Dynamic task-based intermittent execution for energy-harvesting devices

Majid, Amjad Yousef; Delle Donne, Carlo; Maeng, Kiwan; Colin, Alexei; Yildirim, Kasim Sinan; Lucia, Brandon; Pawełczak, Przemysław

DOI
10.1145/3360285

Publication date
2020

Document Version
Final published version

Published in
ACM Transactions on Sensor Networks

Dynamic Task-based Intermittent Execution for Energy-harvesting Devices

AMJAD YOUSEF MAJID, Delft University of Technology
CARLO DELLE DONNE, QuTech, Delft University of Technology
KIWAN MAENG and ALEXEI COLIN, Carnegie Mellon University
KASIM SINAN YILDIRIM, University of Trento and Ege University
BRANDON LUCIA, Carnegie Mellon University
PRZEMYSŁAW PAWEŁCZAK, Delft University of Technology

Energy-neutral Internet of Things requires freeing embedded devices from batteries and powering them from ambient energy. Ambient energy is, however, unpredictable and can only power a device intermittently. Therefore, the paradigm of intermittent execution is to save the program state into non-volatile memory frequently to preserve the execution progress. In task-based intermittent programming, the state is saved at task transition. Tasks are fixed at compile time and agnostic to energy conditions. Thus, the state may be saved either more often than necessary or not often enough for the program to progress and terminate. To address these challenges, we propose Coala, an adaptive and efficient task-based execution model. Coala progresses on a multi-task scale when energy permits and preserves the computation progress on a sub-task scale if necessary. Coala's specialized memory virtualization mechanism ensures that power failures do not leave the program state in non-volatile memory inconsistent. Our evaluation on a real energy-harvesting platform not only shows that Coala reduces runtime by up to 54% as compared to a state-of-the-art system, but also it is able to progress where static systems fail.

CCS Concepts: • **Hardware → Power and energy**; • **Software and its engineering → Embedded software**; **Virtual memory**;

Additional Key Words and Phrases: Energy-harvesting devices, intermittent computing, adaptive software

ACM Reference format:
Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Trans. Sen. Netw.* 16, 1, Article 5 (February 2020), 24 pages.
https://doi.org/10.1145/3360285

5

Authors' addresses: A. Y. Majid and P. Pawełczak, Delft University of Technology, Mekelweg 4, Delft, The Netherlands, Zuid Holland, 2628 CD; emails: {a.y.majid, p.pawelczak}@tudelft.nl; C. D. Donne, QuTech, Delft University of Technology, Lorentzweg 1, Delft, The Netherlands, Zuid Holland, 2628 CJ; email: c.delledonne@tudelft.nl; K. Maeng, A. Colin, B. Lucia, Carnegie Mellon University, 4720 Forbes Avenue, Pittsburgh, PA, USA, 15213; emails: {kmaeng, acolin, blucia}@andrew.cmu.edu; K. S. Yıldırım, Ege University, Department of Computer Engineering, Izmir, Turkey 35100, and University of Trento, Department of Information Engineering and Computer Science, Via Sommarive, 9 I-38123, Povo (TN), Italy; email: kasimsinan.yildirim@unitn.it.

1550-4859/2020/02-ART5
https://doi.org/10.1145/3360285

1 INTRODUCTION

Advances in processor efficiency along with the development of energy-harvesting systems have created a new category of devices that require neither a battery nor a tethered power supply [40, 55, 63]. These devices operate using ambient energy, such as radio-frequency transmissions [19], light [46, 47], and vibration [20]. Incorporating compute, storage, sensing, and communication hardware [14, 51, 76], such devices are a promising technology for use in the Internet of Things [35], in-body [49] and on-body [5] medical systems, and energy-harvesting nano-satellites [14, 45]. Energy-harvesting devices create unique challenges, because they operate *intermittently* when energy is available [29, 40]. An energy-harvesting device buffers energy in a small storage capacitor [21, 23] and operates when a threshold amount of energy has accumulated. Harvestable energy sources are low-power (e.g., nW to μW) compared to a platform's operating power level (hundreds of μW to mW). A device operates briefly until it depletes its buffered energy, after which, it shuts down and recharges to operate again later—corresponding to the *intermittent execution model* [40, 41] composed of operation-power failure-restart cycles. The recharge and discharge intervals—which correspond to the device's inactive and active time—vary depending on the underlying hardware, such as the size of the energy buffer [14], and energy conditions. For example, some devices discharge and restart ≈10 to ≈100 times per second [42, 57, 66].

Upon power failures, a device loses the volatile state in its registers, stack, and SRAM and retains the state of any non-volatile memory, such as FRAM. While capturing periodic checkpoints [33, 57] and sleep scheduling [3, 4, 8] help preserve execution progress, failures can leave non-volatile state incorrect, partially updated. These inconsistencies cause intermittent execution to deviate from continuously-powered behavior, leading to an unrecoverable application failure [11, 41]. Prior work developed two main approaches to deal with data inconsistency for intermittently-powered devices: (i) *software-based programming and execution models* [12, 41, 43, 73] and (ii) *hardware-based architectural support* [29, 42, 48]. Complex hardware architectural changes are expensive to design, verify, and manufacture. New hardware architectures are also inapplicable to existing systems [29, 42]. Software approaches are simpler and applicable to existing devices today. Therefore, this work focuses on software approaches. In particular, we address the key limitation of task-based programming and execution model, that is the *inflexibility* and *energy-unawareness* of statically decomposing a program into tasks.

Drawbacks of Static Task Decomposition. Task-based programming and execution models require a programmer [12, 43, 54] or a compiler [2] to statically decompose a program into a collection of tasks. *Tasks*, top-level functions, can include arbitrary computation that should be executed despite arbitrarily timed power failures. The programmer (or a compiler) explicitly expresses task-to-task control flow. Figure 1 (left) illustrates how a program's tasks execute and shows how task transitions can affect runtime. At each transition, the system incurs an overhead to track and atomically commit modifications to the non-volatile memory to maintain consistency of program state [12, 43]. The more task transitions a program requires, the more commit overhead is incurred by the system at runtime. A programmer may thus create very large tasks in an attempt to reduce task transitions overhead. However, a large task may require more energy to complete than a device's fixed hardware energy buffer can hold, which may lead to a task *non-termination* problem (Figure 1, right). To eliminate this risk, existing systems require the programmer to decompose a program into small tasks to preserve execution progress. These constraints on task sizing lead to the following *dilemma*: Should large tasks be used that are efficient but risk non-termination or small tasks that are guaranteed to complete but incur a high task transition and commit overhead?

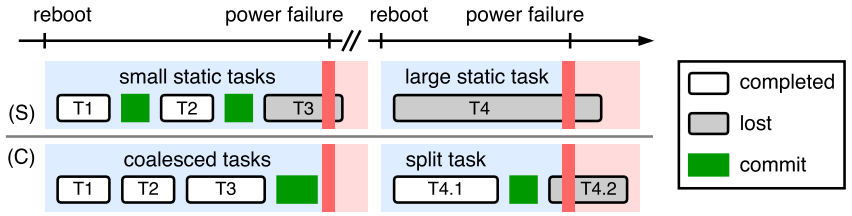


Fig. 1. Task coalescing reduces commit overhead (left), while task splitting enables termination of large tasks (right). S: static legacy task-based runtime, C: Coala (this work).

Challenges and Contributions. We introduce Coala[1]: a new task-based system that employs *adaptive task size execution by task coalescing and splitting*. By means of this novel technique, small tasks can be executed efficiently by trimming unnecessary overhead *dynamically*, meanwhile avoiding the risk of non-termination. Coala accepts any static decomposition and it coalesces (groups) tasks or splits them (see again Figure 1) based on the estimated energy availability without demanding any hardware support. To the best of our knowledge, Coala is the only system that eliminates restructuring and re-compilation of applications considering a device's energy buffer—enabling energy-storage independency for the applications, while keeping execution efficient.

The unique contributions of Coala in relation to challenges of task-based systems revealed by this work are listed below.

- C1 *Overcoming Task Transition Overhead:* Given unpredictable incoming energy, how do we save computation state at task transitions as rarely as possible? Coala tries to minimize task transition overhead by estimating energy conditions at runtime using *recent execution history* as a metric.
- C2 *Dynamic Memory Consistency Handling:* Merging static tasks on the fly creates the need for dynamic memory consistency handling. This leads to the second challenge: How should we dynamically detect data dependencies across coalesced tasks and ensure efficient protection against power interrupts? Coala addresses this challenge by relying on its novel *Virtual Memory Manager* (VMM). The VMM performs real-time dependency tracking to enable protection on a task transition. Individual variables tracking, however, slows down a system dramatically. Therefore, the VMM keeps memory consistent through *privatizing pages* and optimizes bulk shared-data transfer through Direct Memory Access (DMA).
- C3 *Ensuring Task Termination:* A static task decomposition model assumes that each task can execute to completion. If the hardware energy buffer provides inadequate energy to execute each task to completion, then a program will not terminate [13]. This leads to a third challenge: How do we enable the dynamic execution model to progress on a sub-task level? To avoid non-termination under adverse energy conditions, Coala uses a timer-based *partial task commit* mechanism. Partial execution avoids non-termination by committing the intermediate state of a long-running task that has repeatedly failed and restarted.

To asses the benefits of Coala over existing task-based systems, we implemented and tested six benchmarks on a real energy-harvesting platform. Our evaluation shows that Coala reduces runtime overhead by up to 54% and solves task non-termination problem where existing static task-based systems fail.

