Intermittent Asynchronous Peripheral Operations

Adriano Branco*, Luca Mottola*†, Muhammad Hamad Alizai+, and Junaid Haroon Siddiqui+
*Politecnico di Milano (Italy), †RI.Se SICS Sweden, +LUMS (Pakistan)

ABSTRACT

Energy harvesting enables battery-less sensing applications, but causes executions to become intermittent as a result of erratic energy provisioning. Intermittent executions pose challenges to *peripheral consistency* that threaten to leave peripheral-bound workloads in failed states or to impede forward progress of programs. Intermittent *synchronous* peripheral operations are supported in existing literature for specific kinds of peripherals. *Asynchronous* peripheral operations enable *reactive concurrency* in application implementations, which increases reactivity and improves energy consumption, but lack dedicated support in intermittent settings. We present Karma, the first general abstraction and system design to support *both* synchronous and asynchronous operations in an intermittent setting. Karma employs a novel combination of peripheral roll-forward and computation roll-back to a rendezvous point guaranteeing consistency. It remains transparent to application programmers and peripheral driver, which favours portability. Our evaluation, based on three applications running on prototype hardware and using diverse energy sources, indicates that intermittent asynchronous peripheral support provided by Karma boosts data throughput by 83% compared to existing literature.

CCS CONCEPTS

• **Computer systems organization** → *Sensor networks*; *Embedded software.*

KEYWORDS

asynchronous operations, peripherals, intermittent computing, energy harvesting

ACM Reference Format:
Adriano Branco*, Luca Mottola*†, Muhammad Hamad Alizai+, and Junaid Haroon Siddiqui+ *Politecnico di Milano (Italy), †RI.Se SICS Sweden, +LUMS (Pakistan). 2019. Intermittent Asynchronous Peripheral Operations. In *The 17th ACM Conference on Embedded Networked Sensor Systems (SenSys '19), November 10–13, 2019, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3356250.3360033

1 INTRODUCTION

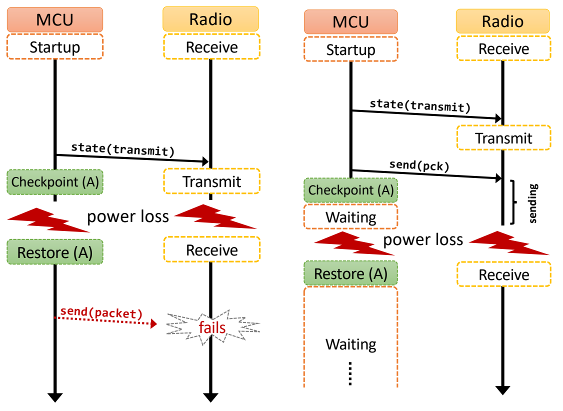
Ambient energy harvesting is enabling battery-less embedded sensing. Devices powered off energy harvesting operate by intermittently executing the software, as energy is available in a small buffer such as a capacitor. System support exists to enable intermittent executions [5, 6, 9, 12, 13, 20, 22, 23, 27, 28, 34, 37], which employs forms of persistence to preserve the state of the computing unit across power failures. Such persistent state is used to resume the execution once energy returns.

Problem. Embedded sensing workloads are most often peripheral-bound. Applications acquire data from the environment through sensors, process the information, and perform actions on the environment or communicate results back to the user, for example, through radio communications. The ability to interact with the external world is at the very essence of embedded sensing [32], and yet necessarily requires the computing unit to interact with peripherals providing the interface with the physical world [21].

Peripherals execute asynchronously with respect to the computing unit. Their functioning is characterized by own states, which are frequently updated due to the execution of I/O instructions or the occurrence of external events, such as the reception of a packet. Information on peripheral states is not automatically reflected in main memory, neither it may be simply queried or restored as it is often the result of non-trivial sequences of commands issued to peripherals and their answers. For example, peripherals operating over I^2C are driven through a specialized protocol where the computing unit acts as a master issuing commands, and peripherals act as slaves returning answers.

The majority of existing solutions for intermittent computing only provide support for the computing unit and expect developers to take care of peripherals [12, 27]. Such a manual one-off effort is anyways necessary because if peripheral states are not restored when resuming executions after a power failure, or this happens without ensuring consistency with respect to the state of the computing unit, applications may fail or forward progress be compromised. These issues represent a threat for *program safety* [1]: executions reach a fail state that is unreachable in a continuous execution, or for *program liveness* [1]: executions fail to reach valid states that would eventually be attained in a continuous execution.

Fig. 1(a) shows an example demonstrating how not preserving peripheral state information leads to a violation of program safety. Say at startup, the radio is automatically initialized in receive state. Later in the execution, the computing unit changes the radio state to transmit, in preparation of a packet send. The system now takes a checkpoint, that is, it dumps the state of the computing unit on non-volatile memory [29, 34]. Following a power failure, the system resumes from checkpoint data when energy is back. The program attempts a send operation, yet the radio is in receive state as no peripheral state information is part of the checkpoint.



(a) Program safety violation. (b) Program liveness violation.

Figure 1: Intermittent peripheral operations cause safety and liveness violations. *In (a), the* **send** *operation is unsafe because the radio reboots in a default receive state. In (b), liveness is compromised as the program forever waits for an asynchronous callback from the radio.*

Violations to program liveness emerge if checkpoints and power failures occur *during* peripheral operations. Such a situation is likely to manifest with *asynchronous* peripheral operations. These are typically structured as separate non-blocking commands and asynchronous callbacks that signal operation completion or relay results [16]. Asynchronous peripheral operations enable *reactive concurrency* [3, 10, 18]: applications are structured as independent code fragments that concurrently execute by sampling sensors, operating actuators, computing, and communicating. Albeit requiring additional development effort, reactive concurrency yields increased reactivity and better energy efficiency [16, 26].

Fig. 1(b) shows an example. The application uses an asynchronous packet send. A callback should be later triggered by the radio to signal the completion of the packet transmission. Say a checkpoint is taken after issuing the packet send and power fails before the completion of the packet transmission. When energy is back and the system resumes from checkpoint data, the execution restarts from a point where the packet send is considered as executed. However, the radio is again re-initialized to the receive state as no peripheral state information is part of the checkpoint. The program is now expecting a callback that signals the completion of an operation the system has no recollection of. Part of the application logic is now stuck awaiting a callback that never arrives.

In both cases, the state of the computing unit and that of peripherals are not *consistent*, that is, the overall system state obtained from combining them does not correspond to any system state reachable in a continuous execution.

Challenges. Designing a general, yet efficient solution to support both (blocking) synchronous[1] and asynchronous intermittent peripheral operations requires to address three challenges:

[1]Although non-blocking synchronous peripheral operations are feasible, for example, using **select/poll** loops as in BSD Unix, these are very rarely supported in low-power embedded operating systems. The discussion that follows only considers a blocking semantics for synchronous peripheral operations.

C1: how to represent the evolution of peripheral states in main memory, so it may be used at the time of resuming computation to form a consistent system state;
C2: how to update this information as the computing unit progresses asynchronously with respect to peripheral operations;
C3: how to schedule peripheral operations whenever they are issued or when resuming after a power failure.

Addressing the three challenges must reconcile generality and efficiency with the resource constraints of target platforms. In intermittent computing, for example, the main computing unit is normally a low-power microcontroller unit (MCU) with little main memory. The energy overhead to support intermittent peripheral operations is ultimately subtracted from the budget for useful computation and communication. Excessive energy overheads may even prevent a program to make eventual progress. On the other hand, peripherals such as sensors or low-power transceivers as used in intermittent computing are generally less sophisticated than peripherals used in mainstream computing.

A few systems enable intermittent executions while preserving peripheral states [4, 7]. As further discussed in Sec. 2, these are often limited in generality with respect to peripheral types, for example, depending on their interface with the computing unit, and most importantly lack support for asynchronous peripheral operations, which enable reactive concurrency in application implementations.

Contribution. To tackle challenge **C1** to **C3**, our contribution spans system design and concrete implementations.

