not a single PL term in title

Coala: Adaptive Task Coalescing for Intermittent Computing on Energy-harvesting Devices

relevance?

Anonymous Author(s)

Abstract

Computation consistency on intermittently-powered devices can be ensured by: (i) checkpointing, where the volatile state of a program is frequently saved to non-volatile memory, or (ii) via tasks, where a programmer splits the code into small idempotent sections. Results so far suggest that a task-based approach performs better than checkpointing. However, existing tasks-based approaches do not adapt to changing energy levels and storage. This makes the code inefficient (when executed on a larger energy buffer than intended) or can potentially completely disable the program (in the opposite case). Therefore, we present Coala: a task-based execution runtime, enabling code portability by means of a new concept of task coalescing. If a code is written for a very small energy buffer, Coala is able to dynamically coalesce these tasks and constructs a bigger (virtual) task to improve execution times up to 4.5 times compared to static task code.

1 Introduction

Advances in processor efficiency along with the development of energy-harvesting systems has created a new category of devices that require neither a battery nor a tethered power supply [35, 48, 57]. These devices operate using ambient energy, such as radio frequency transmissions [15], light [40, 41], and vibration [16]. Incorporating compute, storage, sensing, and communication hardware [45, 69], such devices are a promising technology for use in the Internet of Things [30], in-body [43] and on-body [4] medical systems, and energy-harvesting nano-satellites [39].

Energy-harvesting devices create unique challenges because they operate *intermittently* when energy is available [24, 35]. An energy-harvesting device buffers energy in a small storage, e.g. a capacitor [17], [19] and when a threshold amount of energy accumulates, begins operating. Harvestable energy sources are low-power compared to a platform's operating level. A device operates briefly and when buffered energy is depleted, shuts down and recharges to operate again later. As an example, recharge time may be tens of seconds in radio frequency powered medical device [43, Fig. 3c]. Moreover, charge and discharge times vary by device (due to different capacitor sizes) and some may fail ≈10 to ≈100 times per second [60], [50], [37].

Data Consistency in Intermittent Computing. Software in an energy-harvesting system operates in the *intermittent execution model* [35, 36], with execution interrupted by failure periods. When power fails, a device loses volatile state, i.e., registers, stack, SRAM, and retains its non-volatile state, i.e., FRAM, Flash. While capturing periodic checkpoints [28, 50] and sleep scheduling [2, 3, 7] help preserve execution progress, failures can leave volatile and non-volatile state inconsistent, leading to unrecoverable failures [9, 36].

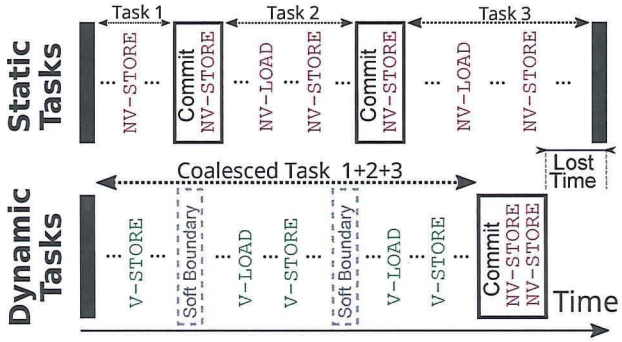


Figure 1. An execution of three statically-defined tasks without (top) and with (bottom) dynamic coalescing. Task coalescing at runtime reduces time and energy overhead in task-based intermittent programming models by performing fewer commits and buffering updates to non-volatile (NV) variables in volatile (V) memory.

There are two main approaches for dealing with data inconsistency for intermittently-powered devices: (i) *programming and execution models* [10, 36, 38, 67] and (ii) *architectures* [24, 37, 42, 63]. New architectures require hardware changes and are inapplicable to today's systems [24, 37], therefore new programming models and compilers are more suitable (as of now). However, they have their own limitations that need to be addressed—the core one being the rigidity of the *static* decomposition into tasks.