The rest of this article is organized as follows. Section 2 provides background on intermittent computing. Section 3 illustrates an overview of Coala. Section 4 describes Coala's task adaptation

[1]Coala's source code is accessible via https://github.com/TUDSSL/Coala.

mechanism. The virtual memory manager immune to power interruptions is explained in Section 5. Implementation details of Coala are given in Section 6. Sections 7 and 8 describe Coala's evaluation methodology and results, respectively. Section 9 positions Coala in the context of related work and Section 10 concludes our work.

2 INTERMITTENT COMPUTING: BACKGROUND

2.1 Energy Harvesting Systems

Energy harvesting devices operate using energy extracted from ambient sources, e.g., radio-frequency transmissions, light. These devices elide tethered power and batteries, instead collecting energy into a capacitor, operating when sufficient charge accumulates, and turning off for recharge upon depletion of the buffer. There are several energy harvesting battery-less platforms. For instance, computational RFIDs include open-source TI MSP430-based [69] WISP [51] (with its variants such as WISPCam [50], NFC-WISP [77] or NeuralWISP [30]), Moo [76], and commercial ones such as Medusa [17]. Other platforms include ambient backscatter tag [39, 44, 52] or battery-less phone [65]. Coala is designed for the demands of existing and future *tiny embedded energy-harvesting platforms* based around general purpose, commodity computing components [62, 71]. Coala targets a device with a memory system that has fast, byte-addressable volatile and non-volatile memory; in particular, our target platform, WISP [62], is equipped with a mixture of SRAM and FRAM. Coala leverages hardware support for fast bulk-copying between memories via DMA [71]. Coala does not require architectural additions to commodity processors as in [29, 33, 42, 64].

2.2 Intermittent Execution

Software running on an energy-harvesting device executes *intermittently*, because power sources are not always available to harvest and buffer sufficient operational energy. An intermittent execution is composed of operating periods interspersed with power failures [12, 41, 43, 73]. The charge-discharge cycle of a device depends on the size of the device's energy storage buffer (a larger buffer allows longer operating periods), current consumption, and incoming power.

A power failure clears volatile state (e.g., registers and SRAM) while non-volatile memory (e.g., FRAM) persists. Upon power failure, control flows to a prior point in the execution: By default, to the beginning of `main()`. Early intermittent systems preserved progress by periodically checkpointing volatile execution context to non-volatile memory [57]. Some of the more recent systems requiring hardware support to checkpoint include [3, 4, 7, 33, 48], Table 1 summarizes the data that get copied to and from the non-volatile memory in service of memory consistency and progress preservation. Checkpointing volatile state alone does not ensure data consistency when the system can directly manipulate non-volatile memory [56]. Precisely, data can get inconsistent when code includes a *write-after-read* (WAR) dependency between operations that manipulate non-volatile memory. Figure 2 illustrates how a program state can become inconsistent in an intermittent execution using a simple example of an average operation over an array of integers. The non-volatile variable `sum` introduces the WAR, being read and written sequentially by the increment operation. If a power failure occurs right before updating the non-volatile index `i` (Figure 2(b), Line 6), then `sum` gets erroneously incremented twice consecutively by the same array element (Figure 2(b), Lines 5 and 8).

2.3 Task-based Intermittent Programming

Task-based execution models [12, 41, 43] ask the programmer to decompose their program into **tasks**, which are regions of code that can contain arbitrary computation, sensing, and communication. Task-based models progress at the granularity of tasks. They re-execute each

```
1  NV int i, sum, x[];
2  i = sum = 0;
3  while (i < N) {
4      checkpoint();
5      sum += x[i];
6      i++;
7  }
8  sum = sum / N;
```

(a) WAR-affected code

```
1  NV int i, sum, x[];
2  sum += x[i]; // i = 0
3  i++;
4  checkpoint();
5  sum += x[i]; // i = 1
6  ↯   // power failure
7  ↻  checkpoint();
8  sum += x[i]; // i = 1
```

(b) WAR-affected execution

Fig. 2. WAR dependency example. NV marks non-volatile variables, `checkpoint()` is a checkpoint of volatile state.

Table 1. The Content of a Checkpoint of a Variety of Intermittent Systems

Model	Data Copied to/from NVRAM
Mementos [57]	Registers + Stack
DINO [41]	Registers + Stack + WAR NV variables
Chain [12]	PC + NV variables used in task
Alpaca [43]	PC + WAR NV variables used in task
Ratchet [73], Clank [29]	Registers (requires NV main memory)
Region Formation [2]	Registers + Updated variables in task

task interrupted by a power failure until it successfully finishes, only then moving on to the next task. Since these models do not rely on capturing an expensive checkpoint, they are usually faster than the checkpoint-based solution [12, 43]. Coala also follows the paradigm of the task-based programming model, where the programmer explicitly expresses the application as a sequence of tasks and the transitions between them. However, unlike previous task-based execution models, Coala's execution model exploits the harvested-while-executing energy to reduce the overhead of protecting an application against power failures.

Task-based models also suffer from data inconsistencies when WAR dependencies are involved. Prior systems tried to tackle this problem by a compiler-automated redo-logging for the variables that are part of the dependency [43] or by statically creating multiple copies of the problematic variable to ensure that no task reads and writes the same copy [12].

2.4 Costs of Previous Models

Compiler-inserted checkpoints or programmer-defined tasks can be both non-terminating and/or inefficient. If a task (or code between two checkpoints) consumes more than the fixed, maximum energy that a device can buffer, then the task will never be able to complete using buffered energy only. Such a task is non-terminating, *prevents forward progress*, and makes the program deadlock. If the task consumes far less energy than what a device can buffer, then the system may operate *inefficiently*, saving the program state more often than needed. Avoiding excessively costly, non-terminating tasks and short, high-overhead tasks is challenging, because estimating the exact energy use of an arbitrary code is complicated. Moreover, when *heterogeneous devices* with different energy buffers (e.g., 20 µF [59] to 0.1 F [76]) are considered, the challenge of approximating the optimal task size becomes much harder, because a large task on one device may become a

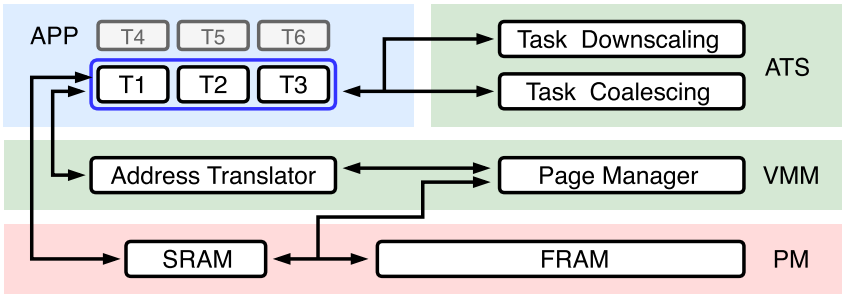


Fig. 3. Coala top-level view. APP: application; ATS: adaptive task scheduler; VMM: virtual memory manager; PM: physical memory.

small task on a device with a larger buffer. Furthermore, even when the energy buffer size is fixed, favorable energy conditions extend intermittent systems uptime, as the amount of energy being harvested while the device is on becomes not negligible. Static intermittent systems ignore this fact, while Coala takes advantage of it by coalescing a longer sequence of tasks, reducing task transitions overhead, which speeds up execution of the application.

Prior systems required additional hardware to monitor the voltage level in the energy buffer [4]. When the voltage level drops below a certain level, application execution is terminated and the computation progress is saved to non-volatile memory. Monitoring the voltage level, however, consumes energy, and allocating sufficient amount of energy (but no more) for saving the computation progress can be complicated [3]. Coala's approach is hardware independent; therefore, it is generic. Coala's runtime *coalesces* (merges) multiple static tasks to amortize the overhead when tasks are too small and *splits* a task when it is too large to complete on a single buffer charge. This on-demand task adaptation feature of Coala facilitates code portability between intermittent heterogeneous devices.

3 SYSTEM OVERVIEW

Coala is a new programming and execution model that supports adaptive task-based execution and eliminates restructuring and re-compilation of applications considering the device's energy buffer. Coala addresses the challenges given in Section 2 by making task-based intermittent applications portable in the sense of different energy storage sizes while keeping their execution efficient. Figure 3 shows an overview of Coala.

Programming and Execution Model. To use Coala, a programmer must (i) convert a plain C code into tasks by encapsulating the code in a top-level set of functions, (ii) sequence the control-flow between these tasks, and (iii) annotate memory accesses that manipulate task-shared data. Then, they compile and link the code against Coala's runtime, producing a Coala-enabled binary. The runtime relies on Coala's novel *adaptive task scheduler* to adapt its execution to the energy conditions. Facilitating efficient task adaptation requires dynamic memory protection, which Coala's *virtual memory manager* handles through *page privatization*.

Adaptive Task Scheduler. Coala's adaptive task scheduler (ATS) makes *energy-aware* scheduling decisions to group tasks together or split a task. By coalescing tasks Coala *amortizes commit and transition costs*, and by splitting a task, after it repeatedly failed to complete, it *avoids the task non-termination problem*. The scheduler uses its recent execution history, which is independent of the hardware, as a metric to estimate energy availability and eventually to decide on the coalesced task size. Section 4 describes ATS.