1) We explore the design options available to tackle three challenges above in a way that is both generic and efficient, as discussed in Sec. 3. Such an effort leads us to a reasoned set of coherent design decisions. These include dedicated abstractions to capture the evolution of peripheral states and techniques to roll-forward peripheral states when resuming, while rolling-back the state of the computing unit to a rendezvous point that guarantees consistency.
2) We implement our design in Karma, as illustrated in Sec. 4 Karma is the first peripheral-independent abstraction and system design to cater for both synchronous and asynchronous operations in an intermittent setting. Unlike existing solutions, it resides as an intermediate layer between application and peripheral drivers, and remains transparent to both. This yields increased portability across applications and peripheral types.

We evaluate a working prototype of Karma on real hardware, using diverse applications and energy profiles to show that Karma supports intermittent asynchronous peripheral operations effectively and efficiently. The results we report in Sec. 5 indicate that, for example, Karma increases data throughput by 83% compared with the state of the art. Nonetheless, we provide evidence that such performance gains originate from the specific design decisions at the basis of Karma.

2 BACKGROUND

We survey works related to our efforts first. Next, we offer quantitative evidence that the benefits of reactive concurrency extend to applications and platforms germane to intermittent computing.

2.1 Related Work

We provide background on system support for intermittent computing and survey approaches that preserve peripheral states.

Computing unit: system support. Using a variety of techniques, existing system support creates persistent state in anticipation of power failures. Two flavours exist.

Solutions exist that employ a form checkpointing to let the program cross periods of energy unavailability [5, 9, 29, 34]. This consists in replicating the application state on non-volatile memory, where it is retrieved back once the system resumes with sufficient energy. Systems such as Hibernus [5, 6] operate in a *reactive* manner: an interrupt is fired from a hardware device that prompts the application to take a checkpoint, for example, whenever the energy buffer falls below a threshold. Differently, systems exist that place explicit function calls in application code to *proactively* checkpoint [9, 29, 34, 37]. The specific placement may be a function of program structure and energy provisioning patterns.

Differently, some approaches offer abstractions that programmers use to define and manage persistent state [12, 27, 28] and time profiles [20]. These approaches particularly target mixed-volatile platforms, while taking care of data consistency issues due to repeated executions of non-idempotent code [33]. For example, Alpaca [28] defines tasks as individual execution units that run with transactional semantics against power failures and subsequent reboots, and channels to exchange data across tasks.

Peripherals: recovery approaches. Sytare [7] requires developers to write ad-hoc routines for saving and restoring peripheral states. The complete device state, called "device context", is saved on non-volatile memory before and after every peripheral operation. The state is accrued from device registers at every peripheral request, which potentially hurts performance. Further, Sytare lacks support for peripherals with write-only registers, which is the common case and whose state cannot be simply queried. Asynchronous operations are not supported as Sytare cannot represent the ongoing operation while the computing unit performs other actions.

RESTOP [4] attempts to address these deficiencies by maintaining an explicit operation log, which is updated based on a static labeling of operations to indicate whether they need to be re-played when the device resumes after a power failure. Asynchronous operations are again not supported because of the inability to represent the state of an ongoing peripheral operations. In addition, the operation log may only grow as executions unfold, eventually causing significant memory and energy overhead during executions and when resuming after a power failure.

Samoyed [30] allows programmers to define atomic peripheral functions, wherein the system selectively captures checkpoints and maintains memory consistency. If the work in an atomic function requires more energy than a device can provide, Samoyed iteratively sub-divides the function into multiple smaller function invocations, each containing a subsequence of the original function's work. Unlike Karma, Samoyed also does not support parallel or asynchronous operations.

Our work seeks to overcome these limitations. The design options we explore in Sec. 3 eventually lead us to a specific peripheral state representation that efficiently supports asynchronous operations. This improves energy efficiency and keeps memory occupation under control.

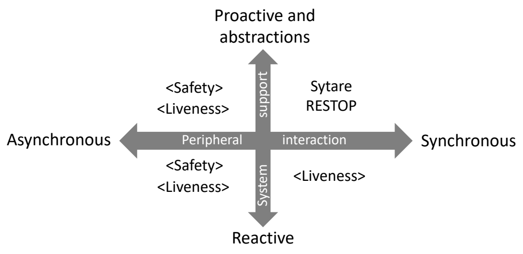


Figure 2: Consolidated view on existing literature, depending on mode of peripheral interaction and system support. *Existing literature only addresses the top-right quadrant. Program liveness is not guaranteed when using synchronous peripheral operations with reactive system support. With asynchronous peripheral operations, neither program safety nor program liveness are ensured, regardless of the system support.*

Peripherals: preventive approaches. Systems exist that prevent intermittent peripheral operations by not issuing peripheral commands until it is guaranteed that the request can run to completion, based on the current energy budget. eM-map [23] and Capybara [13] are examples. The underlying assumption is that the energy requirements of peripheral operations are quantifiable and predictable. In some cases, such as radios, even worst-case assumptions may go totally astray, as the energy cost may vary significantly due to unpredictable factors, for example, random backoffs to handle contention. Platforms that employ separate energy buffers for peripheral operations would suffer from the same problem [19].

Our design is independent of the availability of energy estimates for peripheral operations. If and when available, these information may serve as input for more efficient scheduling of peripheral operations, as we describe in Sec. 3. Differently, Karma still provides efficient support to intermittent peripheral operations by guaranteeing that peripherals and computing unit are synchronized to a consistent state when resuming.

A consolidated literature view. Fig. 2 summarizes our analysis of existing literature. The top-right quadrant maps to the only case that existing literature supports [4, 7]. This holds provided systems calls to checkpoint are placed around peripheral operations or, equivalently, peripheral operations only appear inside computational units with transactional semantics, such as Alpaca tasks.

Differently, synchronous peripheral operations remain a threat to program liveness when using reactive system support, as indicated in the bottom-right quadrant. For example, if a checkpoint is triggered while polling for completion of a synchronous operation, no existing solution may recover the ongoing peripheral operation when resuming. However, the main computing unit re-enters the same polling loop when resuming, this time waiting for a condition to end polling that never occurs.

Asynchronous peripheral operations, on the other hand, are virgin territory. Using available solutions, neither program safety nor program liveness are guaranteed, irrespective of the kind of system support employed. The issue essentially stems from the inability of existing solutions to represent the state of ongoing

Application version	Data throughput	Energy consumption
Busy-wait	11 samples/min	1.08 mJ
Polling	11 samples/min	0.3 mJ
Asynchronous	12 samples/min	0.04 mJ

Figure 3: Performance of an activity recognition application in a one minute execution, depending on peripheral APIs and their implementations. *Using reactive concurrency in application implementations, enabled by asynchronous peripheral operations, allows the system to retain the intended throughput while being energy-efficient.*

peripheral operations while the computing unit executes other actions. In turn, this boils down to the choice of an appropriate abstraction for representing peripheral states, which is at the center of our design discussion in Sec. 3.

2.2 Reactive Concurrency

We argue that the ability to rely on reactive concurrency to write embedded sensing applications using asynchronous peripheral operations is fundamental in an intermittent setting as well. To that end, we provide quantitative evidence here using applications and hardware platforms common in intermittent computing.

Setup. We build a wearable activity recognition application, based on existing detection logic and used as a staple benchmark in intermittent computing [27, 28, 37].

We use an MSP430FR2433 MCU, common in intermittent computing because of the mixed-volatile memory configuration, hooked to a LIS3MDL 3-axis magnetometer via SPI and to a LSM6DSL 3-axis accelerometer/gyroscope via I^2C. Sensors are queried at 100 Hz and their readings combined in batches of 500 samples to infer the current activity of a person among running, walking, sitting, and standing. The result is sent out through a Microchip RN42 BT radio connected via UART; this should happen every 5 sec.

We develop three versions of the application running without dedicated operating system support, that is, on bare hardware, and based on different peripheral interactions: either synchronous—implemented with Busy-wait or Polling—or Asynchronous. These kinds of interactions are generally supported by existing operating systems, including low-power ones. For example, Mbed [31] often employs Busy-wait due to its simplicity. Contiki [14] favors Polling as it blends with protothreads [15]. TinyOS [26] offers a split-phase abstraction for Asynchronous operations. The encoding of the application logic also changes from Busy-wait and Polling to Asynchronous. In the former cases, the data processing is expressed as a single sequential flow. With Asynchronous, data processing is broken down in elementary parts to handle peripheral commands separately from the corresponding answers, which trigger callbacks in the application code.