Task Decomposition of Intermittent Programs. Recently proposed *task-based* programming and execution models [10, 38] advocate for static decomposition of a program (by a programmer) into a collection of tasks. Tasks can include arbitrary computation and, upon completion, are guaranteed to have executed *atomically*, despite arbitrarily-timed power failures. The programmer explicitly expresses task-to-task control flow. The execution in Figure 1 (top) illustrates how the decomposition into tasks affects the time the application takes to complete. Each task transition introduces the overhead of tracking and atomically committing the modifications to non-volatile memory, to maintain consistency of program state [10, 38]. Naively minimizing the number of task transitions, (i.e. the task count), *at compile time* risks

Submitted to PLDI'18, June 18–22, 2018, Philadelphia, PA, USA 2018.

creating a task that requires more energy than the device can buffer in its energy storage. Such a large task would fail to complete unless the environment provides sufficient incoming energy to supplement the energy in the capacitor. To eliminate this risk, the programmer (or the compiler) must decompose the program conservatively into many small tasks, at the cost of *higher task transition overhead.*

One approach to decrease the task count, without assuming a minimum level of incoming power always available, is to predict task energy consumption and choose a decomposition specific to the device capacitor size. However, statically predicting energy of arbitrary input-dependent code with peripheral access is a problem without a general solution. Furthermore, a static decomposition approach prevents portability across devices with different storage capacitors.

Task Coalescing. To address this challenge we propose a *dynamic* task decomposition, that accepts any static decomposition and opportunistically coalesces tasks at runtime when there is sufficient energy. With *task coalescing* atomic tasks are merged into a single (larger) task while preserving atomicity of the merged task. If power fails while a coalesced task is running, execution will restart at the first task in the group. Figure 1 (bottom) illustrates how dynamic coalescing changes the earlier execution and highlights the change in memory model necessary to not break atomicity.

Unfortunately, naively merging two atomic tasks might produce a non-atomic task, that may leave non-volatile memory inconsistent after a partial execution. A merge breaks atomicity when atomic merged tasks form a *write-after-read (WAR)* dependency—for instance if two tasks `{x++};` `{vector[x]=v;}` are merged and if the power failure occurs after `x++`, the value `x` will be increased twice when the merged task is restarted, that leads to an inconsistency. Although WAR dependencies introduced by the *static* task division can be eliminated at compile time [38], those introduced *dynamically* at run-time cannot. To maintain memory consistency in merged tasks, we propose a mechanism for *memory virtualization* that buffers the updates to non-volatile variables in volatile memory before committing them at the dynamic task boundary.

Task coalescing removes the burden from the programmer, because it accepts any task decomposition and improves it dynamically. To further reduce the programmer effort, we propose a compiler pass for *automatic decomposition* of programs into (small) *atomic* tasks. The compiler identifies non-volatile variables shared across tasks as tasks are created, and instruments reads and writes of those variables, using memory virtualization, to keep the data consistent in the presence of power loss. Despite being limited to a subset of the C language, the automatic task decomposition allowed us to port several applications to an intermittent platform with a moderate effort.

Contributions. We develop Coala[1]: the first task coalescing system for task-based programming models on intermittently-powered devices. The capabilities required for effective coalescing mark our contributions:

- Two adaptive coalescing strategies for selecting boundaries to coalesce based on past execution behaviour;
- Software memory virtualisation for transiently-powered devices that preserves atomicity of coalesced tasks;
- Compiler pass for automatic decomposition into small atomic tasks for adaptive coalescing at runtime.

We evaluated Coala end-to-end on a real energy-harvesting device [55] running existing benchmarks [10] and new programs. A comparison to a state-of-the-art task-based system [10] showed that Coala recovers performance on conservative task decompositions missed by systems with static decomposition.

2 Intermittent Computing: Background

2.1 Energy Harvesting Systems

Energy harvesting devices operate using energy extracted from sources such as radio frequency transmissions and solar energy. These devices elide tethered power or a battery, instead collecting energy into a capacitor, operating when sufficient energy accumulates, and upon depleting the energy, turning off and recharging. Many platforms enable intermittent, battery-less energy harvesting-based computation. For instance, computational RFIDs—open-source TI MSP430-based [62] WISP [45] (with its variants such as WISPCam [44], NFC-WISP [70] or NeuralWISP [25]), Moo [69], and commercial ones such as [13]. Other intermittently-powered platforms include ambient backscatter tag [34, 46] or battery-less phone [59].

