EarlyBird: Energy belongs to those who wake up early

Hugo Reymond, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Isabelle Puaut, Erven Rohou

EARLYBIRD: Energy belongs to those who wake up early

Hugo Reymond*, Jean-Luc Béchennec†, Mikaël Briday†, Sébastien Faucou†, Isabelle Puaut* and Erven Rohou*

*Univ Rennes, Inria, CNRS, IRISA, France - name.surname@irisa.fr
†Nantes Université, École Centrale Nantes, CNRS, LS2N, F-44000 Nantes, France - name.surname@ls2n.fr

***Abstract*—By relying on ambient energy, battery-less devices significantly increase the autonomy of IoT devices, enabling maintenance-free operation in remote locations. However, due to the scarcity of ambient energy, these devices rely on capacitors to buffer energy, and alternate between power-off phases where the device is harvesting energy and computation bursts. In most existing techniques, the device resumes execution only when the capacitor is full. However, we argue that doing so is sub-optimal. Instead, we advocate that waking-up the device sooner may yield better performance since the microcontroller consumes less power when operating at lower voltage. To this extent, we introduce EARLYBIRD, a technique that automatically computes a fine-tuned wake-up voltage for each resume point. EARLYBIRD leverages static analysis to determine how much energy is needed before resuming from a given program location, and provides a runtime library to enforce the early wake-up strategy. We evaluated how EARLYBIRD improves existing checkpointing techniques and results show an increase in the number of benchmarks executed per minute of up to 5.65×.**

***Index Terms*—Intermittent Computing, Battery-less devices, Embedded and IoT devices, Implicit Path Enumeration Technique (IPET)**

I. INTRODUCTION

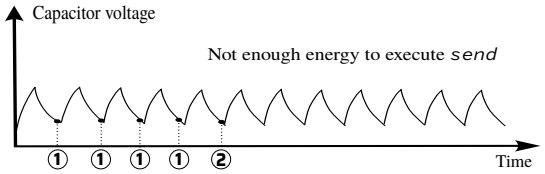
Thanks to research in the energy-harvesting field, it is now possible to power IoT devices directly from ambient energy (solar, piezoelectric, radio frequency – RF –…) This eliminates the need for batteries, and opens opportunities for a "battery-less" future, where IoT devices only rely on a capacitor to store the harvested energy. Battery-less devices can operate without the need for frequent battery replacements, allowing them to be used in remote locations where maintenance is impossible (in space, into building foundations…)

Unfortunately, due to the scarcity and variability of the energy available in the environment, battery-less devices experience frequent power failures. Such power failures result in a device reset, which erase its volatile state (volatile memory, CPU registers…), effectively wiping any progress achieved so far. Therefore, one key challenge with battery-less devices is to preserve the progress of the program across power failures, a property known as *forward progress*.

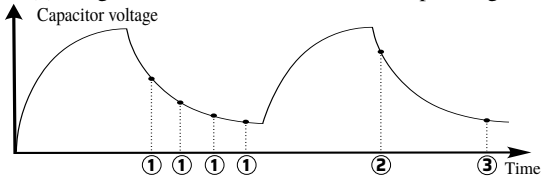
One technique used to enforce forward progress across power failures is *static checkpointing*. It consists in inserting calls to checkpointing routines in the program code, as depicted in Figure 1. At runtime, when the checkpointing routine is called, it saves the program state and the CPU registers into non-volatile memory. Then, in the event of a power failure, the device can load the previous checkpoint to resume program execution. Of course, the device must wait until a certain amount of energy has been harvested before resuming execution.

```
int sum = 0;
for(int i = 0; i<4; i++){
    checkpoint(); ①
    sum += array[i] ⚡
}
checkpoint(); ②
send(sum); ⚡⚡⚡
checkpoint(); ③
```

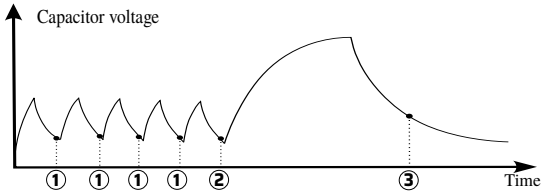
Program instrumented for intermittent execution. The program contains three checkpoints, labeled ①, ② and ③. The function `send` must execute atomically – its entire execution must terminate before a power failure – and requires more energy than the sum calculation.



(a) Program execution with low wake-up voltage.



(b) Program execution with high wake-up voltage.



(c) Program execution with EARLYBIRD. The wake-up voltage when resuming from checkpoint location ① is low, while it is high for ② as the function `send` is energy-intensive

Fig. 1: Example program instrumented for intermittent execution, and possible executions

To determine how much energy has been harvested, battery-less devices can monitor the voltage across the capacitor, as it is directly linked to its state of charge. Battery-less devices programmers/designers can then decide on the *wake-up voltage* of the program, that is the voltage below which the device should be off, harvesting energy. The choice of a wake-up voltage is complex and impacts several parameters, as described below.

P_1 – *Energy consumption*: microcontrollers usually accept a range of input voltages (*e.g.*, from 1.8 V to 3.6 V), but their energy consumption augments as the voltage increases. Thus, diminishing the average input voltage of the microcontroller reduces the energy consumption of the device, which in turn reduces the amount of time spent harvesting. Consequently, a low wake-up voltage implies that the energy available is used more efficiently. This is illustrated on Figure 1, where the execution reaches checkpoint ② faster with a low wake-up voltage (Figure 1a) than a higher one (Figure 1b).

P_2 – *Forward progress*: the choice of a wake-up voltage impacts the program's ability to enforce forward progress across power failures. Setting the wake-up voltage too low may result in scenarios where the program's state cannot be saved before a power failure – thereby preventing any progress. In Figure 1a, the low wake-up voltage fails to execute the entire program as the energy consumed by the `send` function exceeds the available energy upon restart. It is thus crucial for the wake-up voltage to be adapted to the load to execute.

P_3 – *Progress regularity*: it is sometimes important that programs execute as regularly as possible, but when the device is turned off due to a power failure, it cannot react to environmental events. In order to enforce progress regularity, an intermittent system must have short harvesting phases. The choice of a wake-up voltage directly impacts the duration of the harvesting phases, it is thus beneficial to select low wake-up voltage to obtain short harvesting phases.

Given the complexity of choosing a wake-up voltage, leaving this task to a developer might result in bad performance, or even result in the program not making any progress. Moreover, exploring different wake-up voltages and their impact on program execution is a time-consuming process, that must be performed every time the program changes. Thus, existing techniques use a single wake-up voltage for the entire program, overlooking the potential benefits of adapting the wake-up voltage to the workload to execute. We argue that such adaptability is crucial for enhancing program performance.