Results. Fig. 3 shows the measures we gather in a one minute execution. Processing being strictly periodic, the same performance figures remain the same also in the longer term. We draw two fundamental observations.

First, as programmers are unaware of the time taken by peripheral operations, they express application timings regardless of them. This is common practice in embedded programming [24] and bears noticeable effects when the application logic is expressed as a single sequential flow and synchronous calls are used for peripheral operations. In both Busy-wait and Polling, the time spent waiting for synchronous calls to complete progressively shifts the application processing in time. A single minute of execution is sufficient to loose one output sample of the twelve the application should produce. Asynchronous operations allow developers to maintain the expected data throughput despite the application timings being quantified exactly as in Busy-wait and Polling.

Second, despite the energy efficiency of modern MSP430 MCUs, Busy-wait takes a toll on energy consumption. Its energy consumption is more than three times that of Polling, despite the two versions present the same peripheral APIs to programmers. Nonetheless, the additional programming effort due to using asynchronous peripheral operations pays back in terms of reduced energy consumption, which is *one order of magnitude* lower in Asynchronous than in Polling. For a system that runs on harvested energy, such a reduction may prove essential.

The specific application and experimental setting we use are a single instance in a potentially vast landscape, and yet we argue they properly motivate the efforts we describe next.

3 DESIGN

Each of the following sections discusses design options we consider to address the three challenges **C1** to **C3** outlined in the Introduction, along with the rationale for the specific decisions we make.

3.1 C1: State Representation

Existing solutions adopt different approaches, as described in Sec. 2.1. Sytare asks developers to create a peripheral-specific "device context" and to implement specialized routines for saving and restoring. As mentioned, this requires significant developer effort because of the necessary intimate knowledge of, and tight integration with the peripheral driver. RESTOP avoids any explicit state representation; the log implicitly represents the current state of a peripheral as the result of re-applying the entire sequence of commands at boot.

In Karma, we seek to operate at a higher-level of abstraction where the representation of peripheral states and their evolution is easier to define, yet without incurring in performance penalties because of such a design choice.

3.1.1 States. We observe that the behavior of the vast majority of peripherals such as sensors and low-power transceivers is normally described as a *state machine.* State machines appear in sensors and low-power transceivers datasheets. The corresponding drivers often mimic this description, as they are implemented as state machines of sorts. It appears natural to opt for such state representation.

Relying on a state machine to keep track of peripheral states may not be sufficient in general. Depending on what information individual states represent, knowing the state that the peripheral was in at the time of checkpoint may not necessarily allow the system to restore that state after a power failure. For example, this is the case whenever a state is a function of parameters input to peripheral commands, which are not encoded in the state representation. Input parameters may be values taken from arbitrarily large domains, such as a parameter given to an sensor that indicates the warm-up time. It would be overkill to encode input parameters as part of the state representation.

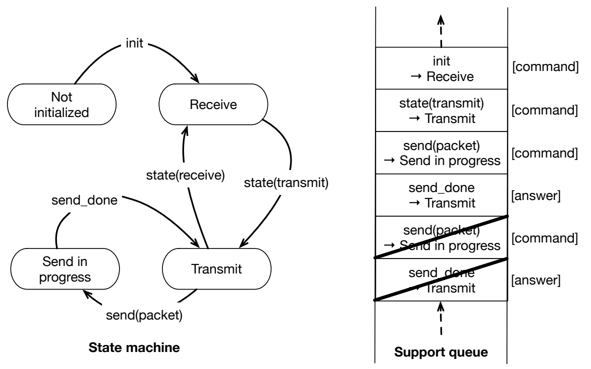

init
Not initialized
Receive
state(receive)
state(transmit)
send_done
Send in progress
Transmit
send(packet)
State machine
init → Receive [command]
state(transmit) → Transmit [command]
send(packet) → Send in progress [command]
send_done → Transmit [answer]
~~send(packet) → Send in progress~~ [command]
~~send_done → Transmit~~ [answer]
Support queue

Figure 4: Abstract example of state machine and support queue. *The state machine represents the current peripheral state and next transitions. The support queue stores input parameters and the sequence of peripheral commands and answers, used when resuming to recover the peripheral state. The combination of the two data structures allows the system to keep track of evolving peripheral states while keeping memory occupation under control.*

We couple the state machine representation with a queue storing the sequence of peripheral commands and corresponding answers. Fig. 4 shows an abstract example. Whenever a command is issued to a peripheral, we update its state machine and push the complete command data to the queue, including parameters and the state the commands leads to. When the peripheral returns an answer, we push the same information to the queue.

The combined use of the two data structures allows the size of peripheral state representation not to grow indefinitely, as it happens in RESTOP. Every time the state machine is updated, we inspect the queue looking for the first entry storing the same target state. If one is found, every entry appearing in the queue after that may be removed. This effectively corresponds to realizing that the execution traversed a loop in the state machine, as in Fig. 4.

Both state machine and support queue are stored in main memory and become part of checkpoint data. The information in these data structures suffices to restore the correct peripheral state after a power failure. Re-issuing the sequence of commands found in the queue, while checking that the peripherals' answers remain the same, takes the peripheral to the same condition it was in after executing the last command before the last checkpoint. Checking answers allows us to identify peripheral failures and signal them to the application; programmers may then handle the situation depending on application requirements.

Adopting this design raises the question of what exactly is a peripheral command or answer, that is, what is the appropriate granularity for state machine transitions.

3.1.2 Transitions. Embedded sensing applications are often logically architected as shown in Fig. 5 [25]. The high-level application logic is split in multiple peripheral-specific application tasks. At this level, an end-to-end functionality describes an application's complete operation with respect to a peripheral, from initialization to tear down. Each peripheral-specific application task interacts with one or more peripheral APIs, which provide an abstraction layer over the peripheral driver. Each operation appearing in the

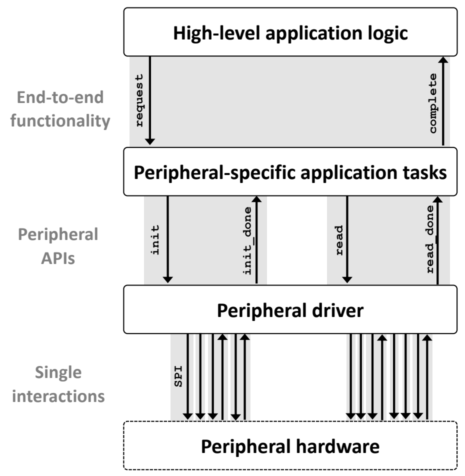


Figure 5: Abstract architecture of embedded sensing applications. *End-to-end functionality describes an application's complete operation with respect to a peripheral, from initialization to tear down. The peripheral APIs expose the functionality of the peripheral driver to the application. Single interactions are single instructions issued to the peripheral, which is how the device driver communicates with the peripheral hardware.*

peripheral API is implemented by multiple individual interactions with the peripheral hardware, which is how the driver implements the low-level operation.

Based on this abstract architecture, there exist three options for considering state transitions, which we discuss next with their pros ($\oplus$) and cons ($\ominus$).

Single interactions. We may consider a peripheral command or answer as the smallest unit of operation of the computing unit with respect to a peripheral, as shown at the bottom of Fig. 5. This is whatever happens as a result of a single instruction issued to a peripheral in case of commands and any single piece of information returned to the computing unit in case of answers. When considering single interactions, the state machine and support queue are updated after every instruction that possibly affects a peripheral and after every interrupt from the peripheral is received.

$\oplus$ Unlike other options discussed next, this option preserves compatibility with asynchronous operations. Commands issued to peripherals and their answers are represented as separate operations. Therefore, a checkpoint occurring after issuing a command, but before executing the callback, yields state information that correctly resume the device.

$\ominus$ As single interactions with peripherals happen inside drivers, the state machine and support queue must be integrated there. This requires intimate knowledge of low-level interfaces and significant development effort. Further, both become larger because of the fine granularity to represent evolving peripheral states, and must accordingly be updated more frequently. Processing and memory overhead increase.