Hardware Assumptions. Coala is designed for the demands of existing and future energy-harvesting platforms based around general purpose, commodity computing components [55, 65]. We assume a device with a memory system that has fast, byte-addressable volatile and non-volatile memory; in particular, our target platform, WISP [55], is equipped with a mixture of SRAM and FRAM. Our implementation leverages hardware support for fast, bulk-copying between memories via DMA [65]. We do not require a particular non-volatile memory technology, nor do we require architectural additions commodity processors [28, 37, 58, 67]. Coala supports I/O behavior similar to [10, 38], allowing safe, synchronous I/O and unsafe, asynchronous I/O.

2.2 Intermittent Execution

Software running on an energy-harvesting device executes *intermittently* because buffered energy is only available *sometimes.* An intermittent execution is composed of operating periods interspersed with power failures [10, 36, 38, 67]. The frequency of failures depends on the size of the device's

[1] To be released at http://anonymized.link

```
1 NV int byte, nBytes, Rmnd, Wdth, char msg[], crcTbl[];
2 for (byte = 0; byte < nBytes; ++byte){
3    checkpoint(); //Inserted checkpoint routine
4    data = functReflectData(msg[byte]) ^ (Rmnd >> (Wdth - 8));
5    Rmnd = crcTbl[data] ^ (Rmnd << 8);}
```

(a) Simplified C code snippet of a CRC calculation from [23]: per-byte message division by a polynomial; NV denotes non-volatile variable declaration.

```
1  checkpoint(); //Correct execution iteration 1
2  data = functReflectData(msg[byte]) ^ (Rmnd >> (Wdth - 8));
3  Rmnd = crcTbl[data] ^ (Rmnd << 8);
4  checkpoint(); //Capture checkpoint before iteration 2
5  data = functReflectData(msg[byte]) ^ (Rmnd >> (Wdth - 8));
6  Rmnd = crcTbl[data] ^ (Rmnd << 8);
7  ϟ  ϟ  ϟ    //Power fails after Rmnd written, before checkpoint()
8  ↻checkpoint(); //Restart from checkpoint; Rmnd remains updated
9  data = functReflectData(msg[byte]) ^ (Rmnd >> (Wdth - 8));
10 ×ERROR: data updated using incorrect Rmnd value
11 ...
```

(b) Execution steps of the loop body in the snippet above: non-volatile checkpointing did not guarantee data consistency as data has been manipulated (line 9) with stale reminder (line 3)

Figure 2. Code example demonstrating effect of write after read on volatile memory checkpointing.

energy storage buffer: a larger buffer allows longer operating periods. Energy-harvesters provide input power orders of magnitude less than operating power, making recharging negligible during operation.

Intermittent execution is different from continuous execution. A power failure clears volatile state (registers, stack, and globals) and non-volatile memory (e.g., FRAM) persists. At a 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 [28, 50], sometimes using hardware support [2, 3, 6, 42, 50].

Checkpoints of volatile state preserve intermittent progress, but do not ensure data consistency [10, 36, 67]. Data may become inconsistent if an attempt to execute some computation *writes* to a non-volatile variable, then power fails, then a second attempt to re-execute the same computation incorrectly *reads* the value written in the first attempt, rather than the variable's original value. The situation occurs when code includes a WAR dependence between operations that manipulate non-volatile variables [36, 38, 67].

Figure 2 illustrates how state can become inconsistent in an intermittent execution using the cyclic redundancy check (CRC) code from MIBench2 [23]. The code computes the CRC for an `nBytes` byte message `msg` with remainder `Rmnd`.

Despite checkpointing at line 3 in Fig. 2a, the code may compute `data` incorrectly because of the read of `Rmnd` on line 4 and the write of `Rmnd` on line 5. Figure 2b shows an intermittent execution. The second iteration writes `Rmnd`, but power fails before the checkpoint. After restarting, line 9 reads the updated value of `Rmnd`, producing an incorrect `data` value.