To this extent, we introduce EARLYBIRD. EARLYBIRD is, to the best of our knowledge, the first technique that addresses the problem of wake-up voltage selection in intermittent systems that uses static checkpointing. EARLYBIRD automatically selects a wake-up voltage for each checkpoint location in the program. The voltage is adapted to the portion of code to be executed when resuming after a power failure. This allows EARLYBIRD to take advantage of the energy-efficiency of low wake-up voltages when the code section is short, while still ensuring forward progress for energy-intensive functions with a higher voltage. In our example, EARLYBIRD would select a low wake-up voltage for checkpoint ①, and a high one for ②. This would result in the execution illustrated Figure 1c.

More precisely, EARLYBIRD leverages static analysis of the program binary to automatically compute the energy required to resume execution from a given location in the program. It then selects the minimum (yet safe) wake-up voltage accordingly. This analysis also provides valuable feedback for the developer as it indicates if the chosen capacitor is large enough to execute the program, or if additional checkpoints are required to divide the program into smaller segments. EARLYBIRD is agnostic about the strategy used for the static selection of checkpoint locations.

At runtime, EARLYBIRD enforces the selected wake-up voltages thanks to a custom checkpoint and restore routine. To avoid any additional burden for the developer, those routines come in a library, as a drop-in replacement for existing checkpointing libraries such as the MEMENTOS library [1].

This paper makes the following contributions:

- We introduce EARLYBIRD, the first technique to address the problem of wake-up voltage selection in intermittent systems based on static-checkpointing. In contrast to manually selected wake-up voltage, EARLYBIRD adapts the wake-up voltage for each checkpoint location. It does so automatically, relieving the developer from this burden.
- We conduct experiments to analyze how the wake-up voltage impacts program execution. We then evaluate EARLYBIRD performance for basic checkpoint placement strategies, and its ability to improve performance of existing state-of-the-art techniques. All experiments are conducted on a real board using a msp430fr5969 microcontroller, using RF energy harvesting.

The remainder of this paper is organized as follows. First, we present in Section II a preliminary study on the impact of the wake-up voltage on program execution. Then, we discuss related works in Section III. Section IV introduces EARLYBIRD. Then, we compare EARLYBIRD to existing wake-up voltage selection strategies and present those results in Section V. Finally, we conclude in Section VI.

II. PRELIMINARY STUDY: IMPACT OF THE WAKE-UP VOLTAGE ON INTERMITTENT PROGRAMS

This preliminary study investigates how the selection of a wake-up voltage affects energy consumption (P_1), forward progress (P_2), and regularity of progress (P_3).

A. Impact of the Operating Voltage on Energy Consumption

As expressed in P_1, the energy consumption of a microcontroller depends on its input voltage, higher voltages resulting in greater energy consumption. In this section, we verify this statement, and quantify the energy savings achievable with a lower input voltage. To accomplish this, we measured the instantaneous power consumption of a TI msp430fr5969 microcontroller over its supply voltage range (from 1.8 V to 3.6 V) with a N6705A power analyzer [2]. The msp430fr5969 ("msp430" for short) is directly powered from the power

analyzer, and executes an infinite loop of NOP instructions. The results are displayed on Figure 2.

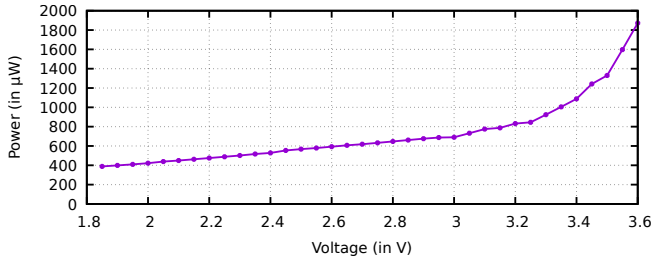


Fig. 2: Evolution of the power consumption over the msp430fr5969 operating voltage range at 1 MHz

We observe that power consumption increases roughly linearly with the operating voltage up to 3.3 V, after which it increases faster. Notably, there is a more than fourfold increase in power consumption between the lowest and highest recommended voltages of the msp430 microcontroller. If we consider only the linear portion, the power consumption experiences a twofold increase between 1.8 V and 3.3 V.

Based on these results, it is evident that substantial energy savings could be obtained by operating the microcontroller at a lower voltage. This motivates us to explore in greater detail the impact of wake-up voltage on program execution.

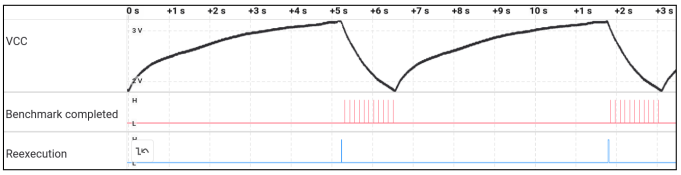
B. Impact of the Wake-Up Voltage on Intermittent Programs

The wake-up voltage dictates the quantity of energy available for an execution cycle as well as the amount of time that will be spent off, recovering this energy. Hence, the choice of a wake-up voltage has significant effects on the program behavior, notably its ability to ensure forward progress (P_2) and its progress regularity (P_3). In this section, we study the impact of two commonly used wake-up voltages strategies on the program behavior.

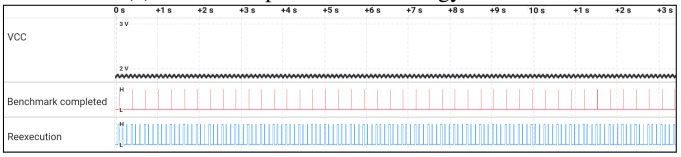
The first strategy, named V-HIGH, monitors the capacitor voltage and resumes execution when the capacitor is almost full (*e.g.*, 3.3 V). Several works rely on this hypothesis to guarantee forward progress [3–5].

The V-LOW strategy, on the other hand, relies on the voltage hysteresis between the power-on voltage (1.88 V) and power-off voltage (1.8 V) of the msp430. This hysteresis, implemented in hardware, aims at preventing oscillations in the chip power supply. In intermittent execution, it becomes a simple way to ensure some energy has been harvested before restarting after a power failure. This enforces a wake-up voltage of 1.88 V. As it does not require any monitoring of the capacitor voltage on the software-side, the strategy is the easiest to set up, and is used in several works such as Van Der Woude and Hicks, or Yıldırım *et al.*, [6, 7].

We first explore qualitatively how the V-HIGH and V-LOW wake-up strategies impact the program execution by looking at execution traces. Then, we analyze how these observations translate into quantitative results. The experimental setup used in the study is the same as the one presented in Section V-A.