End-to-end functionality. We may consider a complete end-to-end sequence of multiple commands and answers as one state machine transition, as shown at the top of Fig. 5. For example, the sequence of commands required to turn a transceiver in transmit mode, sending a packet, and waiting for a signal that the transmission finished may be considered as a singe "send packet" operation.

⊕ The state machine representing evolving peripheral states is likely to become very compact, while the support queue does not grow excessively. The overhead of state maintenance is therefore reduced compared to other options discussed here. Further, keeping track of the sequence of commands and answers does not require knowledge of the inner operation of peripheral drivers, reducing development efforts.

⊖ Compatibility with asynchronous operations is lost, as asynchronous commands and their answers are merged in single state transitions. Power failures occurring halfway in a sequence of operations require the re-execution of a possibly long succession of commands, increasing energy and time overhead when resuming. Moreover, as we discuss next, ensuring atomic executions of peripheral operations and state updates may require to roll back arbitrary code, which is hard in general. Devices also need to be equipped with a sufficiently large energy buffer to complete the entire end-to-end sequence, or forward progress may be compromised if a device never has enough energy for a complete end-to-end functionality at once.

Peripheral APIs. We may consider the operations appearing in the API of peripheral drivers as the individual state transitions in the corresponding state machine, as shown in the middle of Fig. 5. For example, the operation to trigger a sensor read is considered as a state transition as opposed to the corresponding callback that relays the result.

⊕ This option as well preserves the ability to use asynchronous operations. Commands issued to peripherals and their answers are represented precisely at the granularity that developers rely on when adopting a reactive concurrent program design [3, 10, 18]. The necessary run-time support may be implemented as an intermediate layer between application code and peripheral drivers, by simply wrapping the latter and re-exposing the same API, as show in Fig. 6. An understanding of the driver API is sufficient to this end, with no need to comprehend the low-level intricacies of drivers' internals.

⊖ The overhead of state maintenance is higher than when considering end-to-end functionality as the individual state transitions. Conversely, the energy budget to be available at once for completing individual state transitions is greater than when considering single interactions.

We weigh our options and eventually choose to consider API operations as individual state transitions. Lack of support to asynchronous operations when considering end-to-end functionality defeats the very motivation of this work. The increase in peak energy demands compared to individual interactions proves not to be a hampering factor. We quantitatively assess this aspect in Sec. 5 and demonstrate that considering API operations strikes an efficient trade-off between opposing performance objectives.

3.2 C2: Atomicity

As the computing unit executes asynchronously with respect to peripherals, it may happen that a checkpoint takes place after a

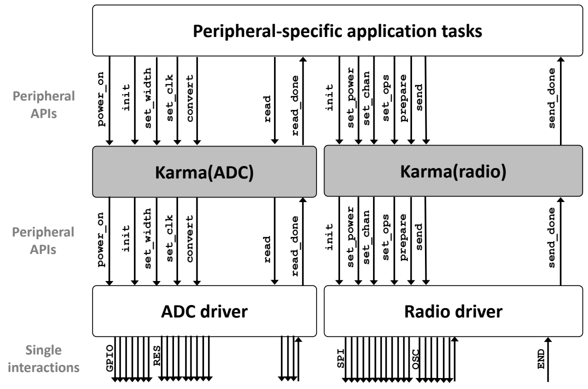


Figure 6: Karma is placed as an intermediate layer between application implementation and peripheral driver. *The picture shows the choice of peripheral APIs for state transitions as opposed to single interactions or end-to-end functionality.*

peripheral operation started but before the corresponding state information are updated. In such a case, we have no way to restore a consistent system state, as we lack information on the peripheral operation being executed at the time of a checkpoint. The opposite, that is, updating state information first and then performing the peripheral operation is not an option, as it may lead to situations where the system thinks a peripheral operation was performed, when it actually did not. We need the change in peripheral state and the update in state information to appear as atomic.

Based on Sec. 3.1, the issue is simply addressed whenever employing system support that inserts explicit system calls that possibly trigger a checkpoint [9, 12, 13, 20, 27, 28, 34, 37]. In such a case, it is sufficient to ensure that such calls are inserted *in application code*, that is, in one of the two topmost layers in Fig. 5, and either before or after peripheral operations. As Karma operates as an intermediate layer between application code and peripheral driver, as shown in Fig. 6, checkpoints would occur before peripheral operations and corresponding state updates are performed, or once calls to peripheral APIs return and state update already occurred.

The case of system support that reactively triggers checkpoints, for example, based on monitoring of a voltage threshold [5, 6, 22, 23], is more complex. In principle, checkpoints may happen at any time, even while executing the low-level peripheral driver code. Two options exist to integrate Karma with such a system support: *i)* changing the conditions that make checkpoints take place in a way to prevent executions lacking the required atomicity, or *ii)* rolling back executions to recover the non-atomic cases.

3.2.1 *Changing conditions for checkpoints.*

Say a checkpoint would normally happen while executing the peripheral driver, but before updating peripheral state information in main memory. In this case, we want the checkpoint to happen before the peripheral operation starts, that is, right before the execution enters the implementation of the peripheral API. Doing so is only possible if the threshold voltage is increased so that the most energy-consuming peripheral operation is not even entered if a checkpoint would normally occur near its very end. This is because we must make sure never to run into cases where a peripheral operation starts without being able to complete and the corresponding state information in main memory are updated. As a further safety measure, we may also opt to disable interrupts all together while executing peripheral drivers.

Such an option, however, is likely detrimental to performance because outside these peculiar cases, checkpoints would occur more frequently than necessary or not occur at all. In the favourable cases, checkpoints take place even though energy is still available to run many more instructions, unnecessarily subtracting resources from useful computation and communication. If the estimates are off, we may enter the implementation of a peripheral API, thereby disabling interrupts, and continue the execution until either the device shuts down without a checkpoint, or interrupts are re-enabled but there is no sufficient energy to perform a checkpoint. For these reasons, we choose to roll back executions.

3.2.2 Rolling back executions. Rolling back arbitrary code is generally a complex problem. The problem we tackle is, however, admittedly simpler that the general case as we only need to roll back specific code fragments, that is, the implementation of peripheral APIs. Moreover, we only need to do so in case a peripheral operation starts but does not reach the point of updating state information.

Our approach is to let the checkpoint happen at an arbitrary point, but to resume only from the start of a peripheral operation in case of a later power failure. This point is the latest time in the execution where a peripheral actual state and the corresponding state information in main memory are consistent. To this end, we need to roll back the execution to the start of the peripheral operation rather to the point where the checkpoint occurs. The instructions between the start of peripheral operation and the checkpoint may, however, alter the overall system state. We describe next how we can undo such changes.

Program counter. Two options exist. In Karma, we adopt a simple solution by storing the value of program counter in a global variable right before accessing the peripheral-specific driver. The value stays there for the duration of the peripheral operation and is invalidated once the latter completes and state information is updated. When resuming, if the global variable is valid, the program counter is restored to its value. This effectively makes the execution resume from right before the start of the peripheral operation if it does not complete before the power failure.

A more complex design may remove the overhead of saving and restoring values to/from the global variable. We may rely on a map of program memory addresses to recognize if we are currently executing a peripheral operation. If so, the address where to resume execution after a power failure may be recovered using the return address in the stack and the `CALL` instruction at the preceding address. In Karma, however, we choose the former design as the overhead of two store operations is small and spares the need of a more complicated resume procedure.

Registers and stack. A benefit of opting to work at the level of peripheral APIs is that peripheral operations have well-defined entry and exit points. The corresponding functions create their own stack frame, later removed when the previous stack pointer is restored. To this end, using a technique similar to the program counter as described above, we also save the current stack pointer in a global variable right before accessing the peripheral APIs. Modifications to subsequent stack frames need not be rolled back if we simply restore the previous stack pointer.

We extend this technique to handle registers. We save their values before entering the implementation of a peripheral API, using a set of global variables. The values stay there for the duration of the peripheral operation and are invalidated once the operation completes. This is performed as part of the Karma intermediate layer that wraps the peripheral APIs, as shown in Fig. 6.

