the robustness of scheduling decisions to energy failures, with higher memory usage.

Fig. 15 reports the effect of different values of k for the three applications we consider, averaged over the RF, Car, Washer, and Jogging traces. In general, higher values of k lead to higher compliance for Themis. We select $k = 10$ as a default value to strike a balance between accurate scheduling decisions at run-time and memory occupation.

Parameter r, instead, defines when the scheduler switches to a group with lower required data yield, as described in Sec. 4.1. Fig. 16 shows the performance in compliance for different r, using the Washer trace. As reported in Tab. 2, this trace features few power spikes ($DF \rightarrow 0$) concentrated in short time intervals ($G \rightarrow 1$), and is generally poor of energy. It thus represents a case where executions are prone to starvation of groups with lower required data yield.

We find that especially in this situation, setting $r = 1/M$, that is, equal to the inverse of the number of groups, provides the best performance. This value allows Themis to borrow iterations in advance from groups with higher required data yield and assign them to those with lower required data yield, thus preventing their starvation under scarce energy.

This holds more as the number of groups increases.

For example, Fig. 16a reports the results for the smart parking application, which includes two groups. We set $r = 1/M = 0.5$. This value provides the best performance, enabling Themis to reach 100% compliance and performing 26.7% better than InK. Other values of r remain viable. For instance, with $r = 0.75$, compliance is close to the optimal case. The choice of r becomes slightly more delicate in the archaeological site monitoring application, especially in conditions of energy scarcity. For example, the second-best choice of r reduces compliance by more than 7% compared to $r = 1/M$.

Note that the sensitivity of Themis to the values of k and r does not deny the gains in compliance compared to the baselines, shown earlier in Fig. 9 for the case with $k = 10$ and r equal to the inverse of the number of groups. The only substantial difference is with the Office energy trace, where Themis performance in compliance drops to 88.7% when averaged over all values of k or r we test. The performance with all other configurations remains roughly stable also when averaged over the same parameter settings.

This is expected, as Themis is fundamentally designed for energy scarcity and intermittent executions. The features of the Office trace, reported in Tab. 2, leads to a rather continue execution. Normally, Themis would opportunistically consume harvested energy to maintain proportional execution across task groups. Under stable power, however, Themis aggressively scales the task group iterations. Sudden energy failures often catch the system mid-execution in energy-hungry task groups, effectively starving subsequent groups and thus penalizing overall compliance in Themis.

Run-time overhead. Themis overhead depends on the number of task groups M and the number of tasks N_g in each group, resulting in a general time complexity of $O(MN_g)$.

We run experiments for each application starting with a single task group and increasing the number of task groups up to the final configuration. We find that Themis introduces a negligible run-time overhead, lower than 0.05% relative to the energy consumption of the most demanding task group. This includes a fixed cost of 195 clock cycles, $0.1\mu J$, and $12.25\mu s$, independent of the number of task groups, with an additional overhead of 185 clock cycles, $0.11\mu J$, and $11.5\mu s$ per task group. These figures are arguably negligible.

6 CONCLUSION

Themis is a task-based programming systems for intermittent computing that aims to schedule task executions to match programmer-provided data yield requirements. Programmers define data yield ratios over subsets of tasks we call tasks groups. Data pipes with

re-usable functionality connect tasks within and across groups. The run-time system jointly considers data yield requirements and the semantics of data pipes to determine task schedules. Our experimental evaluation compares Themis with EA and InK across three different applications, using real-world energy traces. When compared to EA, Themis ensures in average a 127% higher compliance. In contrast, InK ensures compliance levels closer to Themis, but suffers from high variability in energy-scarce scenarios. This makes InK up to 31% less efficient than Themis in matching the required data yield ratios.

Open-source code. https://bitbucket.org/neslabpolimi/themis

Acknowledgments. This work is supported by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] MiBench2. https://github.com/impedimentToProgress/MiBench2.
[2] M. Afanasov et al. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *SenSys*, 2020.
[3] S. Ahmed et al. The Internet of Batteryless Things. *CACM*, 2024.
[4] D. Balsamo et al. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *ESL*, 2015.
[5] F. Bambusi et al. The case for approximate intermittent computing. In *IPSN*, 2022.
[6] J. Beutel et al. X-sense: Sensing in extreme environments. In *DATE*, 2011.
[7] N. A. Bhatti et al. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *TOSN*, 2016.
[8] N. A. Bhatti and L. Mottola. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *IPSN*, 2017.
[9] Q. Chen et al. Harvest energy from the water: A self-sustained wireless water quality sensing system. *TECS*, 2017.
[10] H. Chiang et al. Tethys: Collecting sensor data without infrastructure or trust. In *IoTDI*, 2018.
[11] A. Colin et al. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *SIGOPS*, 2016.
[12] A. Colin et al. A reconfigurable energy storage architecture for energy-harvesting devices. *SIGPLAN Not.*, 2018.
[13] A. Colin et al. Termination checking and task decomposition for task-based intermittent programs. In *CC*, 2018.
[14] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. *SIGPLAN Not.*, 2016.
[15] B. Denby et al. Kodan: Addressing the Computational Bottleneck in Space. In *ASPLOS*, 2023.
[16] K. Geissdoerfer and M. Zimmerling. Learning to communicate effectively between battery-free devices. In *NSDI 22*, 2022.
[17] C. Gini. *Variabilità e mutabilità: contributo allo studio delle distribuzioni e delle relazioni statistiche.[Fasc. I.].* 1912.
[18] J. Hester et al. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *SenSys*, 2014.
[19] J. Hester et al. Persistent clocks for batteryless sensing devices. *ACM Trans. Embed. Comput. Syst.*, 2016.
[20] J. Hester et al. Timely execution on intermittently powered batteryless sensors. In *SenSys*, 2017.
[21] N. Ikeda et al. Soil-monitoring sensor powered by temperature difference between air and shallow underground soil. *IMWUT*, 2020.
[22] B. Islam and S. Nirjon. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *RTAS*, 2020.
[23] B. J. MacLennan. *Principles of programming languages design, evaluation, and implementation.* 1999.
[24] K. Maeng et al. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 2017.
[25] K. Maeng et al. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *PLDI*, 2020.
[26] A. Maioli. ScEpTIC release. http://sceptic.neslab.it/, 2021.
[27] A. Maioli et al. Discovering the hidden anomalies of intermittent computing. In *EWSN*, 2021.
[28] A. Maioli et al. Dynamic voltage and frequency scaling for intermittent computing. *TOSN*, 2025.
[29] A. Maioli and L. Mottola. Alfred: Virtual memory for intermittent computing. In *SenSys*, 2021.
[30] B. Ransford et al. Mementos: System support for long-running computation on rfid-scale devices.
[31] M. Robert et al. Low-cost open-source recorders and ready-to-use machine learning approaches provide effective monitoring of threatened species. *Ecological Informatics*, 2022.
[32] V. Rovelli et al. Extremis: Static frequency switching for battery-less devices. In *ENSsys*, 2024.
[33] A. Sabovic et al. An energy-aware task scheduler for energy-harvesting battery-less iot devices. *IoT-J*, 2022.
[34] A. M. Van Tilborg and G. M. Koob. *Foundations of real-time computing: Scheduling and resource management.* 2012.
[35] K. S. Yıldırım et al. Ink: Reactive kernel for tiny batteryless sensors. In *SenSys*, 2018.
[36] C. Zhang et al. An efficient time-domain end-to-end single-channel bird sound separation network. *Animals*, 2022.