ALGORITHM 1: Coalescing

```
1:  C ← f_reboot(H, C)                    ▷ Update coalescing target by reboot update function
2:  H ← 0                                 ▷ Initialize history
3:  while true do
4:      j ← 0
5:      while j ≤ C do
6:          EXECUTE_TASK(T_z)             ▷ T_z is the zth task executed since the last power failure
7:          W ← f_WEIGHT(T_z)             ▷ Assign new task-dependent weight
8:          j ← j + W
9:          H ← H + W
10:     COMMIT_TO_FRAM()
11:     C ← f_COMMIT(C)                   ▷ Update coalescing target by commit update function
```

Virtual Memory Manager. Coala virtual memory manager (VMM) is the key enabler to ensure memory consistency while coalescing tasks. VMM allows applications to interface with only fast volatile memory pages and privatizes the pages demanded by a coalesced task to solve a novel data consistency problem; namely *task coalescing-induced WAR dependencies*. VMM achieves page privatization by keeping all page modifications in non-volatile memory on a coalesced task transition. Section 5 explains VMM.

4 TASK ADAPTATION

Coala's design includes a novel task scheduler. It uses efficient, energy-aware task coalescing to amortize static task overheads and a timer-based task-splitting mechanism to avoid non-termination of tasks too long for a device's energy buffer.

4.1 Task Coalescing

When a device's buffered energy is sufficient to run multiple tasks without a power failure, committing state after each task is unnecessary overhead. Coala reduces this overhead by coalescing a sequence of tasks and deferring commit operations for all tasks to the end of the sequence. In general, committing involves moving state manipulated by a task into its permanent location in non-volatile memory. In particular, Coala's commit procedure copies dirty pages of memory that a task updated from fast, volatile working memory to slower, non-volatile main memory. We defer the details of paging privatization and commit to Section 5. An effective coalescing strategy must be *aggressive* enough, attempting to coalesce a large number of tasks to amortize commit overhead. However, it must also be *conservative* enough, coalescing only as many tasks as will execute to completion given certain energy conditions, reducing the risk of re-execution penalty for long coalesced tasks.

4.1.1 Generic Design of Task Coalescing Strategies. Algorithm 1 shows the general structure of a coalescing strategy. In the algorithm, C (coalescing target) is total number of static tasks that Coala will next attempt to coalesce. T_i is the ith task executed since the last power failure. H (history) is the total number of tasks executed since the last power failure. W_i is the *weight* of a task. A task's weight is an arbitrary quantity associated with the task that represents its cost in time or energy to execute. Different coalescing strategies may apply different weights to a task, e.g., $W_i = 1\ \forall i$, or $W_i = \alpha E(T_i)$, where $E(T_i)$ is the average execution time of T_i and α is a constant.

4.1.2 Task Coalescing Strategies. Different coalescing strategies adhere mainly to the template in Algorithm 1, varying in only a few *characteristic operations* that the algorithm leaves deliberately abstract. The reboot update function, f_{reboot}, updates C, the coalescing target, after a reboot. The

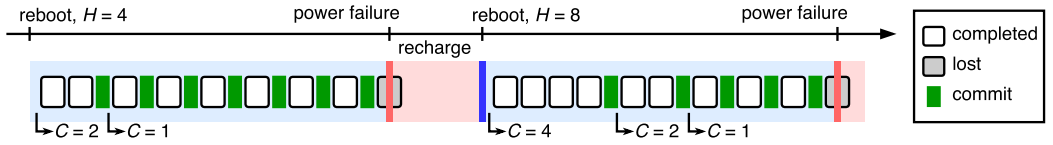


Fig. 4. EG coalescing sample execution across two power cycles. C: coalescing target; H: history.

weight lookup function f_{weight} returns a task's weight. The commit update function f_{commit} updates C after a successful commit. The following paragraphs detail three coalescing strategies that we developed for Coala, albeit not the only possible. In fact, Coala *allows programmers to design their own strategy* to implement and link against Coala's task scheduler.

Energy-Oblivious Coalescing. *Energy-Oblivious* (EO) coalescing strategy treats all tasks as having unity weight and varies the size of the coalescing target linearly. In such a scheme, the number of tasks to coalesce, C, increases by a constant x when a coalesced sequence of tasks commits and decreases by the same constant x when a coalesced sequence of tasks fails to complete, i.e., after a power failure. The characteristic operations of EO are

$$\begin{aligned} f_{\text{reboot}}(\underline{H}, \underline{C}) &= C - x, \\ f_{\text{weight}}(T_{\text{i}}) &= 1, \\ f_{\text{commit}}(C) &= C + x. \end{aligned} \tag{1}$$

EO reacts slowly to the variation in the energy required to execute different tasks and to the variation in the effective quantum of energy available to the device. With the successful commit of each coalesced task sequence, C increases. Eventually, the target may be too high and only a coalesced task composed of a few units will commit without interruption by a power failure. The strategy then linearly decreases C, eventually reaching a value that allows completion. A key limitation of this algorithm lies in the equality of the target decrease in f_{reboot} and the increase in f_{commit}. Let us assume $x = 1$. If, after k successful commits, the target must decrease to its original value $C - k$ due to an energy drop, then EO requires k successive reboots to progress.

Energy-Guided Coalescing. The *Energy-Guided* (EG) coalescing strategy adapts its coalescing target more quickly, to adhere to changes in energy conditions, addressing a key limitation of the EO. It uses its recent execution history, H, to estimate energy availability and alters its target accordingly. The EG algorithm is characterized by the following functions:

$$\begin{aligned} f_{\text{reboot}}(H, C) &= \lceil \rho H \rceil, \\ f_{\text{weight}}(T_{\text{i}}) &= 1, \\ f_{\text{commit}}(C) &= \lceil \gamma C \rceil, \end{aligned} \tag{2}$$

where $\rho, \gamma \in [0, 1]$. By relying on the history of execution EG eliminates the problem of frequent power failures on a single coalesced task. In fact, at each reboot, EG will *conservatively* try to coalesce only a fraction (ρ) of tasks that had successfully completed during the last power cycle. Figure 4 illustrates a snapshot of the operation of EG, with $\rho = \gamma = 0.5$. The figure shows two power cycles following a power cycle (not shown in the picture) whose execution history is $H = 4$. Based on that, C is initially set to 2 and then decreases to 1. Once the value of C reaches one, EG is expecting a power failure, justifying the conservative approach. After the reboot, EG uses the most recent history $H = 8$ to set $C = 4$ and continue execution.

Weighted Energy-Guided Coalescing. The *Weighted Energy-Guided* (WEG) coalescing strategy accounts for non-uniform energy and time costs of a program's tasks when setting C. Each different task in the program consumes a different amount of energy to run to completion. The EO and

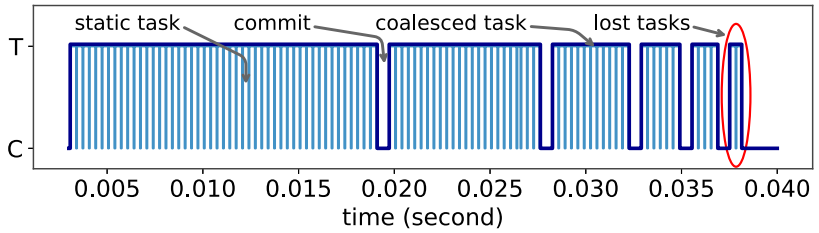


Fig. 5. Anatomy of Coala coalescing. A snapshot of measured data showing how Coala coalesces static tasks. Coala starts with a big coalesced task and shrinks it gradually as the risk of a power failure increases. Assuming an energy buffer is large enough for any single static task or an ability to split a static task (see Section 4.2), eventually the task sequence becomes short enough to complete, because after every reboot the execution always resumes with a full energy buffer. T stands for task code execution and C for commit or power cut.

EG coalescing strategies assume that each task has the same cost: For these strategies, C simply corresponds to a target *count* of tasks to coalesce, regardless of the individual cost of each task. However, if one task executes for 10 s and another executes for 1 s, counting tasks misjudges the amount of *work* in the coalesced tasks (and the history). Instead, WEG associates a non-unity weight with each task in the program and tracks the sum of the weights of tasks in a coalesced sequence. When the sum of weights reaches C, the target, WEG commits the coalesced task. WEG is characterized the same way as EG (Equation (2)), except that $f_{\text{weight}}(T_i) = W_i$.

Figure 5 shows a real measurement of Coala coalesced tasks. It shows how Coala coalesces aggressively when the execution is just re-started, taking advantage of the full energy buffer, and how it reduces the coalesced task gradually to minimize the risk of significant progress loss when a power fails. The average measured static task size is ≈0.28 ms while the commit time (time needed to save the data into non-volatile memory) averages at ≈0.64 ms. This highlights the importance of coalescing.

4.2 Task Downscaling

Coala uses *task downscaling* to make progress through a task that is too large to complete using the buffered energy. Task downscaling executes part of the long task and interrupts its execution with a *partial commit*, after which the long task continues. This timer-based solution is similar to the one proposed in [73]. If power fails, then the partial commit preserves the progress through the task, and execution resumes from the point of the partial commit rather than from the start of the task. Eventually, after some number of partial commits and power failures, the task will complete and execution will proceed.

Detection of Non-terminating Tasks. The key design issue for task downscaling is deciding when to partially commit. A task is likely to be non-terminating if it fails to run to completion twice consecutively. The second incomplete run executes after a power failure, when the device will have fully recharged its energy buffer. If a task cannot complete when executing with a full energy buffer, then Coala marks the task as non-terminating. Task downscaling violates task atomicity in a non-terminating execution, favoring continued progress over atomicity. If a programmer requires a task's atomicity to be preserved, then they can disable task downscaling for that task or a portion of it.