(a) Execution profile with strategy V-HIGH



(b) Execution profile with strategy V-LOW

Fig. 3: Execution profile for strategies V-HIGH and V-LOW. The *VCC* curve represents the voltage across the capacitor; the *Benchmark Completed* signal is raised when the benchmark was successfully terminated; the *Reexecution* signal is raised when the platform starts re-executing code.

1) Influence of Wake-up Voltage on Execution Profiles: To evaluate the effect of the wake-up voltage on the execution profile, we ran the *aes* benchmark iteratively using both V-HIGH and V-LOW strategies. The benchmark is taken from the Mibench2 benchmark suite and is instrumented with call to checkpointing routines inserted before function returns. Using a signal analyzer, we monitored the evolution of the voltage across the capacitor, as well as a signal raised every time the benchmark was successfully executed.

Before conducting any quantitative analysis, the observation of Figure 3 gives us an overview of the impact of wake-up strategies on the program execution. What immediately strikes is the difference in scale of the duration of intermittency cycles between the two strategies: execution phases for the V-HIGH strategy last around 1 second, while they only last 50 ms with V-LOW. As a consequence, we observe that benchmark completions are evenly distributed in time with the V-LOW strategy while they are clustered with the V-HIGH strategy. Moreover, the long harvesting phases in V-HIGH reduce the system's progress regularity (P_3). Nonetheless, as the V-LOW strategy experiences more frequent power failures, it spends a significant amount of time re-executing portions of code whose progress could not be saved.

2) Influence on Program Performance: In this section, we quantitatively compare the V-HIGH and V-LOW strategies. First of all, we assess the ability of the strategy to enforce forward progress, visible in Table I. While the V-HIGH strategy consistently ensures program progress across all three benchmarks, the V-LOW strategy fails to do so with *aes* and *rc4* when checkpoint routines are far apart.

To investigate the amount of useful work achieved with each strategy, we monitored how many time the *aes* benchmark could be executed in two minutes. We observed a significant throughput difference, with an average of 190 executions completed for V-HIGH and 295 for V-LOW.

To explain this difference, we dive into the activity of the de-

TABLE I: Ability of the V-LOW and V-HIGH strategies to ensure forward progress. For more details, checkpoint placement methods (*LL/FR*) and (*S-10%/S-100%*) are described in Section V.

	Frequent checkpointing (*LL/FR*)			Occasional checkpointing (*S-10%/S-100%*)		
	aes	*crc*	*rc4*	*aes*	*crc*	*rc4*
V-LOW	✓	✓	✓	✗	✓	✗
V-HIGH	✓	✓	✓	✓	✓	✓

vice during the experiment. Figure 4 displays, for the V-HIGH and V-LOW strategies, the proportion of time spent executing code and saving checkpoint (Execute), sleeping waiting for the capacitor to replenish (Sleep), restarting and loading previous checkpoint (Restore) and reexecuting portions of code whose progress could not be saved (Reexecute).

First of all, the V-LOW strategy spends about twice as much time executing, which explains the throughput increase. We can also see that, because of its higher power consumption, the V-HIGH strategy spend around 76.7 % of its time harvesting energy, whereas the V-LOW strategy only 44.9 %. However, as the V-LOW strategy encounters frequent power failures, it spends a significant portion of time restarting the msp430 and restoring the previous checkpoint. Reexecution time is marginal here, as the program state is saved frequently, but this is not always the case, as we will see in the experimental evaluation of EARLYBIRD.

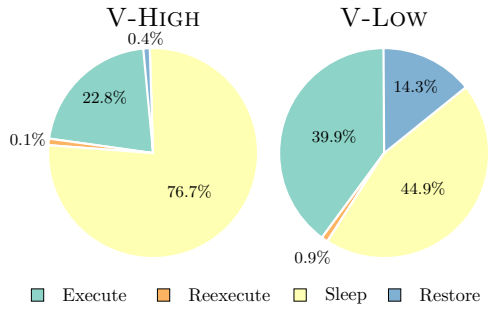


Fig. 4: Device activity during the experiment executing *aes*, with the V-HIGH and V-LOW wake-up strategies.

Overall, this study confirms the wake-up voltage is a crucial parameter for battery-less devices, as it deeply impacts the program behavior and its performance.

III. RELATED WORKS

A. Intermittent Computing Strategies

Running a program on battery-less devices requires mechanisms to persist the program state across power failures. While some research work has focused on designing specialized hardware for this task [8, 9], commercial off-the-shelf microcontrollers currently lack support for intermittent computing. Consequently, significant effort has been directed towards software-based solutions. We distinguish three main families of techniques: *dynamic checkpointing*, *task-based* and *static checkpointing*.

Dynamic checkpointing is the most user-friendly technique, as it does not require any modification of the program to be used. Instead, it relies on voltage- or time-based triggers to halt program execution and start the checkpointing process. However, as the developer does not specify where in the code the checkpoint should be performed, it becomes impossible to define portions of code that must execute atomically, known as *atomic sections*. Atomic sections are crucial for peripheral handling, as one cannot simply resume peripheral operation after a power failure.

In contrast, task-based techniques require the developer to organize their code as atomic tasks, and only save checkpoints on transition between tasks. This approach often requires developers to explicitly define the data shared between tasks, resulting in reduced checkpoint sizes and low overhead. However, the process of separating a program into tasks can be burdensome for developers.

Finally, static checkpointing offers a balance between the ease of programming of dynamic checkpointing and the efficiency of task-based checkpointing. It consists in inserting checkpointing routines in the program, as shown in the code Figure 1. It supports atomic sections and can benefit from checkpoint size optimization thanks to compile-time analyses. Moreover, in contrast to task-based checkpointing, it allows to define checkpoint locations at a fine grain (inside a loop, a function). Table II summarizes the pros and cons of each class of techniques. In this paper, we focus on static checkpointing techniques.

TABLE II: Pros and cons of families of technique to support intermittent execution

Criterion	Dynamic	Task-based	Static checkpointing
Support atomic sections	No	Yes	Yes
Ease of utilization	+++	-	+
Checkpointing overhead	High	Very Low	Low
Granularity	Instruction	Function	Instruction
Examples	[10–13]	[7, 14–18]	[1, 5, 19, 20]

B. Existing Works on Application-Tailored Wake-Up Voltage

Works that investigate application-tailored wake-up voltage primarily revolves around task-based checkpointing techniques. Most scheduling techniques indirectly implement such mechanism, as they make sure that a task is scheduled only when enough energy is available in the capacitor [21–23].