A more efficient, peripheral-specific approach might only save the registers used in a particular call. The corresponding performance gain would come at the cost of increased effort for every peripheral API to be supported, which we mantain is not worth it. Since most functions save registers on the stack, yet another approach may be to invoke Karma inside the peripheral API implementation right after saving registers on the stack. We opt not to adopt this approach, as it would require to change the internals of peripheral API implementations.

Main memory. Peripheral operations may change memory outside the stack either through pointer arguments or by accessing global variables. As these changes would be captured by a checkpoint that occurs during the execution of a peripheral operation, that is, inside the peripheral API implementation, we need to roll back these changes to align the state of main memory with the restored program counter, as per the earlier description.

Regardless of whether memory accesses happen through pointers or by accessing global variables, read operations do not cause an issue as they do not change the state; nothing is to be rolled back in this case. Pointer arguments of peripheral operations are often designated to be read-only or write-only. For example, read-only arguments are provided for configuration parameters, whereas write-only arguments are often encountered to provide memory buffers to store results, for example, a packet just received. Even the latter do not cause an issue because they are written out again when the a peripheral operation (re-)starts. This leaves read-write pointer arguments and writes to global variables as the remaining cases to deal with. Multiple design options exist.

One option is to perform some form of static analysis, which would necessarily include pointer aliasing [39], to understand if and where the checkpoint may include a different memory state compared to the one at the time we save the program counter. For example, a write to a global variable is a problem only if it is possible that the program may read this value after resuming and before over-writing it again. In addition to the complexity of the required analysis, changes to the peripheral API implementation would be needed to avoid such situations.

Another option is to instrument the peripheral API implementation to track all changes in main memory. Incremental checkpoint techniques exist [2, 8, 17] that may be adapted to support the roll-back procedures we require. In essence, we would need to rely on the differential information to keep the previous values of all memory locations changed by the device driver outside its stack frame. When resuming after a power failure, we replace the values in main memory included in the checkpoint with those that we saved before the checkpoint.

A close analysis of the peripheral API implementation of a total of 52 diverse peripherals, as found in the codebases of the TinyOS [26], Contiki [14], and Mbed [31] operating systems, reveals that this is a non-problem. None of these peripheral drivers include any write to global variables. This is not surprising, as relying on global variables would potentially create problems when linking the codebase with applications. Similarly, only one of the 52 peripherals we examine, that is, the ATMG76 CO gas sensor, include read-write pointer parameters in its Contiki API. The same peripheral is supported in TinyOS without that. Based on this, we opt not to include any run-time support to roll-back main memory in Karma, and rather provide a code analysis script that warns programmers of this potential issue whenever a driver is found with either writes to global variables or read-write pointer arguments.

Note that this variety of design choices is possible because atomicity is only required at the level of peripheral API implementation and not in application code. Placing restrictions or performing any form of sophisticated code analysis on the latter would be impractical. However, peripheral drivers are smaller self-standing software components that can be possibly verified, adapted, and certified to work correctly with interruption.

3.3 C3: Scheduling

The problem is two-pronged, as Karma needs to take two scheduling decisions. One is scheduling of a peripheral operations to bring it back to a consistent state compared to the main computing unit when resuming after a power failure. The other is scheduling of asynchronous calls after they are issued from the application code. Karma has control on the latter as well, as it intercepts calls directed to the peripheral APIs.

Unlike the previous discussion, we adopt both design options described next. Developers get to choose what to employ at run-time, based on peripheral features and application requirements.

3.3.1 Eager scheduling. Eager scheduling is the obvious choice. Under eager scheduling, the operations in the support queue are scheduled right after the main computing unit resumes execution. Similarly, asynchronous calls are scheduled as soon as they are issued by application code.

For some peripherals, eager scheduling may not be optimal. If a particular peripheral is costly to initialize, for example, in terms of energy or time, and that peripheral is not used during a particular burst of intermittent execution, the energy spent on its initialization is wasted. Such a situation is not unlikely to happen. A single iteration of simple code may require up to 17 checkpoints [34], each corresponding to a single burst of execution.

Similarly, in a situation of energy scarcity, the application may request to start an asynchronous operation on an energy-hungry peripheral that is unlikely to complete. Karma is going to re-issue the operation when resuming after the imminent power failure; meanwhile, the energy spent on the earlier incomplete operation is wasted. As the application is composed of asynchronous code fragments, chances are high that the wasted energy could be invested somewhere else in useful computation or communication.

3.3.2 Lazy scheduling. Lazy scheduling works in a complementary manner compared to eager scheduling.

Under lazy scheduling, we postpone recovering a peripheral state until the first use of the peripheral. This ensures that the energy and

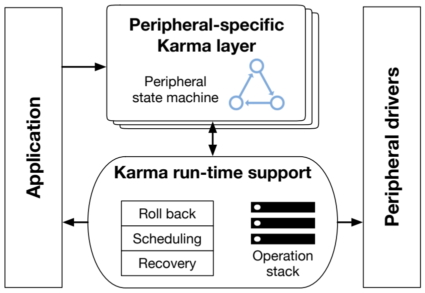


Figure 7: High-level architecture of Karma. *Peripheral-specific Karma layers include the definition of the corresponding state machine. Generic run-time support is in charge of rolling back executions, scheduling peripheral operations, and recovering the peripheral state when the system resumes. The operation stack maintains the list of operations still to be performed.*

time overhead of state recovery for a peripheral are only invested when necessary. The disadvantage, however, is that the execution time for that first peripheral operation is unpredictable from the application point of view. This is because Karma transparently recovers the peripheral state before issuing the actual operation.

The only exception to this processing is when the state information for a certain peripheral indicate the start of an asynchronous operation. In this case, lazy scheduling is not applicable. We must recover the peripheral state immediately as the application is expecting a callback from that operation.

Asynchronous operations may be similarly postponed. This allows the system not to waste energy and time in situations of energy scarcity, as discussed above, and is particularly useful when partially-executed peripheral operations may cause undesirable side effects, for example, on the external environment. Consider for example a radio packet only partially transmitted, or an actuator that only performs a part of the intended action.

Lazy scheduling of asynchronous operations is realized in Karma as a "guard condition" that prevents a state transition until the condition becomes true. For example, a guard condition based on energy can compare the energy estimate for an operation to the remaining energy. When resuming, Karma re-evaluates the guard conditions of all pending peripheral operations. In the presence of multiple such pending operations, we adopt a simple scheduling technique [11] that avoids starvation of the most energy-hungry operations, which may be deferred indefinitely otherwise.

4 IMPLEMENTATION

We provide a few implementation highlights and directions for adopting Karma. Fig. 7 shows the high-level architecture.

As a result of the design choices described in Sec. 3 and our implementation strategy, the application-level consequences of employing Karma are minimal. Programmers keep relying on the same peripheral APIs without being aware of when or how Karma transparently intervenes to ensure that intermittence does not cause incorrect program behaviors.

Peripheral-specific Karma layer. A peripheral-specific Karma intermediate layer includes the definition of the corresponding state machine. Ideally, driver developers should provide this definition, to make the peripheral generally compatible with intermittent executions. We maintain this should not represent a hampering factor, because state machines are often the formalism of choice to describe hardware operations and represent a common programming abstraction of hardware description languages.

When creating a peripheral-specific Karma layer, a few guidelines are worth considering. For example, when defining transitions in a state machine, synchronous calls need not create new states unless they change the peripheral configuration. To resume the peripheral state, in fact, a synchronous call that performs a one-shot operation becomes transparent. For example, a sensor that performs a synchronous read from a "calibrated" state returns to the same state once the operation completes. Resuming the peripheral state is not affected by the possible completion of the former operation.

Asynchronous operations require at least one additional state to represent the situation where the operation is running and the application code is waiting for the callback. Moreover, peripherals that allow multiple concurrent asynchronous operations require explicit states for all combinations of such operations such that, no matter the order that operations are issued, the state machine converges to the same state once all asynchronous operations complete.

Run-time support. We provide generic run-time support that couples with the peripheral-specific Karma layers, including roll-back procedures as per Sec. 3.2 and scheduling policies as per Sec. 3.3.