5 MEMORY MANAGEMENT

Resolving WAR dependencies by considering static task boundaries is not sufficient to keep non-volatile memory consistent when tasks are coalesced. Not handling WAR dependencies on a

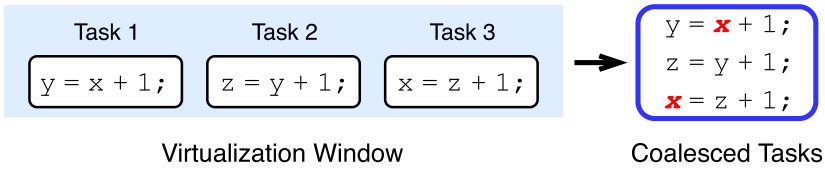


Fig. 6. Task coalescing-induced WAR dependency (in the case of this example, at variable x). Such dynamic dependencies cannot be resolved at compile time and they break memory consistency.

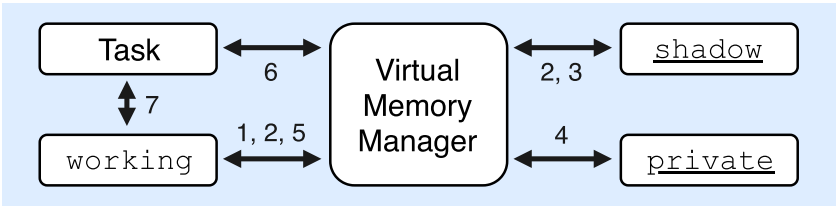


Fig. 7. Interaction among task, memory manager, and memory buffers upon RP and WP and order of occurrence. Steps 2, 3, 4, and 5 are conditional.

coalesced task scope introduces a new problem, which we denote as *task coalescing-induced WAR dependency*. This problem is illustrated in Figure 6. Therefore, Coala implements a novel *Virtual Memory Manager* (VMM) that enables safe task coalescing and ensures efficient data manipulation.

VMM overview is given in Figure 7. VMM abstracts the physical address space of the non-volatile memory (FRAM) and divides it into `private` and `shadow` (underlined locations are non-volatile) buffers, and each buffer is divided into pages. `private` holds the consistent version of pages after each commit (and on a reboot), while `shadow` enables atomic two-phase commit: it allows Coala to ensure a persistent and consistent set of non-volatile pages before copying them to `private` (Section 5.3).

The VMM prohibits applications from directly accessing these buffers. Instead, it redirects any request to the `working` buffer: a relatively small buffer located in the volatile memory (SRAM), which has a lower latency and energy cost to access than FRAM. It populates the `working` buffer with the privatized pages from non-volatile memory requested by tasks. With the help of `shadow`, the `working` buffer can serve more pages than its own capacity.

When a coalesced task completes, the VMM ensures that *dirty*, i.e., modified, pages are committed to FRAM and visible to subsequent tasks. If a power failure interrupts a task, then all temporary modifications to pages in `working` (and `shadow`) occurred after the last commit are discarded to keep data consistency. On a commit, the VMM atomically copies, through a two-phase commit, all dirty pages back into their location in the non-volatile memory.

5.1 Address Translation and Variable Access

A task must access protected non-volatile variables through Coala's restricted memory interface. The interface includes `RP`(v), to read the value of variable v, and `WP`(v) to assign a value to variable v. The implementation of `RP` is shown in Algorithm 2, and `WP`'s implementation is similar except that accessed page are marked as dirty. `RP` and `WP` operations translate a variable's physical address in non-volatile memory into a *virtual address* in `working` in volatile memory. In Coala, a virtual address is composed of a *page tag* that identifies the page and a *page offset* that identifies a byte.

After address translation (Figure 7, Step 6), a task accesses the protected variable's location in the volatile `working` buffer (Step 7). The VMM keeps track of the page tags for the pages currently

ALGORITHM 2: RP(variable v)

```
1: t ← GetTag(v)
2: i ← {j | GetTag(working[j]) = t}                    ▷ Search page
3: if i = ∅ then                                       ▷ Variable in a resident page?
4:     i ← PageFault(t)                                ▷ Swap page in
5: o ← GetOffset(v)
6: return working[i][o]                                ▷ Return from page
```

resident in the working buffer. When a task accesses a variable, it compares the variable's page tag to tags of the pages in working (Algorithm 2, Line 2). If the accessed variable's page tag is not found in the page buffer, then the operation incurs a *page fault* (Line 4). The byte is accessed in the page buffer at the index of the resident page and the variable's page offset (Lines 5 and 6).

5.2 Page Faults and Page Swapping

When accessing a protected variable with RP or WP, the memory manager first searches the variable's page in the working buffer (Figure 7, Step 1). If the page is not found there, then a page fault is incurred and a new page needs to be swapped in. If the working buffer is full, then a page fault on memory access requires the VMM to swap out one of the pages in the working buffer (a *victim page*) preserving updates made to that page. If a task modified any byte in the victim page (i.e., using WP), then the page is dirty and its changes have to be persisted to shadow in non-volatile memory (Figure 7, Step 2). If the accessed page was previously modified and swapped out since the last power failure, then the most recent version of the page is in shadow (Step 3), otherwise it has to be retrieved from private (Step 4). Finally, the page is copied to the volatile buffer (Step 5).

5.3 Atomic Two-Phase Commit of Dirty Pages

When the last task in a sequence of coalesced tasks completes, Coala must commit *all* dirty pages in working and shadow, copying them back to their original locations in private.

To make the commit atomic, Coala commits dirty pages in two phases, as shown in Algorithm 3. The first phase copies dirty pages from working to the non-volatile shadow (Line 4). The second phase commits pages from shadow to private (Line 12). If power fails during the first phase, then the whole commit is aborted, and the execution restarts from the most recently committed point (i.e., from the beginning of a coalesced task). If power fails in the second phase, then the commit process safely resumes on reboot. The second phase depends on some runtime metadata. The committing bit indicates that a commit is in progress and is set before the first page is committed (Line 9) and cleared after the last page is committed (Line 16). The shadowCount records the number of dirty shadow pages to be committed and the VMM clears the counter when commit completes (Line 14). The commitIndex indexes the next page to be committed (Line 11) and the VMM clears the index at the end of the phase (Line 15).

Coala's commit is efficient, because it maintains an index of dirty pages instead of iterating through all potentially dirty pages to check their state. Another source of efficiency lies in the second phase of the commit: The VMM does not copy page content from shadow to private; instead, it swaps pointers in an indirection table that maintains the pages as a double buffer.

Memory Consistency. Coala's paging mechanism ensures that a task only ever executes using consistent protected data. During task execution, modifications to protected data do not affect the private buffer, because a task reads and writes the volatile working buffer only, and modified pages are kept in shadow until commit. A power failure erases the contents of the working buffer, preventing a re-executing task from observing updates from a previous execution attempt.

ALGORITHM 3: Two-phase commit

```
1:  procedure CommitPhase1                                  ▷ On completion of a coalesced task
2:      for i ∈ 0..|working| − 1 do
3:          f_dirty, t ← TagFlag(working[i])
4:          if f_dirty then                                 ▷ Is the page content modified?
5:              shadow[Tag(working[i])] ←DMA— working[i]
6:              shadowList[shadowCount] ← t
7:              shadowCount ← shadowCount + 1
8:      CommitPhase2
9:  procedure CommitPhase2                                  ▷ commitIndex 0 on first boot
10:     committing ← true
11:     while commitIndex < shadowCount do
12:         t ← shadowList[commitIndex]
13:         commitToPrivate(shadow[t])          ▷ Copy shadow to private by swapping their pointers
14:         commitIndex ← commitIndex + 1
15:     shadowCount ← 0
16:     commitIndex ← 0
17:     committing ← false
18: procedure OnBoot                                        ▷ Invoked on every boot
19:     if committing then CommitPhase2
20:     shadowCount ← 0
```

Clearing shadowCount as part of the second phase commit (Line 15) ensures that all accesses to protected variables in subsequent tasks correctly access their consistent memory locations in private. This solves the coalescing-induced WAR dependency problem (see Figure 6 again).

5.4 Dynamic Paging in Coala

Coala asks the programmer to use its RP and WP API methods on every access to a protected variable (Section 6.1). These API invocations present a risk of high overhead, because there is a dynamic check on every read and write. Despite the risk of per-access overhead, Coala's dynamic memory protection scheme brings several benefits over a static approach (i.e., [43]). First, the limitations of static analysis preclude some uses of pointers due to potential pointer aliasing. For example, in the presence of arbitrary pointer operations, a function call using a function pointer, or an interrupt within a task, the system cannot statically analyze the memory behavior. Second, a static approach *cannot handle task coalescing*, because a protected variable's lifetime, i.e., from first use to commit, is unknown at compile time. Coala's dynamic, per-access instrumentation supports arbitrary use of pointers and enables task coalescing.

6 IMPLEMENTATION

We implemented Coala's programming and execution model as a runtime library and API that a programmer can use to make a plain C program intermittency-safe.

6.1 Application Programming Interface

Coala's API adds only a few syntactic constructs to a C-based language, summarized in Table 2.
New Tasks. The TASK annotation on a function declaration statically allocates a non-volatile constant variable holding a task's weight and declares that the function is a task.