Model	Data Copied to/from NVRAM
Mementos [50]	Reg. + Stack
DINO [36]	Reg. + WAR variables
Chain [10]	PC + Channel data
Alpaca [38]	PC + WAR variables
Ratchet [67], Clank [24]	Reg. (requires NV main memory)

Table 1. Memory consistency enforcement overheads of intermittent execution models; *Reg.*: the entire register file, *PC*: program counter, *Channel data*: variables explicitly task-shared by the programmer, *WAR variables*: variables involved in WAR dependences (*NV*: non-volatile).

Checkpoint-based systems risk violating memory consistency [36]. To maintain consistency, prior systems [36, 67] *version* a subset of non-volatile data with the checkpoint. Task-based systems [10, 38], which we focus on, ensure consistency using programmer, compiler, and runtime support.

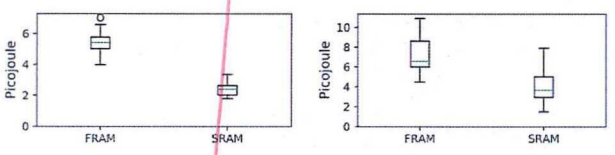
2.3 Task-based Intermittent Programming

Task-based execution models [10, 36, 38] ask the programmer to decompose their program into tasks. A task is a function with no caller containing arbitrary computation, sensing, and communication. A programmer describes task control-flow as a *task graph*. Task flow happens at programmer demarcated transition points that may be conditional on program values. Task-based programming abstractions guarantee that tasks execute *atomically*, regardless of power failures. Task-based runtime systems ensure task-atomic semantics by ensuring that repeated task executions are idempotent. The key to idempotence is ensuring that non-volatile updates made by an interrupted task are never visible to a future task execution.

There are several run time strategies to ensuring task idempotence. One approach [38] is to identify non-volatile data involved in WAR dependences in a task (like DINO [36] and Ratchet [67] did for checkpoints), execute the task using private copies of those data, and commit the private copies on task completion. Another way is to statically create multiple versions of non-volatile data shared by tasks and ensure that no task reads and writes the same version [10]. Regardless of the strategy, task-based systems execute statically-defined tasks atomically, completing in one or more attempt.

2.3.1 Costs of Task-based Models

Memory Consistency Enforcement Overhead. Intermittent execution models incur overheads to checkpoint data [28, 36, 50, 67], manage channels [10], or privatize and commit data [38]. Table 1 compares overheads for recent intermittent execution models (cf. [38, Sec. 2.4].) The mechanism responsible for overhead in each model varies, but the bulk of overhead in all models is a manipulating non-volatile memory. Checkpointing moves data to and from non-volatile memory, channel accesses manipulate non-volatile



(a) Cost of *write* operation **(b)** Cost of *read* operation

Figure 3. Cost of accessing volatile (SRAM)/non-volatile (FRAM) memory during write/read operation.

data [10], privatization copies from and commits to non-volatile memory [38], and idempotence solutions [67] use *only* non-volatile memory.

To understand these non-volatile memory overheads, for the MSP430FR5969 [65] MCU we measured the energy consumption of volatile (SRAM) and non-volatile (FRAM) memory operations. We used EDB [9] to monitor voltage drop on a capacitor powering the device across 1600 read and write accesses to each memory type, randomizing to avoid caching. Figure 3 summarizes the data, showing that a non-volatile memory access consumes around 1.5 times the energy of a volatile access in this device. In an intermittent device, energy cost corresponds to decreased run time between power failures. The cost of non-volatile memory motivates Coala, which virtualizes memory to primarily use SRAM.

Fixed-size Tasks are Inflexible and Inefficient. If a programmer writes a task that consumes *more* energy than the device can buffer, the task will never complete using buffered energy *preventing forward progress* by repeatedly re-executing. If a programmer writes tasks that consume far *less* energy than a device can buffer, the system may operate *inefficiently*. When a task and its successor both complete, the two tasks were interleaved by a task boundary that preserves progress and state (i.e., checkpoint or commit). However, absent a power failure, the work to preserve state was *unnecessary*, incurring overhead on the tasks' executions.