A dedicated recovery module brings peripherals back in a consistent state when resuming after a power failure. An operation stack maintains a list of calls to peripheral APIs that, because of lazy scheduling, are to be performed at a later stage or whenever their guard conditions become true. In addition to re-evaluating the guard conditions when resuming, we set up a periodic timer to check whether any guard recently changed to true, for example, because of a partial recharge while executing.

The decoupling between the peripheral-specific Karma support and the generic run-time allows the system to preserve complete independence across different peripherals. Parallel peripheral operations are therefore supported by design.

5 EVALUATION

We quantify the performance of Karma using real hardware. In the following, Sec. 5.1 describes the experimental setting, whereas Sec. 5.2 to Sec. 5.4 report the results. We conclude that

1) energy overhead of Karma is limited when checkpointing or resuming as well as when the program executes, and markedly lower than the baselines we consider;
2) energy overhead performance and support to asynchronous operations ultimately yield a 83.3% data throughput improvement compared to baselines;
3) in the benchmarks we consider, memory overhead of Karma in static RAM data and code never reaches anywhere close to the limits of the target platforms.

5.1 Settings

We design our experimental setting to be both realistic and repeatable, as described next.

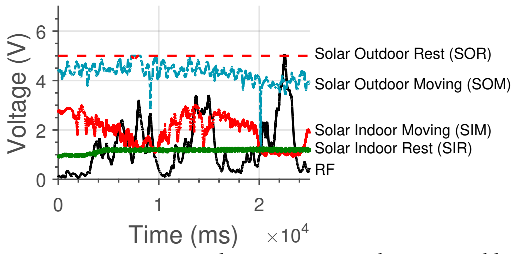


Figure 8: Energy traces. *The SOM trace is the most stable and has highest energy content. The RF trace is the most variable and with least energy content. The other traces provide different intermediate degrees of variability and energy content.*

Applications. We build three real-world applications on top of a TI development board equipped with an MSP430-FR2433 MCU. We develop two versions of each application. One version only uses synchronous peripheral operations, indicated with the -synch suffix. Synchronous peripheral operations are implemented with high-frequency polling whenever possible or with busy-wait otherwise. The other version uses asynchronous peripheral operations, indicated with the -asynch suffix. The implementation of the application logic is refactored from -synch to -asynch to take advantage of reactive concurrency, as discussed in Sec. 2.2.

The first application, called MIC, uses a Silicon SPW2430HR5H analog microphone to implement voice-activated environment monitoring. Whenever the microphone hears a verbal command "temperature" or "humidity", the program queries an attached SHT11 sensor via I^2C to read the corresponding quantity. For energy efficiency, we use a TI TLV voltage comparator that fires an interrupt to the MCU whenever the microphone hears anything above a certain threshold, which we experimentally determine as representative of background noise. The sensor reading is then communicated via a Microchip RN42 Bluetooth radio connected via UART.

The second application is a staple environmental monitoring application, called ENV. Once per minute, the device reads temperature and relative humidity using an SHT11 sensor and ambient light using an array of four ISL29004 sensors facing different directions. Sensors are connected to the MCU via I^2C. Upon acquiring a new sample, a moving window of the last ten readings of each quantity is averaged and the result is transmitted using a TI CC1101 sub-GHz radio connected via SPI.

The third application, termed AR, is the activity recognition application described in Sec. 2.2, running on the hardware we already employed for our motivating example.

Energy. We consider five energy traces, obtained from diverse sources and in different settings. Fig. 8 shows an excerpt, plotting the instantaneous voltage reading over time.

One of the traces is the RF trace from MementOS [34], recorded using the WISP 4.1 [35]. We collect four additional traces using a mono-crystalline solar panel [36] and an Arduino Nano to measure the voltage output. We attach the device to the wrist of a student to simulate a fitness tracker. The student roams in the university campus for outdoor measurements (SOM) and in a research lab for indoor measurements (SIM). Alternatively, we keep the device on the ground right outside the lab for outdoor measurements (SOR) and at desk level in our research lab for the indoor measurements

Application	Checkpoint/resume energy (no peripheral support)	Karma	KarmaLow	Sytare	RESTOP
MIC-ssynch	314μJ/135μJ	+3.1%/+9.3%	+6.2%/+9.2%	+5.1%/+9.3%	+15.1%/+29.1%
MIC-asynch	321μJ/132μJ	+3.3%/+9.2%	+5.9%/+9.4%	N/A	N/A
ENV-ssynch	409μJ/121μJ	+2.8%/+10.3%	+5.8%/+10.4%	+5.7%/+9.8%	+17.3%/+27.3%
ENV-asynch	421μJ/119μJ	+3.2%/+11.1%	+6.1%/+11.0%	N/A	N/A
AR-ssynch	327μJ/122μJ	+2.9%/+11.3%	+5.0%/+11.4%	+4.9%/+10.1%	+19.3%/+28.3%
AR-asynch	312μJ/102μJ	+3.8%/+10.9%	+5.9%/+11.1%	N/A	N/A

Figure 9: Energy overhead when checkpointing and resuming. *Karma has the lowest overhead overall, due to the abstractions used for state representation and the roll-back techniques for the MCU. Sytare and KarmaLow perform worse when checkpointing, whereas RESTOP shows worst performance because of the ever-growing operation log.*

Application	Base cost	Karma	KarmaLow	Sytare	RESTOP
MIC-ssynch	52μJ	+2.9%	+5.9%	+4.3%	+22.7%
MIC-asynch	9μJ	+7.3%	+11.2%	N/A	N/A
ENV-ssynch	43μJ	+2.3%	+5.3%	+5.1%	+38.6%
ENV-asynch	5μJ	+8.3%	+13.2%	N/A	N/A
AR-ssynch	27μJ	+1.8%	+5.1%	+4.9%	+22.7%
AR-asynch	3μJ	+7.9%	+12.1%	N/A	N/A

Figure 10: Energy overhead compared to continuous executions. *The energy overhead of Karma is smallest across the board. Sytare's performance comes at the cost of significant developer effort to integrate with peripheral drivers. KarmaLow and RESTOP suffer from an inappropriate choice of state representation.*

(SIR). Fig. 8 visually demonstrates the extreme variability and considerable differences among the traces we consider.

The energy traces are fed to the device using a Renesas digital power supply driven by an RL78/I1A controller. Similar to Ekho [38], such a setup allows us to replicate the exact V-I curve the device would experience if attached to the actual energy harvester while considering the equivalent resistance offered by the device, and yet retain repeatability across applications and experimental settings. By virtue of this setup, we also quantitatively assess the amount of peripheral-related intermittence that our device experiences during the experiments. Depending on the energy trace, this ranges from about one out of three (ten) peripheral operation that are interrupted because of a power failure in the RF (SOR) trace.

Checkpoints occur by dumping the entire device state, including registers and program counter, on the on-board FRAM, similar to Hibernus [5, 6]. We experimentally determine the voltage threshold to trigger the checkpoint and use a secondary TI TLV voltage comparator to fire a corresponding interrupt to the MCU. The device is attached to a 802μF capacitor, which we find to be sufficient for all applications we consider to make eventual progress.

Metrics and baselines. We compare the performance of Karma against Sytare [7] and RESTOP [4], which we apply to all peripheral drivers we use in the applications above. Neither of these supports asynchronous peripheral operations, so -asynch versions of the three applications are only measured with Karma.

When using Karma, the implementation of the peripheral-specific Karma layers, including the definition of the state machines, required from half a day to an entire day of work by a M.Sc. student; the development effort is thus reasonable. In addition, we consider a different version of Karma that works at the level of single interactions with peripherals, as described in Sec. 3.1, and call it KarmaLow. This is instrumental to obtain quantitative evidence that the trade-offs discussed in Sec. 3.1 play in favour of our design decisions. We do not carry out the same exercise by considering end-to-end functionality, as they loose compatibility with asynchronous operations and thus defeat the motivation for this work.

Based on real executions lasting for at least 10 hours[2] in every configuration and the information obtained from an attached Saleae FTG456 logic analyzer, we compute staple performance metrics in intermittent systems [5, 9, 29, 34, 37]

Energy overhead, that is, the additional energy cost due to supporting intermittent peripheral operations, either when checkpointing and resuming or during program execution;

Data throughput, that is, the net amount of application data produced by the considered application over time, as transmitted by the on-board radio and including charging times;

Memory overhead, that is, the amount of additional static RAM data and program memory required when using a specific support for intermittent peripheral operations.