An interesting approach to adapt the energy available at wake-up has been proposed by Colin *et al.*, [24]. Instead of adapting the wake-up voltage, it selects a single wake-up voltage but adjusts the capacitor size for each task. To do so, the authors designed a reconfigurable array of capacitors, which is parameterized with user annotations in the code.

Some studies directly incorporate capacitor voltage into task execution scheduling, either by defining a per-task minimum voltage requirement [18] or by proposing voltage interrupts that trigger task execution [7]. However, in both cases, this voltage must be selected manually, which can be complex and error-prone for developers.

To our knowledge, no research in static checkpointing has explored tailoring wake-up voltage to the application's needs at the granularity of the checkpoint. In most papers, the wake-up voltage is either unspecified or corresponds to a full buffer.

C. Related Works on Static Checkpointing Techniques

While no work in static checkpointing techniques focuses on checkpoint-tailored wake-up voltage, several works take an energy-aware approach.

Several studies focus on the checkpoint placement guided by the capacitor size. They modify the program binary and insert checkpoint to partition the program into regions immune to power failures, using worst-case execution time data [5, 25] or worst-case energy consumption data [3, 26].

Some techniques take a energy-aware approach at runtime, such as the authors of MEMENTOS [1] or Zhao *et al.*, [19], who skip the checkpointing process if the capacitor voltage is above a given threshold.

While many studies have focused on energy awareness, none have specifically addressed when the device needs to wake up, despite the significant impact of the wake-up voltage on device execution. EARLYBIRD aims to fill this gap by investigating the optimal wake-up voltage for intermittent computing devices.

IV. EARLYBIRD: AUTOMATIC SELECTION AND ENFORCEMENT OF WAKE-UP VOLTAGE

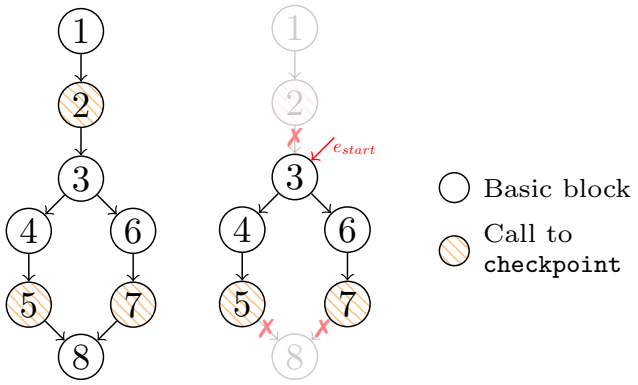
EARLYBIRD automatically selects the wake-up voltage adapted for each location in the program where the execution can be resumed from (*i.e.*, after `checkpoint()` routines). Our goal is to wake the system as soon as possible after a power failure, to improve progress regularity, and benefit from energy reduction due to low-voltage execution. However, EARLYBIRD must also ensure that the wake-up voltage is sufficiently high to reach the next checkpointing routine(s), in order to enforce forward progress.

To do so, EARLYBIRD relies on two components: a static analysis operating on a program's binary, that selects the minimal wake-up voltage for each checkpoint location, and a library to enforce the selected wake-up voltages at runtime.

A. Determination of Wake-up Voltage using Static Analysis

EARLYBIRD takes as input a static placement of checkpoints in the code of the application, through for example calling a `checkpoint()` routine. Determining the wake-up voltage is performed for each checkpoint location, using static analysis of the program binary code. The analysis, for a given checkpoint location, determines the largest amount of energy required to reach the next checkpoint location(s). Additionally, an alert is raised if the selected capacitor size is insufficient to reach the next checkpoint, and highlights the critical path that may require additional checkpoints.

The analysis operates on the control flow graph of the program (a control flow graph, or CFG, is a graph in which nodes represent basic blocks – sequences of instructions without internal branching – and edges represent control flow between them). Specific basic blocks are dedicated to function calls and only contain the function call instruction.



(a) `main` CFG (b) Analysis from basic block 3

Fig. 5: Control flow graph of a function `main` (a), and representation of an analysis when resuming execution from basic block 3 (b).

Figure 5a depicts a CFG. Hatched basic blocks represent function calls, in this simple example calls to the checkpointing routine `checkpoint()`. The static analysis of EARLYBIRD, for each checkpoint location c (for example basic block 2) aims at finding the maximum energy consumption among all paths p such that:

- p starts directly after the given checkpoint location c (in our example basic block 3);
- p ends with a checkpoint location that is reachable from c (in our example basic blocks 5 or 7);
- the basic blocks from p do not contain any checkpoint location except its last basic block.

Enumerating all paths to identify which one is the most energy consuming may lead to combinatorial explosion as in general the number of paths in a program is exponential. Therefore, the static analysis of EARLYBIRD leverages IPET (Implicit Path Enumeration Technique) [27], originally designed for Worst-Case Execution Time (WCET) estimation, to identify the worst-case energy consumption required from one checkpoint to reach the subsequent ones.

1) Original Implicit Path Enumeration Technique (IPET): IPET implicitly explores all possible execution paths within a program to determine its worst case execution time or energy consumption. IPET proceeds by solving an Integer Linear Programming (ILP) problem. The variables in the ILP problem are the execution counts of basic blocks (n_i) and edges ($e_{i \to j}$) in the CFG. The ILP problem constrains the values of these variables to model:

1) *The control flow between basic blocks*:

$$n_i = \sum_{j \in pred(i)} e_{j \to i} = \sum_{j \in succ(i)} e_{i \to j} \qquad (1)$$

expressing that the number of execution of a node n_i is equal to the number of times its incoming and outgoing edges are executed;

2) *Function calls*, for every call site of a function:

$$n_{call\_site} = n_{callee\_entry\_point} \quad (2)$$

3) *Loop bounds*, for every basic block i in a loop body:

$$n_i \leq M \times \sum_j e_{j \to k} \quad (3)$$

with M the maximum number of iterations of the loop and $j \to k$ the entry edges of the loop;

4) *Entry point*. The execution count of the first basic block s of the function whose WCET/WCEC is computed is set to 1:

$$n_s = 1 \quad (4)$$

ILP solvers then maximize the objective function $\sum n_i \times C_i$, with C_i the constant representing the WCET (respectively Worst Case Energy Consumption - WCEC) of basic block i.

2) EARLYBIRD*'s modifications to IPET:* The IPET formulation was designed to determine the WCET/WCEC of an entire function (usually *main()*), and (implicitly) explores paths, with no constraint besides flow constraints. In EARLYBIRD, the paths to be explored are constrained as follows:

C1 Considered paths start after a call to routine `checkpoint()` and end with a call to routine `checkpoint()`, but must not contain any other call to `checkpoint()`;

C2 The entry point of the analysis (following a call to routine `checkpoint()`, for example basic block 3) may be at any location in the code (inside a function, inside a loop), and is not necessarily the first basic block of a function.