Avoiding excessively costly, non-terminating tasks and short, high-overhead tasks is a programming challenge, given fixed hardware with a fixed energy buffering capacity. Buffer sizes from prior work vary widely from 20 μF [52] to 0.1 F [69]. Sizing tasks complicates *porting* code from one device to another. An excessively costly task on a device with a small energy buffer may be a relatively short, task on a device with a larger buffer. The key problem is that using existing systems, the programmer statically sizes tasks for a fixed energy buffer. Coala's *task coalescing*, described in Section 4, is motivated by these observations.

3 Coala: System Overview

Coala is a new programming and execution model for intermittent computing on energy-harvesting devices. Coala addresses the challenges outlined in Section 2 to make task-based intermittent programs *accessible*, *efficient* and *flexible*. Coala accomplishes this goal with a constellation of a new

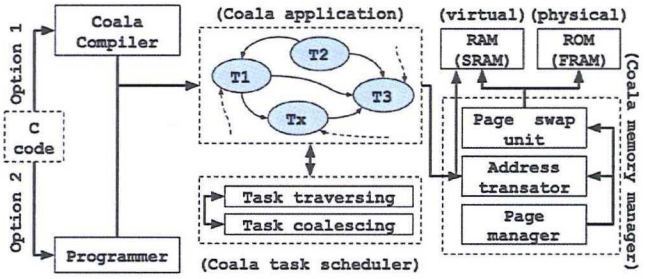


Figure 4. Coala top-level view.

Keyword	Description
`task foo() {...}`	Task definition
`origin task foo() {...}`	Origin task definition
`next_task(t)`	Transition to task t
`u = RVAR(v)`	Read protected variable v into u
`WVAR(v, u)`	Write u to protected variable v

Table 2. Coala's programming interface.

programming model, compiler, and run time software system support, refer to Figure 4.

Coala Programming Model. To use Coala, a programmer first writes plain, imperative C code. The programmer then has an option to *manually* or *automatically* translate the program into code for Coala's task-based programming model. To manually translate, the programmer decomposes the program into tasks, and annotates memory accesses that manipulate data shared by multiple tasks. To automatically translate, the programmer simply uses Coala's compiler (described in Section 6), which decomposes a program into tasks and annotates accesses to task-shared data. After translating to task-based code and compiling, the programmer has an executable Coala binary.

Coala's programming interface, listed in Table 2, consists of constructs for defining tasks [10, 38] and accessors for protected variables shared between tasks. These annotations are added either by the programmer (in the manual translation mode) or the compiler (in automatic translation mode). As in prior task-based intermittent programming models [10, 38], tasks are functions that are not allowed to have any direct callers but receive control by explicit `next_task` statements. In Coala, a task transition does not atomically commit the memory operations of the current task—commit is deferred until the boundary of the coalesced task is reached. The programmer identifies each task by adding the `task` keyword to the function's declaration and designates one task as the *origin* task at which execution begins. After a power failure, Coala restarts from the last (partially) executed coalesced task. A Coala program manipulates *protected* state—accessed by more than one task—using the *read variable* (RVAR) and *write variable* (WVAR) operations. These operations convey to Coala's memory manager which data must be buffered in volatile memory and committed.

swap Sec 3 and 4

Coala Memory Manager. At runtime Coala's *virtualizing memory manager* (Section 5) protects changes to non-volatile data against becoming inconsistent after a power failure. Relying on annotations of accesses to protected variables, Coala's memory manager *pages* these data into volatile memory from non-volatile memory as they are needed. Tasks cannot modify protected variables directly in non-volatile memory. If the working set exceeds the size of volatile memory, pages are swapped out to non-volatile memory without committing the changes, leaving original locations of protected variables unchanged. When a coalesced task completes, Coala atomically commits modified pages to their original locations in non-volatile memory.

Coala Task Scheduler. The scheduler *coalesces* statically-defined tasks into dynamic tasks sized to match the available energy. By default, tasks run in sequence and each task commits as it completes, potentially incurring unnecessary overhead for commits between consecutive, non-failing tasks. Coala dynamically coalesces consecutive tasks by deferring the commit of an earlier task until the completion of a later task. Coala's memory management mechanism is essential for coalescing, because commits are deferred by keeping dirty pages in volatile memory across coalesced boundaries. After a coalesced sequence of tasks completes, Coala commits pages updated by all of the tasks to non-volatile memory, reducing the commit overhead through batching.