Energy overhead represents the actual cost for employing a given peripheral support for intermittent computation. Such an energy cost is subtracted from the budget to perform useful computations or to operate peripherals for application-level tasks, and must therefore be minimized. The impact of energy overhead on end-user performance shows as a reduction in data throughput, which is the performance figure that ultimately represents a system's overall efficiency. Memory overhead, on the other hand, provides an indication of the feasibility of a given peripheral support against the constraints of target platforms.

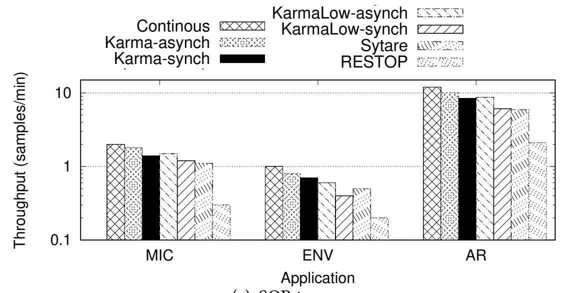
We run application MIC by providing the exact same voice input every 30s, so that executions are comparable when using different configurations. Processing in the other two applications is the same regardless of the sensor inputs.

5.2 Results → Energy Overhead

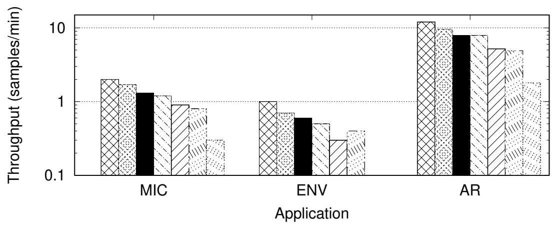
We perform two kinds of measurements. First, we assess the additional energy cost of including peripheral state information in a checkpoint, either at the time of taking the checkpoint or when resuming. Next, we investigate the additional energy consumption during program execution, due to the different bookkeeping that either peripheral support system requires.

Checkpointing and resuming. The energy overhead when checkpointing and resuming stems from *i)* additional processing depending on the chosen abstraction to capture the evolving peripheral states, and *ii)* additional data to write on non-volatile memory together with checkpoint data.

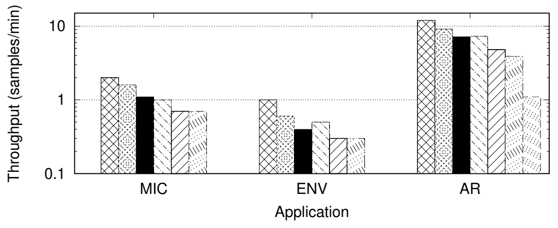
[2]Whenever the available energy traces do not span such a time, we simply replicate them as needed.



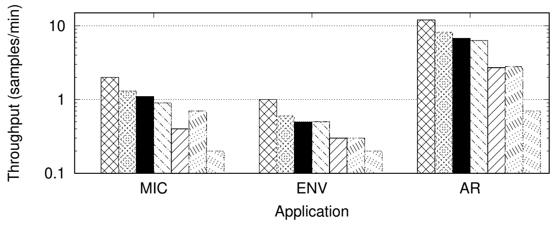
(a) SOR trace.



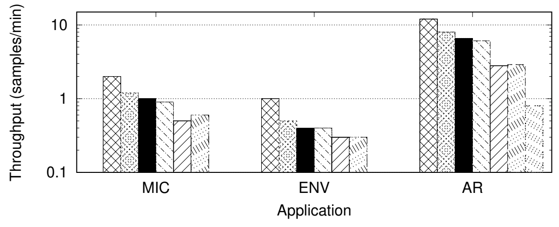
(b) SOM trace.



(c) SIM trace.



(d) SIR trace.



(e) RF trace.

Figure 11: Data throughput. *Karma provides best performance because of reduced overhead when checkpointing and resuming, plus limited energy overhead while executing. Sytare and RESTOP offer lower throughput because of higher energy overhead. The major gain comes from asynchronous operations, which enable reactive concurrency. This improves data throughput up to 83.3% compared with the baselines that only run -synch versions.*

Fig. 9 shows average results over the 10-hour executions. The energy overhead figures are remarkably consistent for the same peripheral support system. Karma has the lowest overhead both when checkpointing and when resuming, due to the abstraction we choose to capture the evolution of peripheral states as well as the roll-back technique we employ for the MCU.

KarmaLow and Sytare have comparable performance than Karma when resuming, as the two systems essentially perform the same sequence of operations to bring peripherals back to the required state. Both operate at a lower level of abstraction when checkpointing, which is detrimental to their energy performance at that stage. RESTOP shows worst performance as the operation log grows as the system continues to run, so the energy required to checkpoint the log increases as a result.

Among applications, ENV has most sensors to deal with, thus the energy cost of checkpointing and resuming are higher. The energy cost of checkpointing is generally higher than resuming due to the need to write on non-volatile memory, whereas only reading is needed when resuming.

Program execution. Exclusively for understanding the net additional overhead during program executions, we run the applications over a continuous stable power supply: they are not intermittent. Such an overhead is due to keeping track of the evolution of peripheral states so that they can be brought back to the correct state.

Fig. 10 shows our energy measurements in a single application run. Karma's overhead is remarkably limited, as it reaches a 2.9% (8.3%) worst case for -synch (-asynch) versions. We maintain that this performance is enabled by the design choices we make, described in Sec. 3, including the abstraction we choose and the operation granularity we opt to consider. The resulting design allows us to place Karma as an intermediate layer between application and peripheral drivers. This induces reduced energy overhead even when supporting -asynch versions, which makes it a viable option to develop programs that take advantage of reactive concurrency.

The results of KarmaLow provide quantitative evidence for this observation. The energy overhead is markedly higher for both -asynch and -synch versions. As discussed in Sec. 3.1, a finer granularity in capturing evolving peripheral states causes higher overhead as state information are to be updated more frequently. This corresponds to no other benefit compared to Karma, as both equally support asynchronous operations.

When considering -synch versions, the energy overhead of Sytare is generally twice that of Karma, whereas RESTOP shows one order of magnitude higher energy overhead. Sytare requires significant developer effort because of the tight integration with peripheral drivers. We argue that the performance of RESTOP stems from the the abstraction chosen to keep track of evolving peripheral states, which imposes significant processing for every interaction with the peripheral. This is particularly evident for ENV, because of the non-trivial control logic necessary to drive the CC1101 radio.

5.3 Results → Data Throughput

Fig. 11 shows the data throughput in intermittent executions depending on application and energy trace. The plot is in log scale.

Whenever programmers are limited to synchronous operations, Karma improves data throughput by a minimum 32% compared to the baselines, while remaining within a 36% bound from the performance of a continuous execution. This is the combined effect of reduced information to be included in the checkpoint data,

Application	Static RAM data/code (no peripheral support)	Karma	KarmaLow	Sytare	RESTOP
MIC-ssynch	7.67KB/42.6KB	+10.3%/+3.2%	+10.4%/+3.2%	+9.3%/+2.9%	+4.2%/+2.7%
MIC-asynch	9.12KB/43.2KB	+9.9%/+3.1%	+9.8%/+3.2%	N/A	N/A
ENV-ssynch	5.89KB/31.53KB	+12.3%/+3.8%	+12.4%/+3.9%	+10.1%/+2.8%	+4.4%/+2.2%
ENV-asynch	8.19KB/32.87KB	+12.1%/+3.6%	+12.2%/+3.7%	N/A	N/A
AR-ssynch	6.23KB/33.63KB	+11.1%/+3.3%	+11.2%/+3.4%	+10.8%/+3.5%	+3.8%/+2.8%
AR-asynch	8.31KB/35.92KB	+11.8%/+3.2%	+11.9%/+3.1%	N/A	N/A

Figure 12: Memory overhead for static RAM data and code. *Karma has slightly higher memory overhead than the baselines, which comes as the cost not to integrate with the peripheral driver. The overall memory consumption remains within the limits of target platforms, with plenty of room still available for additional functionality.*

more efficient recovery after a power failure, and a limited energy overhead when executing, as discussed in Sec. 5.2.