Constraint C1 is accounted by (virtually) pruning the CFG (*i.e.*, not generating any IPET constraint) for the nodes and edges that violate constraint C1. Let us call s the entry point for the analysis (basic block following the first call to routine `checkpoint()`. A Breadth-First Search (BFS) algorithm starting from s explores the CFG (following regular edges and function returns); all traversed nodes and edges are added to a reachable set R. The search stops when a call to routine `checkpoint()` is found. Constraints are generated only for the nodes/edges in set R. Figure 5b represents the CFG considered in the analysis, after pruning (pruned nodes/edges are shaded).

Accounting for constraint C2 is more tricky, due to the arbitrary location of the start basic block s in the CFG.

To account for the case where s is located inside a function `g` (*i.e.*, the path starts in a function body), the function call constraint of the original IPET (Equation 2) must be modified, as the existing constraint only links the execution of the basic block calling the function `g` and the entry point of `g`, without any constraint to model function returns. *Virtual inlining* is used to manage function calls. Equation 2 is replaced by constraints on the virtual edges that represent the function entry (from calling basic block to function entry) and return (from function return to basic block following the function call). This is depicted in Figure 6, where basic block 3 that calls function `g` is replaced by the body of `g`.

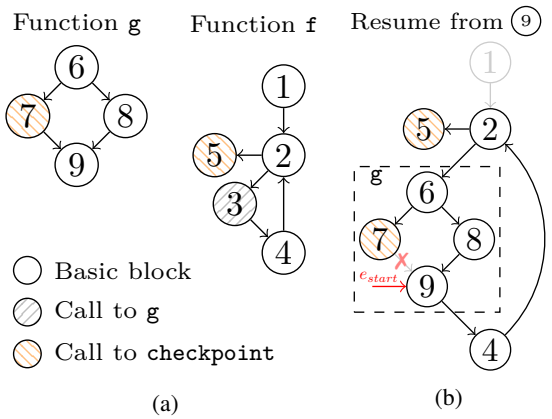


Fig. 6: Control flow graph of two functions `f` and `g`, where `f` calls `g` in basic block 3 (a). The analysis when resuming execution from basic block 9 is displayed in (b).

It may also happen that the start basic block s is located within a loop. In this case, the entry point might be executed multiple time (*e.g.*, basic block 9 in Figure 6b). This is incompatible with the current entry point constraint formulation ($n_s = 1$), so we introduce a new fictitious edge denoted as e_{start} that points to the entry point n_{start}, and set the execution count of this edge to 1.

Moreover, since the entry of the loop is now n_{start} and not the original loop entry, the IPET constraint on loop bounds (Equation 3) has to be modified, and is now $n_i \leq M \times (e_{start} + e_{j \to k})$ for all basic blocks i in the loop.

B. Enforcing a wake-up voltage at runtime

Enforcing a wake-up voltage raises two challenges. First, re-compiling a program to account for the wake-up voltages computed off-line might change the program binary, and thus its energy consumption, which is not desirable. Second, an energy-efficient way to monitor the capacitor voltage is required.

These two challenges are addressed through the provision of a checkpointing library with two simple checkpointing routines: `checkpoint(VOLTAGE)` and `checkpoint()`. The first routine allows to manually select a wake-up voltage for the given checkpoint. It lets the user stay in control of the wake-up voltage and bypass EARLYBIRD if needed. For example, this can be used to ensure that a peripheral executes in its recommended voltage range.

With the second routine, `checkpoint()`, the user delegates the burden of selecting a wake-up voltage to EARLYBIRD. At compile time, the `checkpoint()` routine is replaced by a `checkpoint(<PLACEHOLDER>)`. This placeholder is later replaced in the final binary by the value of the computed wake-up voltage. This operation is automated and transparent for the user.

Then, at runtime, when a checkpoint location is reached, the wake-up voltage is given as a parameter of the checkpointing routine, and saved alongside the checkpoint. On restart, the

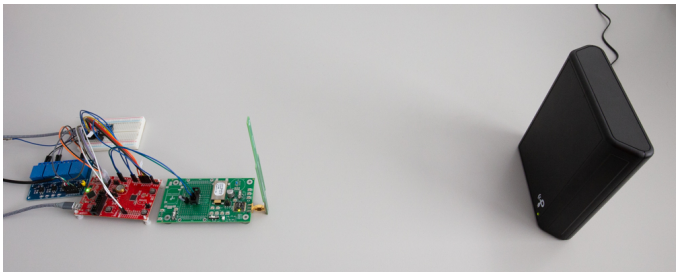


Fig. 7: Experimental setup. The msp430 (in red) is powered by the P2110 RF harvester (in green), with the RF emitter visible on the right (the distance between the transmitter and the harvester is not to scale here). The msp430 is monitored by an Arduino RP2040, and is connected to a laptop to flash new programs via several relays (in blue). Those relays isolate the msp430 when running an experiment.

wake-up voltage is read from the saved checkpoint, and it is the responsibility of the `restore` routine to make sure enough energy has been harvested before resuming program execution. To do so, we provide two mechanisms: a *software-assisted* and a *hardware-only* solution.

The software-assisted is designed to be compatible with most boards, as it only requires the board to be equipped with an analog-to-digital converter (ADC) and a timer. The timer periodically wakes-up the processor, which triggers a measure of the voltage across the capacitor, and checks if it is above the selected threshold.

The hardware-only requires an ADC that is able to perform a measure triggered by a timer, and that is equipped with a window comparator, that raises an interrupt when a software-defined threshold has been reached. This mechanism does not requires any CPU computation, as everything can be done in hardware: a timer is set up to tick at the chosen rate, and trigger the measure of the capacitor voltage. This voltage is compared to the threshold inside the ADC, and the CPU is only woken-up when this threshold has been reached.

V. Experimental Evaluation

A. Experimental Setup

We evaluated EarlyBird on the msp430fr5969 microcontroller ("msp430" for short). The msp430 is popular in the battery-free community as it features an energy-efficient non-volatile ferromagnetic RAM (FRAM) that enables low-overhead checkpointing. In our experiments, the device is powered using a Powercast P2110 RF energy harvester, similarly to previous works [5, 7, 28]. The RF source is a 3 W RF emitter positioned 1 meter away from the harvester. A capacitor of 100 µF is used as energy buffer. Figure 7 depicts the experimental setup.