Table 2. API Summary; **T**: Set of all Tasks, **V**: Set of all Protected Variables, [, s]: Optional Argument

Method	Arguments
INIT(t)	$t \in \mathbf{T}$: scheduled task on first boot
RUN()	—
TASK(t, w_t)	$t \in \mathbf{T}$: task name, w_t: weight of task t
NEXT_TASK(t)	$t \in \mathbf{T}$: task to be run next
PV(p, v [, s])	p: type, $v \in \mathbf{V}$: name, s: array size
u := RP(v)	$v \in \mathbf{V}$: protected variable to read, u: dest. operand
WP(v) := u	$v \in \mathbf{V}$: protected variable to write, u: source operand
SM(p, m [, s])	p: type, m: name, s: array size
DISABLE_PC()	—
ENABLE_PC()	—

Task Transitions. NEXT_TASK marks the task to be executed after the current one and it can be invoked along any control path to dynamically determine the next task.
Protected Variables. The PV annotation on a variable statically allocates a protected non-volatile variable. The variable must then be accessed with the RP and WP API methods at runtime to ensure correct operation. The SM annotation is used to align C structure data type within a page.
Initialization. The behavior of the API method INIT is very similar to NEXT_TASK, with the addition of performing preliminary kernel initializations, including hardware setup.
Execution. The programmer passes control to Coala's task scheduler by calling RUN after device initialization.
Partial Commit. The DISABLE_PC() and ENABLE_PC() allow a programmer to disable partial commit around certain code.

6.2 Initialization Procedure

On a reboot, Coala's scheduler does a number of system level operations before executing a task. First, it updates the coalescing target according to the applied coalescing strategy. Then, it finishes any interrupted commit and clears the list of dirty shadow pages. After that, the scheduler sets the program counter to the next task to run, which Coala tracks in non-volatile memory. Before executing the task, Coala checks whether there is a partially committed task to resume, which requires Coala to restore the volatile state, including the program counter. If there is no in-progress, partially committed task, then Coala starts executing and coalescing tasks.

6.3 Task Coalescing

Parameters. Equations (1) and (2) parametrize the behavior of the coalescing strategies in terms of x, ρ, and γ. We experimented with a range of values and then used the ones that yielded the best performance: $x = 1$ (for EO) and $\rho = \gamma = 1$ (for EG and WEG).

Weights for WEG. The effectiveness of the WEG hinges on correctly identifying the weight of each task, which WEG assumes is *statically* available. Profiling the time and energy cost of tasks in a program is a difficult, orthogonal problem [2, 13]. WEG could use the result of an arbitrarily sophisticated profiling procedure. To produce a concrete result in this article, we give WEG access to a simple profile of task runtime (collected offline) using a single fixed input. WEG stores the profile in a lookup table that maps a task's identifier to its weight, making the information available to Coala's scheduler at runtime.

6.4 Task Downscaling

Timer Strategy. After identifying a task as likely non-terminating, Coala must decide when during the task's execution to partially commit the task's state. Coala sets a timer at the start of the likely non-terminating task, initializing it to a very large value (e.g., an estimate of the maximum execution time using the device's energy buffer). If the timer expires before power fails, then Coala partially commits, retaining the timer value to use for future partial commits. If the timer does not expire before power fails, then Coala halves the timer and tries again. The exponential decrease of the timer value converges rapidly to a usable one.

Partial Commit and Task Atomicity. Some applications may prevent downscaling a task because of a need for task atomicity. For example, sampling an analog signal requires consecutive samples at a known interval, or the digitally sampled signal is meaningless. In such a case, the programmer can disable partial commit for a task or a span of code, marking the code with a pair of `DISABLE_PC` and `ENABLE_PC` annotations. These annotations respectively halt and resume the partial commit timer.

6.5 Paging

Efficiency. Coala uses address-based page tagging to make finding a variable efficient. The upper bits of a variable's memory address identify its page, and the lower bits denote the variable's offset in its page. The total number of pages in memory, P, determines the number of tag bits, which is $\log_2 P$. Furthermore, the VMM moves pages of data efficiently using hardware-accelerated DMA support.

Alignment. Page tagging imposes a data alignment requirement. The page size S must be a power of two. Pages must be aligned to an S-byte boundary for efficient memory access. To preserve alignment, when typedef'ing a C structure for protected variables, the programmer has to use the API method `SM` on all members of the structure (only when defining the structure).

Page Eviction. When a page in `working` has to be swapped out to make room for a newly requested one, an eviction policy has to be chosen. While we opted for the simplicity of FIFO, any other replacement policy, such as Least Recently Used, could work in its place.

We experimented with 32-, 64-, 128-, and 256-B pages, an 8-KB non-volatile `shadow` buffer and an 8-KB non-volatile `private` buffer, and a `working` buffer of 1 KB.

7 METHODOLOGY

We prototype Coala and use its API to implement a set of benchmark applications representative of the embedded domain. We build the applications and deploy the binaries onto a real energy-harvesting device.

7.1 Experimental Setup

We used three different setups to evaluate Coala: (i) an RF-powered energy-harvesting device, WISP 5.1 [51, 62], with a fixed capacitor size of 47 μF; (ii) an MSP-EXP430FR5969 launchpad [68] powered by a BQ25570 [67] solar power harvester that is connected to an IXYS SLMD121H04L solar cell [32] for experimenting with different energy buffers; and (iii) an MSP-EXP430FR5969 launchpad [68] that is continuously powered.

Every benchmark used in Section 8 was run repeatedly on each platform for a few minutes to ensure capturing a diverse power trace. The exact number of iterations depends on the ambient power intensity and the application itself. In our experiments, the number of complete runs ranges from 4 to 125.

Table 3. Characteristics of Benchmarks Used for Evaluation

App.	Tasks	SLOC	Description
ar	10	428	activity recognition using a KNN
bc	10	371	several bitcount algorithms
cuckoo	14	426	Cuckoo Filter with pseudo-random values
dijkstra	5	198	Dijkstra shortest path algorithm
fft	8	449	Fast Fourier Transform
sort	4	167	selection sort algorithm

WISP contains the MSP430FR5969 [69] MCU with 64 KB of non-volatile (FRAM) memory and 2 KB of volatile (SRAM) memory and was configured to 1-MHz clock speed. We powered WISP using an RF signal generator emitting a 20-dBm sinusoidal wave at 915 MHz. The signal generator was connected to the Laird RFMAX S9028PCRJ 8-dBic antenna [58]. The antenna was oriented towards and in parallel with WISP's antenna, and no objects obstructed the path. We affixed WISP with a paper harness at the edge of a table at a height, from the table surface, of 10 cm. For distance-controlled experiments we positioned WISP at $d = \{15, 30, 50\}$ cm from the exciter antenna. To obtain execution time, the software toggled GPIO pins at sections of the code under profile, and the Saleae [60] logic analyzer measured intervals between edges in the signal. For continuous power experiments, the execution time was measured using the clock features in TI Code Composer Studio IDE version 7.1.

7.2 Software Benchmarks

We evaluated Coala using six benchmarks that are often used in embedded systems (summarized in Table 3). All applications were compiled using MSP430 GCC [72] version 6.4.0 with `-O1` as optimization flag. The source code for all benchmarks is released in [10].

Comparison with Alpaca Using the GCC Compiler. We compare Coala against Alpaca [43], the state-of-the-art task-based system for intermittent computing. For a fair comparison, the task decomposition of the benchmarks is ensured to be the same for both systems. Since Alpaca's compiler pass is implemented only for LLVM, we could not use that implementation to instrument the benchmarks and compile them with GCC. Instead, we *manually* performed the instrumentation done by the compiler pass. The instrumentation consists of identifying WAR dependencies and adding code and memory allocations to make a private copy of the affected variables. By compiling both systems with the same compiler (GCC), we ensure that our comparison in Section 8 is fair.

8 EVALUATION

Our evaluation quantitatively demonstrates that Coala (i) *reduces memory protection overhead*, (ii) *improves execution speed* in most cases, and (iii) *is able to progress* where static systems suffer from a task non-termination loop.

8.1 Characterization of Overhead

To characterize Coala's overhead we experimented with WISP positioned at 15 cm away from the signal generator antenna. We have broken down Coala's overhead to explain the source of its improved performance.

Overhead Reduced by Coalescing. For each coalescing strategy from Section 4 (EO, EG, and WEG) and for a baseline without coalescing (NC), we have measured the time spent on executing (i) useful

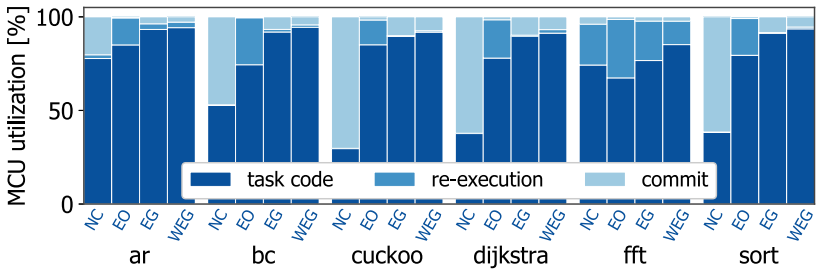


Fig. 8. Breakdown of overhead for the proposed coalescing strategies: NC (no coalescing), EO (energy-oblivious), EG (energy-guided), and WEG (weighted energy-guided). All coalescing strategies reduce total overhead and maximize useful work. EG and WEG perform better than EO.