4 Task Coalescing

When incoming energy is sufficient to run multiple consecutive tasks, committing state after each task is wasteful. Coala mitigates this waste by deferring the expensive commits and performing them in a batch. In the resulting execution, several tasks are *merged* into a single virtual task—a process we call *coalescing*. When N tasks are coalesced, $N-1$ boundaries are crossed and some number of pages are dirtied. The first $N-2$ executions of `next_task` statement skip committing dirty pages to non-volatile memory, leaving the work of committing pages modified by any of the N tasks to the statement $N-1$. Coalescing is likely to reduce the total number of non-volatile memory accesses, because when coalesced tasks access the same pages of data, only the last version of the page needs to be committed.

Trade-off between Speed and Wasted Effort. Coala can coalesce an arbitrary number of consecutive tasks. However, as more tasks coalesce, their collective commit overhead amortizes better, but the risk of wasting work also increases. If power fails during a long sequence of coalesced tasks, execution will restart from the last commit, i.e. the first task in the sequence, losing the progress made by any of the coalesced tasks. The challenge to coalescing tasks is determining how many tasks to coalesce before committing.

Task Coalescing Strategies. The likelihood that a task group will complete depends on the total amount of energy consumed by all tasks in the group, i.e. the size of the dynamic task. A good coalescing strategy must moderate the risk of wasted work and capitalize on benefits of deferred commits by adapting the target size of the dynamic task to the energy available at runtime. The general structure of our strategies is to increase the target task size by some positive delta C_{up} after the dynamic coalesced task completes, and decrease it by some negative delta C_{down} upon a reboot after a power failure. The choice of C_{up} and C_{down}, including their units, defines the strategy and its performance. For example, a naive strategy that fixes C_{up} and C_{down} to "one static task", will be very slow to react to changes in incoming energy, since its dynamic task size can only change by the size of one (small) static task per charge-discharge cycle. Working from these observations, we propose and evaluate two coalescing strategies for Coala: **History-aware Coalescing (HC)** and **Task-size- and History-aware Coalescing (TSHC)**.

4.1 HC: History-aware Coalescing

stripped

Assuming tasks of similar energy costs, an accessible predictor of a suitable dynamic task size is the number of successfully completed static tasks in the recent past. The HC policy keeps track of the number of static task boundaries crossed since the last boot to set a *target* dynamic task size. This target is expressed in units of "one static task" and persisted in non-volatile memory. When target number of boundaries is crossed, the policy stops coalescing, thereby setting the size of the dynamic task at the target.

The target is conservatively initialized to zero before the origin task runs for the first time, i.e. $T_0 \leftarrow 0$. After each successful end of a dynamic task, the policy adjusts the target upwards, to explore the possibility for larger dynamic tasks. The optimistic upward adjustment of the target is counteracted by the downward adjustment that takes place after a dynamic task fails to complete after energy is exhausted, i.e. on each reboot. The adjustments are geometric by a parametrizable factor α, i.e. $T_{i+1} \leftarrow \lceil \alpha T_i \rceil$, where $\alpha > 1$ for upward and $\alpha < 1$ for downward adjustment. In our experiments, we empirically tuned the controller parameters to $\alpha_{\text{up}} = 1.5$ and $\alpha_{\text{down}} = 0.5$. The absolute values of the parameters provide a reaction time that is large enough to filter out outliers, due to short peaks or lulls in energy supply or due to disparate task energy cost. The asymmetry reflects conservatism: fast reaction to a decreased energy availability – which prioritizes reducing wasted executions over attempts to extract maximum gain – and slower reaction to increased energy – which cautiously pushes for a larger gain.

4.2 TSHC: Task-size- and History-aware Coalescing

stripped

The HC policy used the number of tasks in the group as a rough proxy for the energy cost of the task group. With some up-front investment of developer effort and modest time and space overhead, a more precise proxy for task size can be obtained by profiling the tasks at design time. The TSHC