We implemented EarlyBird as two main tools. The binary analysis of EarlyBird is implemented as an extension to the open-source worst-case execution time estimation tool Heptane [29]. The EarlyBird's runtime is developed as a C library for the msp430 microcontroller. All sources are available online[1].

We evaluated the techniques on three benchmarks from the Mibench2 benchmark suite [30]: *aes*, *crc* and *rc4*. Checkpointing routines were inserted in the code of benchmarks using four checkpoint placement heuristics:

- The first two heuristics place checkpoints based on the code structure and represent checkpointing schemes where the program state is saved frequently. These heuristics, borrowed from Mementos [1], are called *LL* (for Loop Latch) and *FR* (for Function Return). They insert calls to the checkpointing routine respectively inside each loop (at each basic block source of a back-edge, termed Loop Latch) and before each function return.
- The two other checkpoint placement heuristics take as input the capacitor size. Placement is such that execution from one checkpoint location can reach the next checkpoint(s) with the energy contained in the capacitor. Using the checkpoint placement technique of Schematic [26], we generated two checkpoint placements: one for the actual capacitor size and another one assuming that the capacitor is 10 times smaller than it actually is. We refer to them as *S-100%* (full capacitor) and *S-10%* (10 % of the capacitor).

The worst-case energy model used in our experiment estimates the worst-case energy consumption of a basic block based on its execution time and the microcontroller current consumption data documented in its datasheet [31]. It is computed using the following formula:

$$E = V \times I \times T \tag{5}$$

with

- V=3.3 V the microcontroller operating voltage;
- I=1620 µA the microcontroller maximum current consumption executing instructions in sequence at 16 MHz;
- T the basic block worst-case execution time.

The experimental protocol consists in executing a benchmark as many times as possible, during two minutes, using harvested energy. The number of times the benchmark completes is used as the main metric to evaluate performance. To monitor the execution status, the runtime library is instrumented to signal the execution states such as sleeping, re-execution or checkpointing. In all experiments, all the program data except the code are stored in volatile memory.

A word about capacitor voltage monitoring: Before diving into the actual evaluation of EarlyBird, we first evaluated the software-assisted and hardware-only voltage monitoring mechanisms. For each mechanism, we measured the time required to replenish the capacitor from 1.8 V to 3 V. With both strategies, the capacitor voltage is sampled at a fixed rate, defined by its period T. We ran experiments with T varying from 5 ms to 100 ms.

[1] https://gitlab.inria.fr/early-wake-up/early-wake-up-experimental-setup/

Surprisingly, the software-assisted technique proved to be the most efficient, with a time to replenish the capacitor shorter by 40 % on average compared to the hardware-only technique (1.2 s, against 1.7 s on average for the hardware-only technique). Indeed, despite having to regularly wake up the CPU to measure the voltage, the software-assisted technique allows for shutting down the ADC when not in use. In contrast, the hardware-only technique requires the ADC to be always active. Overall, the energy savings achieved by shutting down the ADC during idle periods outweighs the additional energy consumption associated with regularly waking up the CPU. For the following experiments, the software-assisted technique is used, with sampling period T of 40 ms.

B. Impact of EARLYBIRD *on Performance*

We evaluate EARLYBIRD against two wake-up voltage strategies, named in the following V-HIGH and V-LOW. As introduced in Section II, both strategies select a unique wake-up voltage for the program: 3.3 V for V-HIGH and 1.88 V, the msp430 built-in wake-up voltage, for V-LOW. Figure 8 displays the overall throughput improvement of V-LOW and EARLYBIRD strategies relative to the V-HIGH strategy. This improvement is computed as the ratio $r = \frac{N_{strategy}}{N_{\text{V-HIGH}}}$, with N_X the number of benchmarks completed when using strategy X. The raw number of benchmark executed for each technique is available in Table III.

TABLE III: Number of benchmarks executed with the V-HIGH, V-LOW and EARLYBIRD wake-up strategies.

Checkpoint Placement	Benchmark	Number of benchmarks executed V-HIGH	V-LOW	EARLYBIRD
LL	aes	15	24	34
	crc	10	12	21
	rc4	21	27	46
FR	aes	190	295	415
	crc	3934	6000	8035
	rc4	44	53	92
S-10%	aes	344	0	524
	crc	1426	1389	2122
	rc4	12	13	20
S-100%	aes	372	0	440
	crc	1580	1308	2303
	rc4	848	0	1266

The V-LOW strategy allows for a performance improvement as compared to V-HIGH in some cases, but fails at executing some benchmarks under some checkpoint placements. Moreover, the improvement obtained with V-LOW is limited, as considerable time is spent re-executing portions of code whose progress could not be saved. In contrast, the EARLYBIRD strategy outperforms both V-HIGH and V-LOW. EARLYBIRD demonstrates significant performance improvements across all benchmarks compared to V-HIGH, with a geometric mean improvement of 1.76×. When compared to V-LOW in settings where it ensure forward progress, it provides a geometric mean improvement of 1.57×.

The performance improvement of EARLYBIRD can be explained by looking at how the experiment time is split across sleeping time, computation time, re-execution time and checkpointing, as displayed in Figure 9. While EARLYBIRD is agnostic to the checkpoint placement method, the choice of a checkpoint placement deeply impacts the execution of an intermittent program. With frequent checkpointing techniques (*LL* and *FR*), a significant portion of time is spent checkpointing. It becomes extreme with *LL*, where computations are nearly not visible on the Figure. In occasional checkpointing techniques, much less time is spent checkpointing at the cost of an increased time spent re-executing code. Nonetheless, regardless of the checkpointing technique, a significant portion of time is dedicated to replenishing the capacitor (sleeping). V-HIGH spends most of its time sleeping because its power consumption is higher due to the msp430 operating at a higher voltage on average. In contrast to V-HIGH, V-LOW and EARLYBIRD benefit from being active a larger portion of the time. However, due to its low wake-up voltage, V-LOW encounters power failures more frequently, and thus spends an important portion of time restoring the saved program state, and re-executing code that was not committed.

Overall, the experiments show that, by tailoring the wake-up voltage to each checkpoint, EARLYBIRD:

- ensures the progress of the program in all benchmarks, with all checkpoint placement heuristics;
- reduces the sleep time, as the execution at low voltage reduces the device energy consumption;
- increases the amount of time spent executing, which leads to a significant increase in the number of benchmarks it is able to execute compared to a strategy that fully replenish the capacitor (V-HIGH).