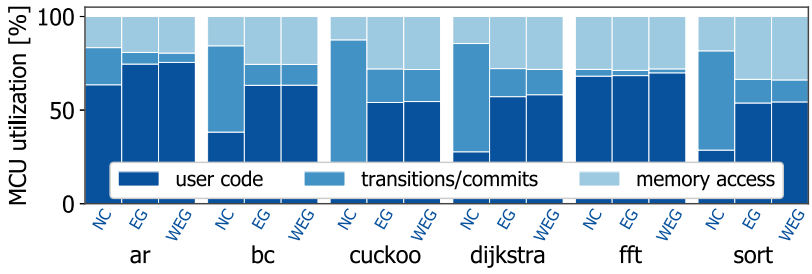
task code, (ii) task code wasted due to a power failure, and (iii) commits to non-volatile memory at the end of each (coalesced) task. The overhead incurred by each coalescing strategy is broken down in Figure 8. Without coalescing enabled (NC), the re-execution penalty is the smallest, because the amount of work that can happen between commits and may have to be re-executed if interrupted is reduced when work from multiple static tasks is not combined. However, any gain from a reduced re-execution penalty is canceled out by the increased commit overhead that is incurred at the end of each static task. Across all benchmarks, all Coala's coalescing strategies reduce more commit overhead than the re-execution overhead they add. This net overhead reduction is greatest in EG and WEG strategies compared to the EO strategy. We attribute this discrepancy to EO's energy-unaware adjustment to the coalescing target. In the subsequent experiments, we focus on the better-performing EG and WEG.

Coala's Kernel Overhead. Figure 9(a) breaks down the time spent on executing user task code versus the time spent on kernel operations. When coalescing is not enabled the commit overhead is highest. The overhead for memory accesses increases in percentage when enabling coalescing, but not in absolute terms. For both coalescing strategies (EG and WEG) accesses to protected variables constitute about 30% of the runtime overhead. Dynamic address translation necessary on each protected access is the most critical bottleneck for Coala.

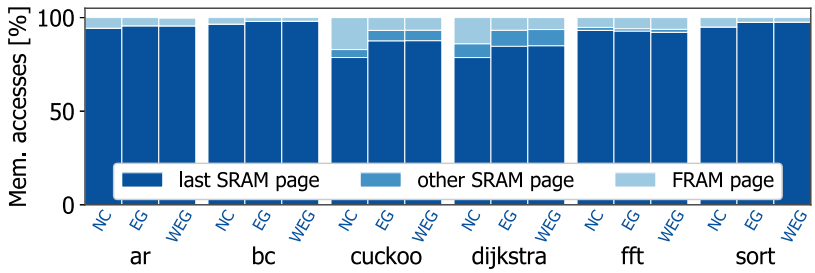
Protected Memory Accesses Breakdown. Figure 9(b) breaks down protected memory accesses into three categories. Each type of protected access incurs a different overhead. Accessing the most recently used SRAM page is of the cheapest kind. Accessing a different page in SRAM has a slightly higher cost. Finally, accessing a page that needs to be swapped in from FRAM into SRAM is the most expensive. The results in the figure show that the overwhelming majority of accesses are of the cheapest kind, which motivated us to optimize this accesses in our implementation. Only *cuckoo*, *dijkstra*, and *fft* have non-negligible number of accesses to a different SRAM page, which is due to the larger working set and a less regular access pattern in these applications. In general, memory access patterns are shaped by the application, and the more program state is protected, the higher the rate of page swaps.

8.2 Execution Time

Having shown in Section 8.1 that coalescing reduces overhead, we now investigate the outcome of this reduction on the total execution time. We first investigate different variants of Coala and then compare the best variant to Alpaca [43]. Additionally, we compare Coala performance running with different energy buffer sizes.



(a) Kernel overhead breakdown



(b) Protected memory accesses breakdown

Fig. 9. Coala's internal overhead. NC, no coalescing; EG, energy-guided; WEG, weighted energy-guided.

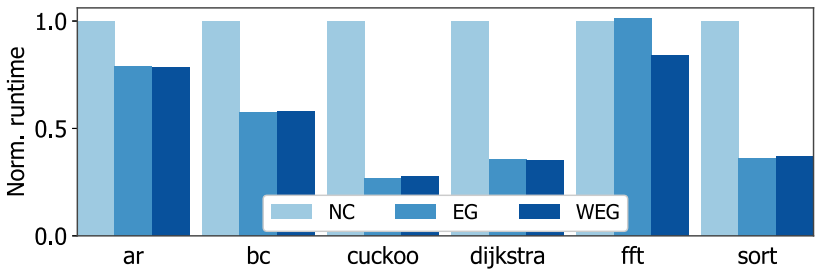


Fig. 10. Coala's coalescing performance. Application execution time with coalescing (EG and WEG) normalized to the execution time without coalescing (NC).

Speedup with Coalescing. Figure 10 shows Coala's runtime of two coalescing strategies (EG, WEG) normalized to the runtime without coalescing (NC). The results show that all benchmarks complete faster with coalescing than without coalescing: from 25% (*ar*) up to 70% (*sort*). This speedup is a consequence of the reduced overhead demonstrated in Section 8.1. However, the magnitude of the speedup is (1) highly application-dependent and (2) largely similar across the two coalescing strategies, with the exception of *fft*. In some cases (*bc*, *cuckoo*, *sort*) WEG's task weighting system is counter-productive. This occurs in task decompositions with energy-uniform tasks, where counting tasks disregarding their energy consumption provides an equal amount of information with a smaller effort. In *fft*, tasks are not uniform, and accounting for their different weights is beneficial. In fact, the lack of task energy awareness is detrimental: With EG *fft* runs slower than without any coalescing (NC). The speedup is highest for *bc*, *cuckoo*, *dijkstra* and *sort*, because their tasks are relatively small and are easily coalesced.

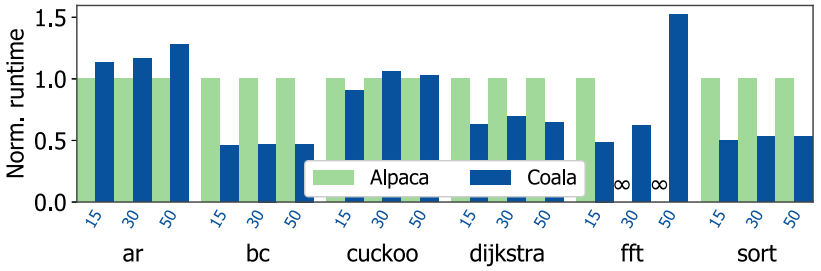


Fig. 11. Coala's execution time normalized to Alpaca's, for three distances from the energy source (15, 30, and 50 cm).

Table 4. Comparison of Coala Performance Running the *Sort* Application on Two Different Capacitor Sizes

Cap. size (μF)	**Exp. time** (s)	**On/Off cycle**	**Coalesced task** (ms)	**Runs**	**Runtime** (ms)
47	1205	12.93%	33	85	183
470	1205	12.79%	88	87	177

The results show that Coala optimizes its coalesced task size based on the energy buffer size. **Coalesced task** refers to the length of the first coalesced task after a reboot, **Exp. time** shows the experiment duration, the **Runs** column lists the number of complete runs of the application during the experiments. **Runtime** is the device collective uptime needed to finish a single iteration of the *sort* application. On/Off cycle is the ratio of time the device was powered (On) to total experiment time.

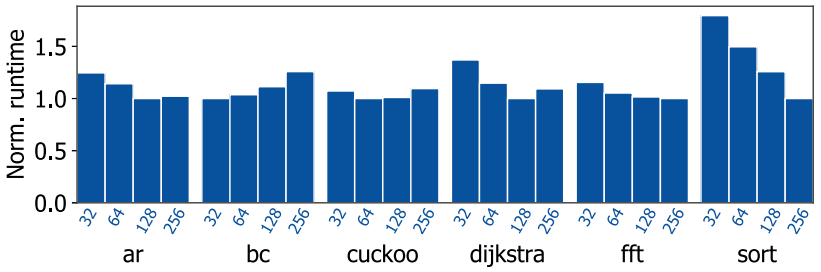
Benefits of Adaptive Tasks. We now compare Coala's performance to Alpaca [43]—a *non-adaptive* task-based system with tasks fixed at compile time. Figure 11 shows the average execution time of each application for Coala and Alpaca, normalized to the latter or, when not possible, to 1 s. Coala provides a performance benefit compared to Alpaca for most applications. For example, it is 54% faster than Alpaca when executing the *bc* application. In general, the speedup is greatest for applications with repeated WAR dependencies throughout their code, particularly involving arrays (*dijkstra*, *fft*, and *sort*). Coala's VMM successfully amortizes the overhead of protecting memory that is accessed in such patterns. In applications without locality among accesses to protected variables Coala incurs overhead from memory virtualization that causes its performance to be comparable to (or worse than) Alpaca (*ar*, *cuckoo*).

Due to its static progressing behavior, Alpaca was unable to complete the *fft* benchmark on distances larger than 15 cm[2]. This is marked with ∞ signs in Figure 11. Coala, however, managed to complete *fft* by enabling its task downscaling at 30 and 50 cm from the energy source.

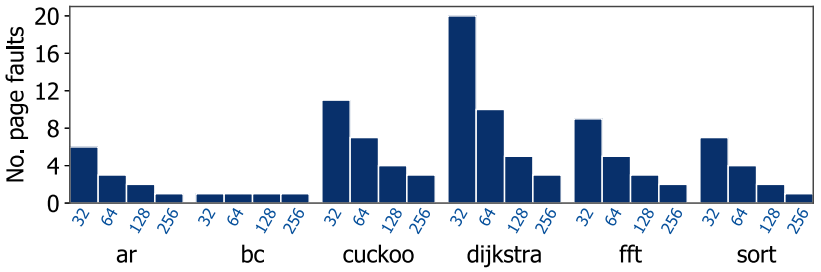
Different Capacitor Sizes. Table 4 shows how Coala optimizes its Coalescing task size based on the amount of buffered energy at runtime. This means that applications implemented in Coala are portable across devices with different capacitor sizes without recompilation; they are also more resilient to degradation in capacitor size due to temperature and device lifetime.