These observations are confirmed by the performance of Sytare and RESTOP. Sytare is the second-best performing solution, and yet it often only delivers about 50% of the throughput attainable in a continuous execution and requires a tight integration with peripheral drivers. RESTOP, on the other hand, greatly suffers as the log to be included in checkpoint data grows with time and thus energy performance progressively deteriorates. RESTOP often only delivers a fraction of the throughput of a continuous execution.

The major performance gain shows when using asynchronous operations, which neither Sytare nor RESTOP support. In the applications we consider, data throughput of the -asynch versions improves by 83.3% compared with the baselines that can only run -synch versions. In addition to the beneficial effects of reduced energy overhead, the ability to leverage reactive concurrency unlocks great performance gains. This requires accordingly restructuring implementations, which we demonstrate to be worth doing.

Interestingly, the same ratio of performance improvement when using asynchronous operations applies to KarmaLow as well, but the absolute values are lower. Opting for a finer granularity in capturing evolving peripheral states thwarts the advantages of the specific state representation we employ. The performance of KarmaLow ends up being comparable to Sytare, thus showing no advantages over the existing state of the art. What we learn from this is that performance improvements are unlocked only by a reasoned set of coherent design decisions, as we discuss in Sec. 3.

Comparing the plots in Fig. 11 with each other, we also note how Karma's higher energy efficiency yields better resilience against variable energy provisioning patterns. The performance with the SOR trace, shown in Fig. 11(a), is the best in absolute terms, as the trace is the least variable and supplies the highest energy content. Karma is the least affect by the increased variability and reduced energy content of the SOM, SIM, and RF traces, shown in Fig. 11(b), Fig. 11(c), and Fig. 11(e) respectively. Conversely, RESTOP is down to less than one sample per minute in all applications using the RF trace, because of the significant energy overhead of storing the entire log as part of the checkpoint data. This subtracts significant energy from an already thin budget.

5.4 Results → Memory Overhead

Fig. 12 reports our measurements. Adding peripheral support for intermittent computing, regardless of what solution is adopted, never exceeds the memory budget of the target platform: the additional memory consumption in either static RAM data or code is generally limited. Memory overhead for intermittent peripheral support generally stems from a fixed cost due to the necessary run-time support, plus a variable cost that is a function of the number of peripherals and the complexity of interacting with them.

RESTOP has the least memory overhead of all considered systems. The log data structure requires little run-time support and a small amount of static RAM data. Sytare, Karma, and KarmaLow trade more general peripheral support for an increase in memory overhead for both figures. Karma's and KarmaLow's memory overhead are generally comparable as the occupation of static RAM and code only partly depends on the granularity to capture evolving peripheral states. Karma's figures are slightly greater than Sytare's mainly because of the decision not to integrate with the peripheral driver as Sytare, which requires additional bookkeeping.

These observations apply particularly to the worst-case setting of ENV, which employs a total of six peripherals, demonstrating overall scalability against the number of peripherals. Note how the static RAM data for the -asynch versions is generally higher than the -synch versions. This is due to the increased use of global data to make it accessible across asynchronously-executed segments of code. Despite the increase in static RAM data, the percentage overhead of Karma remains comparable to the -synch versions. The overall memory consumption never reaches anywhere close to the limits of the target platform and plenty of room remains available for additional functionality.

6 CONCLUSION

Peripheral operations present fundamental unsolved challenges to developers of intermittent computing systems. Support to asynchronous peripheral operation is, in particular, lacking in the state of the art. This prevents developers from employing reactive concurrency in application implementations, which potentially leads to higher reactivity and better energy efficiency.

We addressed these challenges through a reasoned set of coherent design decisions and concrete implementations, leading to efficient system support for intermittent peripheral operations. Our system Karma is based on dedicated abstractions to capture the evolution of peripheral states and techniques to roll-forward peripheral states when resuming, while rolling-back the state of the computing unit to a rendezvous point that guarantees consistency. Our evaluation using prototype hardware and diverse real energy traces indicates that the energy overhead of Karma is significantly lower than the considered baselines, data throughput improves by 83.3%, and memory overhead leaves plenty of room available on the target platforms for additional functionality.

REFERENCES

[1] Bowen Alpern and Fred B Schneider. 1987. Recognizing safety and liveness. *Distributed computing* 2, 3 (1987), 117–126.

[2] Faycal Ait Aouda, Kevin Marquet, and Guillaume Salagnac. 2014. Incremental checkpointing of program state to NVRAM for transiently-powered systems. In *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip*.

[3] Joe Armstrong. 2013. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf.

[4] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18, 1 (2018), 172. https://doi.org/10.3390/s18010172

[5] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. 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 (2016), 1968–1980. https://doi.org/10.1109/TCAD.2016.2547919

[6] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* 7, 1 (March 2015), 15–18. https://doi.org/10.1109/LES.2014.2371494

[7] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (12 2018).

[8] Naveed Bhatti and Luca Mottola. 2016. Efficient state retention for transiently-powered embedded sensing. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks*.

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

[10] Frédéric Boussinot and Robert De Simone. 1991. The ESTEREL language. *Proc. IEEE* 79, 9 (1991), 1293–1304.

[11] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 443–454.

[12] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. *SIGPLAN Not.* 51, 10 (Oct. 2016), 514–530. https://doi.org/10.1145/3022671.2983995

[13] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. 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 (ASPLOS '18)*. ACM, New York, NY, USA, 767–781. https://doi.org/10.1145/3173162.3173210

[14] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*. IEEE Computer Society, Washington, DC, USA, 455–462. https://doi.org/10.1109/LCN.2004.38

[15] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/1182807.1182811

[16] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2014. The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not.* 49, 4 (July 2014), 41–51. https://doi.org/10.1145/2641638.2641652

[17] Saad Hamed, Naveed Bhatti, Junaid Siddiqui, Hamad Alizai, and Luca Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[18] Maurice Herlihy. 1993. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 5 (1993), 745–770.

[19] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/2809695.2809707

[20] Josiah Hester, Kevin Storer, , and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. *Sensys'17* (2017), 13. https://doi.org/10.1145/3131672.3131673

[21] Michael Jackson. 1995. The world and the machine. In *International Conference on Software Engineering*. IEEE, 283–283.

[22] H. Jayakumar, A. Raha, and V. Raghunathan. 2014. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. 330–335. https://doi.org/10.1109/VLSID.2014.63

[23] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Trans. Embed. Comput. Syst.* 16, 3, Article 65 (April 2017), 23 pages. https://doi.org/10.1145/2983628

[24] P. Koopman. 2010. *Better Embedded System Software*. Carnagie Mellon Press.

[25] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A Brewer, and David E Culler. 2004. The Emergence of Networking Abstractions and Techniques in TinyOS.. In *NSDI*.

[26] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2004. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*. Springer Verlag.

[27] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. *SIGPLAN Not.* 50, 6 (June 2015), 575–585. https://doi.org/10.1145/2813885.2737978

[28] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920

[29] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 129–144.

[30] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-time Checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1101–1116. https://doi.org/10.1145/3314221.3314613

[31] mbed. 2017. *IoT OS*. goo.gl/u918jX.

[32] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Design Automation Conference*. IEEE, 731–736.

[33] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM, 5.

[34] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 159–170. https://doi.org/10.1145/1961295.1950386

[35] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (Nov 2008), 2608–2615. https://doi.org/10.1109/TIM.2008.925019

[36] IXYS SolarMD. 2018. *SLMD481H08L*. http://ixapps.ixys.com/ (accessed 2018-02-28).

[37] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 17–32. http://dl.acm.org/citation.cfm?id=3026877.3026880

[38] Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob Sorber. 2011. Ekho: Bridging the Gap Between Simulation and Reality in Tiny Energy-harvesting Sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower '11)*. ACM, New York, NY, USA, Article 9, 5 pages. https://doi.org/10.1145/2039252.2039261

[39] Sean Zhang, Barbara G Ryder, and William Landi. 1996. Program decomposition for pointer aliasing: A step toward practical analyses. *ACM SIGSOFT Software Engineering Notes* 21, 6 (1996), 81–92.