As detailed in the introduction, the wake-up voltage also impacts the length of sleeping phases. Long sleeping phases may lead to the device being unaware of its environment for long periods of time. This is illustrated by measuring the time between two benchmark completions, as displayed in Table IV. When looking at the median value, we observe that for a benchmark like *crc*, having a short execution time, most of the times the interval between benchmark completions is 40 ms, which is the benchmark duration. Indeed, the benchmark can be executed several times in a computation burst, even with a capacitor that is not full. In this case, the intervals between benchmark completions that are above the 99 % quantile correspond to the sleep time. For *crc*, the V-HIGH strategy shows an important sleep time compared to the benchmark execution time, and V-LOW demonstrates the more regular benchmark execution. The other benchmarks (*aes* and *rc4*) have a longer execution time and cannot execute entirely with a full capacitor. In both case, EARLYBIRD keeps a low time between each benchmark completion, demonstrating its reactivity.

C. Ability of EARLYBIRD *to Improve Existing Checkpointing Techniques*

In this section, we evaluate how EARLYBIRD can improve two existing checkpointing techniques, namely MEMENTOS and SCHEMATIC. Compared to the checkpointing strategies used in Section V-B, that were clearly already inspired from

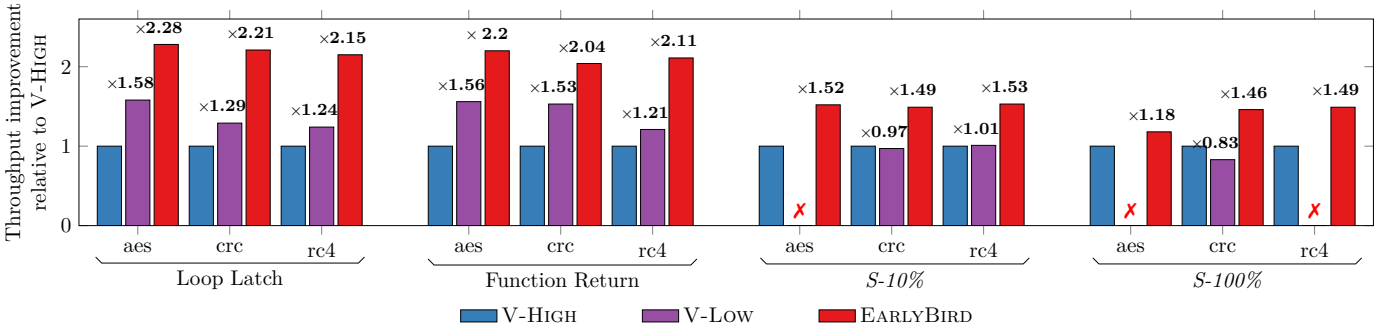


Fig. 8: Throughput improvement relative to V-HIGH (ratio of number of benchmarks executed). A red cross (✗) indicates that execution could not complete. Note that bars from different checkpointing strategies cannot be directly compared due to the normalization with respect to V-HIGH. For detailed data on the number of benchmark completions, refer to Table III.

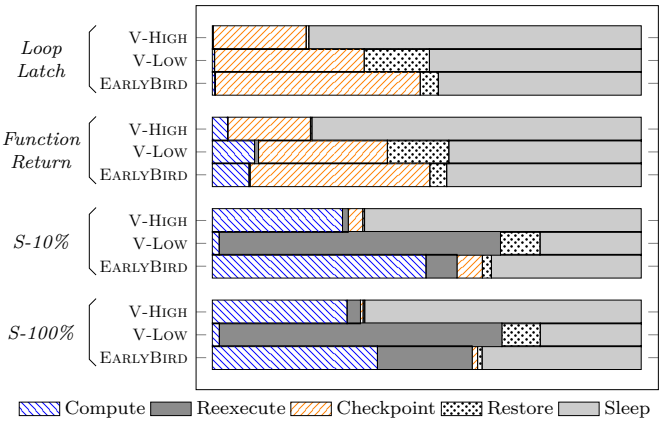


Fig. 9: Distribution of time on the *aes* benchmark.

TABLE IV: Interval between two benchmark completions in milliseconds (Median, 95 % quantile and 99 % quantile). The checkpoint placement is *S-10%*.

Benchmark	Wake-Up	Median	95 %	99 %
aes	V-HIGH	117	1020	1064
	V-LOW	Could not complete		
	EARLYBIRD	208	318	327
crc	V-HIGH	40	40	927
	V-LOW	40	91	92
	EARLYBIRD	40	40	139
rc4	V-HIGH	9240	9533	9662
	V-LOW	9062	9327	9375
	EARLYBIRD	6115	6251	6337

MEMENTOS and SCHEMATIC, here we use all optimizations implemented in MEMENTOS and SCHEMATIC, for example the ability of MEMENTOS to skip checkpoints at run-time.

MEMENTOS [1] inserts checkpointing routines on Loop Latches (*LL*) or Function Returns (*FR*) at compile-time. Then, at runtime, energy is saved by skipping unnecessary checkpoints, *i.e.*, checkpoints are saved only when a low-voltage threshold has been reached (here 2 V). We extended MEMENTOS with EARLYBIRD capabilities: we resume program execution as soon as enough energy has been harvested (originally MEMENTOS was waiting for full capacitor replenishment). Moreover, we adapt the skip decision: instead of testing if the current energy is above a given threshold, we test that there is enough energy to reach the next checkpoint(s).

SCHEMATIC [26] is a technique that places checkpoints in a program given the size of the device capacitor, using static analysis together with a worst-case energy consumption model. It also performs memory allocation of the program variables (in volatile and non-volatile memory, facility that was not evaluated here). In contrast to techniques presented before, SCHEMATIC makes sure that the capacitor is fully replenished after each checkpoint, and that checkpoints are close enough to be reached with a full capacitor. Checkpoint placement in SCHEMATIC is constrained by the code structure, potentially leading to conservative placement decisions (*e.g.*, if multiple iterations of a loop can execute with a full capacitor but not the entire loop, a checkpoint is still inserted in the loop). We extended SCHEMATIC with EARLYBIRD: instead of waiting for the capacitor to be full, we wait until it has enough energy to reach the next checkpoint(s).

The results for MEMENTOS are visible in Table V. When augmented with EARLYBIRD, MEMENTOS experiences a important throughput improvement with all benchmarks and checkpoint placement, with on average 1.74× more benchmarks executed. The results for SCHEMATIC are visible in Table VI. Augmenting SCHEMATIC with EARLYBIRD allows a performance improvement greater than four (geometric mean of ×4.63).

TABLE V: Number of benchmarks executed with MEMENTOS and MEMENTOS augmented with EARLYBIRD.