We see that Coala scales up its coalesced task size with a bigger energy buffer and vice versa. This allows it to reduce the time-to-completion of the applications. For example, the *sort* runtime is reduced from 183 to 177 ms when the capacitor is changed from 47 to 470 μF. It should be emphasized that a device with a bigger energy buffer suffers less from power failures (but requires longer charging time). Note that while the on intervals and the off intervals are both longer for the larger capacitor, the fraction of time the device is executing the application as opposed to charging

[2]At distances less than 15 cm the total amount of energy available for task execution includes significant amount of energy being harvested while the device is executing in addition to the stored energy.



(a) Execution time normalized to lowest per-application



(b) Number of page faults per application run

Fig. 12. Effect of page size (in bytes).

(On/Off Cycle column) is nearly the same across the two capacitors, because it is a function of the average input power.

Overall, Coala shows better performance than its counterpart, and it is able to overcome the *big task* (i.e., *fft* tasks) problem that static task-based systems suffer from.

8.3 Virtual Memory Performance

We characterize the performance of Coala's virtual memory sub-system in an experiment on a continuously powered evaluation board, as described in Section 7.1.

Effect of Page Size on Runtime. Figure 12(a) shows the execution time as a function of page size (in bytes), normalized to the lowest per-application performance among the set of page sizes. The data suggest that there is a page size that minimizes execution time. The best page size is not the same for each application. Nevertheless, if a choice must be made for all applications, then 128B pages are the best option.

Effect of Page Size on Page Faults. Figure 12(b) reports the number of page faults, per application run, as a function of page size (in bytes). The smaller the page the more likely that a memory access will land outside that page and that a new page will have to be swapped in. This trend is visible for all applications, except for *bc*. The total amount of data accessed by *bc*, as well as its working set, is small. Even with the smallest page, all accesses are contained within that page, and no page fault occurs. Without any page faults to begin with, increasing the page size only yields overhead.

9 RELATED WORK

Intermittently powered Devices. There is a large body of research on energy-harvesting and ambient-powered embedded devices summarized in [31, 34, 35, 55, 61, 74]. For instance, a new

wave of embedded systems powered by radio waves is emerging [19, 30, 39, 45, 49–51, 53, 59, 70, 76, 77]. The emergence of such hardware platforms has led to the development of instrumentation, debugging, and prototyping tools for such systems [1, 11, 14, 25–27, 66].

Checkpointing-based Systems. Early work on energy-harvesting runtimes, such as Dewdrop [8], assumed simple computations that complete on a predictable burst of energy. Support for computation that spans power failures was first achieved with statically placed conditional checkpoints in Mementos [57]. DINO [41] addressed the consistency problem by selectively versioning non-volatile state within the checkpoints. Ratchet [73] ensured consistency by placing a checkpoint at the beginning of each idempotent region in the code. Ratchet has a similar technique to task splitting with a core difference: The checkpoint is of a fixed size, while Coala privatizes a varying number of memory pages. Clank [29] ensured consistency with custom hardware that dynamically tracks WAR dependencies in memory accesses and checkpoints on demand. The consistency problem was also approached with a combination of undo- and redo-logging in software [2]. Just-in-time checkpointing, such as Quickrecall [33] and Hibernus++ [3], eschews inconsistency by saving all volatile state immediately before a power failure and halting the execution. Unlike Coala, such systems rely on introspection hardware to monitor supply voltage and on accurate worst-case bounds on checkpoint cost.

In all of the above systems, with the exception of DINO [41], checkpoints are dynamic, i.e., the programmer does not have explicit control over the point at which the code may be resumed after power failure. Dynamic checkpointing systems make it difficult for the programmer to respect application-level atomicity constraints, such as correlating sensor readings. Checkpointing systems that copy most volatile state scale poorly as the size of volatile memory increases. QuickRecall, Clank, and Ratchet reduce the copying overhead by allocating the stack in non-volatile memory, which requires more time and energy to access than volatile memory (cf. Section 2.4) and not a viable option for off-chip non-volatile memory. In addition, QuickRecall and Clank require custom hardware.

Task-based Systems. Alternatives to checkpointing are recent systems based on static tasks, such as Chain [12], Alpaca [43], InK [75], and Mayfly [28]. Using static tasks, they eliminate the need to checkpoint volatile state. Using channel-based memory models [12, 28] or automatic privatization and redo-logging [43] they avoid checkpointing overheads. Moreover, task-based models facilitate respecting application-level atomicity constraints. Coala also relies on statically defined tasks to avoid checkpointing volatile state. However, unlike prior systems, Coala coalesces its statically defined tasks at runtime into more efficient dynamic tasks that adapt to changing energy conditions. Coala's mechanism for ensuring memory consistency also differs from the channel-based [12] and privatization-based [43] mechanisms in prior systems. Coala keeps memory consistent through memory virtualization optimized for bulk accesses to task-shared data with high locality.

Task Decomposition. In contrast to Coala's construction of coalesced tasks at runtime, prior work has proposed to optimize task size at compile time. CleanCut [13] program analysis statically estimates energy consumption and splits the program into tasks until all tasks consume less energy than the device can store. An alternative program analysis generates different versions of a program with different task sizes and empirically selects the best among them [2]. HarvOS [7] takes a hybrid approach that uses a program analysis to place a minimal number of conditional checkpoints that test the energy level at runtime before copying state, like Mementos [57]. Unlike Coala, such compiler-based approaches face the challenge of statically predicting energy consumption of arbitrary input-dependent code with peripheral access, which is a problem without a general solution. Furthermore, a static decomposition approach prevents portability across devices with

different storage capacitors. In contrast, Coala avoids forcing any assumptions at compile time and adapts to energy storage capacity and incoming energy conditions at runtime.

Memory Virtualization. Prior work on embedded systems has studied a variety of memory virtualization strategies relating to Coala. TinyOS [38] and nesC [18] support dynamic memory management. Later work extended the memory manager to support memory virtualization backed by flash memory [36] and to ensure memory and type safety [15]. SOS [24], Contiki [16], and T-kernel [22] also developed memory management abstractions that virtualize memory size and provide safe and indirect access. Maté [37] developed full virtual machine support for sensor nodes, virtualizing not just memory resources, but other states and peripherals. The goal of Coala is to provide consistent, intermittent execution, leveraging the benefits of efficient bulk copying. In contrast, prior efforts focused more on programmability and runtime reliability properties provided by virtual memory.

A related domain is unbounded, page-based transactional memory and deterministic parallel runtime systems [6, 9]. These works have a different mechanism and purpose than Coala—ensuring that data are consistent and deterministically updated during concurrent executions. Their similarity with Coala is in managing state to ensure consistency at the granularity of pages to amortize checking and tracking costs. Coala's paging implementation, which keeps a shadow page for each page to use during commit, is similar to the shadow paging scheme used for transactional commit [9].

10 CONCLUSIONS

Software for intermittently powered energy-harvesting devices requires a dedicated runtime system. Coala is a new task-based system whose distinguishing feature is its adaptability to changing energy conditions at runtime. When more energy is available, Coala makes faster progress through the computation by coalescing statically defined tasks. When less energy is available, progress is latched at sub-task granularity. Coala's page privatization system ensures that program state in non-volatile memory remains consistent and amortizes the cost of state transfer between volatile and non-volatile memory. Our evaluation across applications on a real embedded energy-harvesting device demonstrates the utility of Coala, as well as the practicality of the proposed implementation.

REFERENCES

[1] Henko Aantjes, Amjad Y. Majid, Przemysław Pawełczak, Jethro Tan, Aaron Parks, and Joshua R. Smith. 2017. Fast downstream to many (Computational) RFIDs. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'17)*. IEEE.

[2] Sara S. Baghsorkhi and Christos Margiolas. 2018. Automating efficient variable-grained resiliency for low-power IoT systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. ACM, 38–49.

[3] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 12 (Dec. 2016), 1968–1980.

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

[5] Amay J. Bandodkar, Wenzhao Jia, and Joseph Wang. 2015. Tattoo-based wearable electrochemical devices: A review. *Electroanalysis* 27, 3 (Mar. 2015), 562–572.

[6] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA'09)*. ACM.

[7] Naveed Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded devices. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'17)*. ACM/IEEE.

[8] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An energy-aware runtime for computational RFID. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI'11)*. USENIX, 197–210.

[9] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. 2006. Unbounded page-based transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. ACM, 347–358.

[10] Coala Project. 2018. Coala Website. Retrieved April 12, 2018 from https://github.com/TUDSSL/Coala.

[11] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, 577–589.

[12] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA'16)*. ACM, 514–530.

[13] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the International Conference on Compiler Construction (CC'18)*. ACM, 38–49.

[14] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM.

[15] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. 2007. Efficient memory safety for TinyOS. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'07)*. ACM, 205–218.

[16] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN'04)*. IEEE.

[17] Farsens. [n.d.]. Medusa-M2233 UHF RFID battery-free device. Retrieved July 11, 2017 from http://www.farsens.com/en/products/medusa-m2233.

[18] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, 1–11.

[19] Shyamnath Gollakota, Matthew Reynolds, Joshua Smith, and David Wetherall. 2014. The emergence of RF-powered computing. *Computer* 47, 1 (Jan. 2014), 32–39.

[20] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis, and Gil Zussman. 2014. Movers and shakers: Kinetic energy harvesting for the internet of things. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM.

[21] Maria Gorlatova, Aya Wallwater, and Gil Zussman. 2013. Networking low-power energy harvesting devices: Measurements and algorithms. *IEEE Trans. Mobile Comput.* 12, 9 (Sep. 2013), 1853–1865.

[22] Lin Gu and John A. Stankovic. 2006. T-kernel: Providing reliable OS support to wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*. ACM, 1–14.

[23] Deniz Gündüz, Kostas Stamatiou, Nicolo Michelusi, and Michele Zorzi. 2014. Designing intelligent energy harvesting communication systems. *IEEE Commun. Mag.* 52, 1 (Jan. 2014), 210–216.

[24] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*. ACM, 163–176.

[25] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'14)*. ACM, 330–331.

[26] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'15)*. ACM, 5–16.

[27] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'17)*. ACM.

[28] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'17)*. ACM, 17:1–17:13.

[29] Matthew Hicks. 2017. Clank: Architectural support for intermittent computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*. ACM.

[30] Jeremy Holleman, Dan Yeager, Richa Prasad, Joshua R. Smith, and Brian Otis. 2008. NeuralWISP: An energy-harvesting wireless neural interface with 1-m range. In *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS'08)*. IEEE.

[31] Kaibin Huang and Xiangyun Zhou. 2015. Cutting the last wires for mobile communications by microwave power transfer. *IEEE Commun. Mag.* 53, 6 (Jun. 2015), 86–93.

[32] IXYS Corporation. 2016. IXOLAR^{TM} High Efficiency SLMD121H04L Solar Module. Retrieved Oct. 2, 2019 from http://ixapps.ixys.com/DataSheet/SLMD121H04L_Nov16.pdf.

[33] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Proceedings of the International Conference on Embedded Systems Design*. IEEE, 330–335.

[34] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor C. M. Leung, and Yong Liang Guan. 2015. Wireless energy harvesting for internet of things. *IEEE Commun. Mag.* 53, 6 (Jun. 2015), 102–108.

[35] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. 2016. Advances in energy harvesting communications: Past, present, and future challenges. *IEEE Commun. Surveys Tuts.* 18, 2 (Second Quarter 2016), 1384–1412.

[36] Andreas Lachenmann, Pedro José Marrón, Matthias Gauger, Daniel Minder, Olga Saukh, and Kurt Rothermel. 2007. Removing the memory limitations of sensor networks with flash-based virtual memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*. ACM.

[37] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, 85–95.

[38] Philip Levis, Sam Madden, Joseph Polastre, Rober Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. 2005. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, Werner Weber, Jan M. Rabaey, and Emile Aarts (Eds.). Springer, Berlin, 115–148.

[39] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. 2013. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM Special Interest Group on Data Communications Conference (SIGCOMM'13)*.

[40] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent computing: Challenges and opportunities. In *Proceedings of the Summit on Advances in Programming Languages (SNAPL'17)*.

[41] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM.

[42] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 526–537.

[43] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. In *Proceedings of the Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA'17)*. ACM.

[44] Amjad Yousef Majid, Michel Jansen, Guillermo Ortas Delgado, Kasım Sinan Yıldırım, and Przemysław Pawełczak. 2019. Multi-hop backscatter tag-to-tag networks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'19)*. IEEE, 721–729.

[45] Zachary Manchester. 2016. KickSat. Retrieved October 2, 2019 from https://kicksat.github.io/.

[46] Robert Margolies, Maria Gorlatova, John Sarik, Gerald Stanje, Jianxun Zhu, Paul Miller, Marcin Szczodrak, Baradwaj Vigraham, Luca Carloni, Peter Kinget, Ioannis Kymissis, and Gil Zussman. 2015. Energy-harvesting active networked tags (EnHANTs): Prototyping and experimentation. *ACM Trans. Sen. Netw.* 11, 4 (Nov. 2015), 62:1–62:27.

[47] Robert Margolies, Guy Grebla, Tingjun Chen, Dan Rubenstein, and Gil Zussman. 2016. Panda: Neighbor discovery on a power harvesting budget. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'16)*. IEEE.

[48] Azalia Mirhoseini, Ebrahim M. Songhori, and Farinaz Koushanfar. 2013. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom'13)*. IEEE, 216–224.

[49] Phillip Nadeau, Dina El-Damak, Dean Glettig, Yong Lin Kong, Stacy Mo, Cody Cleveland, Lucas Booth, Niclas Roxhed, Robert Langer, Anantha P. Chandrakasan, and Giovanni Traverso. 2017. Prolonged energy harvesting for ingestible devices. *Nat. Biomed. Eng.* 1, 22 (Feb. 6, 2017), 1–8.

[50] Saman Naderiparizi, Aaron N. Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R. Smith. 2015. WISPCam: A battery-free RFID camera. In *Proceedings of the IEEE Conference on Radio Frequency Identification (RFID'15)*. IEEE, 166–173.

[51] Aaron Parks, Ivar in 't Veen, Saman Naderiparizi, and Jethro Tan. 2014. WISP 5.0 Firmware Git. Retrieved July 10, 2017 from https://github.com/wisp/wisp5.

[52] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. 2014. Turbocharging ambient backscatter communication. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'14)*.

[53] Shwetak N. Patel and Joshua R. Smith. 2017. Powering pervasive computing systems. *IEEE Perv. Comput.* 16, 3 (2017), 32–38.

[54] Dimitris Patoukas, Kasim Sinan Yildirim, Amjad Yousef Majid, Josiah Hester, and Przemysław Pawełczak. 2018. Feasibility of multi-tenancy on intermittent power. In *Proceedings of the International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys'18)*. ACM, 26–31.

[55] R. Venkatesha Prasad, Shruti Devasenapathy, Vijay S. Rao, and Javad Vazifehdan. 2014. Reincarnation in the ambiance: Devices and networks with energy harvesting. *IEEE Commun. Surv. Tuts.* 11, 1 (First Quarter 2014), 195–213.

[56] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*. Article 5, 1–3.

[57] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2012. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, 159–170.

[58] RFMAX. 2015. S9028 Technical Specification. Retrieved from April 13, 2018 from http://rfid.atlasrfidstore.com/hubfs/Tech_Spec_Sheets/RFMAX/ATLAS_RFMax_S9028.pdf.

[59] Saul Rodriguez, Stig Ollmar, Muhammad Waqar, and Ana Rusu. 2016. A batteryless sensor ASIC for implantable bio-impedance applications. *IEEE Trans. Biomed. Circ. Syst.* 10, 3 (Jun. 2016), 533–544.

[60] Saleae. 2017. Logic 16 Analyzer. Retrieved from July 28, 2017 from http://www.saleae.com.

[61] Alanson P. Sample, Benjamin H. Waters, Scott T. Wisdom, and Joshua R. Smith. 2013. Enabling seamless wireless power delivery in dynamic environments. *Proc. IEEE* 101, 6 (Jun. 2013), 1343–1358.

[62] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.* 57, 11 (Nov. 2008), 2608–2615.

[63] Tolga Soyata, Lucian Copeland, and Wendi Heinzelman. 2016. RF energy harvesting for embedded systems: A survey of tradeoffs and methodology. *IEEE Circ. Syst. Mag.* 16, 1 (First Quarter 2016), 22–57.

[64] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. 2017. Nonvolatile processors: Why is it trending. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'17)*. IEEE, 966–971.

[65] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R. Smith. 2017. Battery-free cellphone. *Proc. ACM Interact. Mob. Wear. Ubiq. Technol.* 1, 2 (Jun. 2017), 25:1–25:20.

[66] Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R. Smith. 2016. Wisent: Robust downstream communication and storage for computational RFIDs. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'16)*. IEEE, 1–9.

[67] Texas Instruments. 2014. Ultra Low Power Management IC, Boost Charger Nanopowered Buck Converter Evaluation Module. Retrieved Oct. 2, 2019 from http://www.ti.com/tool/BQ25570EVM-206.

[68] Texas Instruments, Inc.[n.d.]. MSP-EXP430FR5969 Launchpad. Retrieved April 13, 2018 from http://www.ti.com/tool/MSP-EXP430FR5969.

[69] Texas Instruments Inc. 2014. Overview for MSP430FRxx FRAM. Retrieved July 10, 2017 from http://ti.com/wolverine.

[70] Stewart J. Thomas, Reid R. Harrison, Anthony Leonardo, and Matthew S. Reynolds. 2012. A battery-free multichannel digital Neural/EMG telemetry system for flying insects. *IEEE Trans. Biomed. Circ. Syst.* 6, 5 (Oct. 2012), 424–435.

[71] TI Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). Retrieved July 26, 2017 from http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[72] TI Inc. 2018. TI GCC. Retrieved April 17, 2018 from http://www.ti.com/tool/MSP430-GCC-OPENSOURCE.

[73] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX, 17–32.

[74] Hubregt J. Visser and Ruud J. M. Vullers. 2013. RF energy harvesting and transport for wireless sensor network applications: Principles and requirements. *Proc. IEEE* 101, 6 (Jun. 2013), 1410–1423.

[75] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'18)*. ACM, 41–53.

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

[77] Yi Zhao, Joshua R. Smith, and Alanson Sample. 2015. NFC-WISP: A sensing and computationally enhanced near-field RFID platform. In *Proceedings of the IEEE Conference on Radio Frequency Identification (RFID'15)*. IEEE.

Received January 2019; revised June 2019; accepted September 2019