Benchmark	Checkpoint Placement	Benchmarks Executed: MEMENTOS	Benchmarks Executed: MEMENTOS +EARLYBIRD	Improvement
aes	LL	21	38	×**1.78**
	FR	137	228	×**1.66**
crc	LL	19	34	×**1.79**
	FR	992	1773	×**1.79**
rc4	LL	45	71	×**1.79**
	FR	81	131	×**1.62**

TABLE VI: Number of benchmarks executed with SCHEMATIC and SCHEMATIC augmented with EARLYBIRD, with *S-100%* checkpoint placement.

Benchmark	Benchmarks Executed		
	SCHEMATIC	SCHEMATIC +EARLYBIRD	Improvement ratio
aes	142	549	×**3.87**
crc	407	2299	×**5.65**
rc4	296	1351	×**4.56**

VI. CONCLUSION

This paper has presented EARLYBIRD, a novel technique for improving checkpointing techniques in battery-less devices. By automatically computing a wake-up voltage tailored to each checkpoint, EARLYBIRD offers a promising approach for improving the performance of existing techniques. Experimental results demonstrated the effectiveness of EARLYBIRD in increasing throughput, highlighting its potential to enhance the efficiency of IoT devices operating in energy-constrained environments. While EARLYBIRD already significantly improves the performance of existing techniques, we believe further enhancements are possible. Future research could explore incorporating execution context, such as loop iterations, to fine-tune the wake-up voltage at runtime. Additionally, enhancing capacitor monitoring techniques, by predicting when the capacitor voltage will meet the required level, could reduce unnecessary voltage measurements and further optimize energy consumption.

ACKNOWLEDGMENT

This work has received a French government support granted to the Labex CominLabs excellence laboratory and managed by the National Research Agency in the *Investing for the Future* program under reference ANR-10-LABX-07-01.

REFERENCES

[1] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *ASPLOS XVI*, 2011. DOI: 10.1145/1950365.1950386

[2] KEYSIGHT, "N6705a DC Power Analyzer."

[3] B. Yarahmadi and E. Rohou, "Compiler optimizations for safe insertion of checkpoints in intermittently powered systems," in *SAMOS*, 2020. DOI: 10.1007/978-3-030-60939-9_12

[4] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Efficient intermittent computing with differential checkpointing," in *LCTES*, 2019. DOI: 10.1145/3316482.3326357

[5] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity," in *RTAS*, 2022. DOI: 10.1109/RTAS54340.2022.00012

[6] V. D. Woude, Joel and M. Hicks, "Intermittent Computation without Hardware Support or Programmer Intervention," in *OSDI*, 2016. ISBN 978-1-931971-33-1

[7] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive Kernel for Tiny Batteryless Sensors," in *SenSys '18*, 2018. DOI: 10.1145/3274783.3274837

[8] M. Hicks, "Clank: Architectural Support for Intermittent Computation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017. DOI: 10.1145/3079856.3080238

[9] D. Pala, I. Miro-Panades, and O. Sentieys, "Freezer: A Specialized NVM Backup Controller for Intermittently Powered Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. DOI: 10.1109/TCAD.2020.3025063

[10] H. Jayakumar, A. Raha, and V. Raghunathan, "QuickRecall: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers," in *27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, 2014. DOI: 10.1109/VLSID.2014.63

[11] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *Embedded Systems Letters, IEEE*, 2015. DOI: 10.1109/LES.2014.2371494

[12] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems," *IEEE Transactions on Computers*, 2019. DOI: 10.1109/TC.2018.2889080

[13] A. J. Neto, A. Caulfield, C. Alvares, and I. De Oliveira Nunes, "$\mathcal{D}$iCA: A hardware-software co-design for differential check-pointing in intermittently powered devices," in *ICCAD*, 2023. DOI: 10.1109/ICCAD57390.2023.10323895

[14] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An energy-aware runtime for computational RFID," in *NSDI'11*, 2011.

[15] B. Lucia and B. Ransford, ""Dino:" A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, 2015. DOI: 10.1145/2813885.2737978

[16] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *OOPSLA*, 2016. DOI: 10.1145/2983990.2983995 pp. 514–530.

[17] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Coala: Dynamic Task-based Intermittent Execution for Energy-harvesting Devices," *ACM Transactions on Sensor Networks*, 2020. DOI: 10.1145/3360285

[18] A. Sabovic, A. K. Sultania, C. Delgado, L. D. Roeck, and J. Famaey, "An Energy-Aware Task Scheduler for Energy Harvesting Battery-Less IoT Devices," *IEEE Internet of Things Journal*, 2022. DOI: 10.1109/JIOT.2022.3185321

[19] M. Zhao, C. Fu, Z. Li, Q. Li, M. Xie, Y. Liu, J. Hu, Z. Jia, and C. J. Xue, "Stack-Size Sensitive On-Chip Memory Backup for Self-Powered Nonvolatile Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017. DOI: 10.1109/TCAD.2017.2666606

[20] V. Kortbeek, S. Ghosh, J. Hester, S. Campanoni, and P. Pawełczak, "WARio: Efficient code generation for intermittent computing," in *PLDI*, 2022. DOI: 10.1145/3519939.3523454

[21] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Lazy Scheduling for Energy Harvesting Sensor Nodes," in *IFIP*, 2006. DOI: 10.1007/978-0-387-39362-9_14

[22] B. Islam and S. Nirjon, "Scheduling Computational and Energy Harvesting Tasks in Deadline-Aware Intermittent Systems," in *RTAS*, 2020. DOI: 10.1109/RTAS48715.2020.00-14

[23] F. Yang, A. S. Thangarajan, G. S. Ramachandran, W. Joosen, and D. Hughes, "AsTAR: Sustainable Energy Harvesting for the Internet of Things through Adaptive Task Scheduling," *ACM Transactions on Sensor Networks*, 2021. DOI: 10.1145/3467894

[24] A. Colin, E. Ruppel, and B. Lucia, "A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[25] N. A. Bhatti and L. Mottola, "HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing," in *IPSN*, 2017.

[26] H. Reymond, J.-L. Béchennec, M. Briday, S. Faucou, I. Puaut, and E. Rohou, "SCHEMATIC: Compile-Time Checkpoint Placement and Memory Allocation for Intermittent Systems," in *CGO*, 2024. DOI: 10.1109/CGO57630.2024.10444789

[27] Y.-T. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1997.

[28] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2017.

[29] D. Hardy, B. Rouxel, and I. Puaut, "The Heptane Static Worst-Case Execution Time Estimation Tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, 2017. DOI: 10.4230/OASIcs.WCET.2017.8

[30] Matthew Hicks, "Mibench2: MiBench benchmark suite ported for IoT devices." 2016. [Online]. Available: https://github.com/impedimentToProgress/MiBench2

[31] Texas Instruments, "MSP430FR5969 datasheet," Tech. Rep., 2012.