POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN COMPUTER SCIENCE AND ENGINEERING

BREAKING PROGRAMMING ABSTRACTIONS BOUNDARIES TO INCREASE THE EFFICIENCY OF INTERMITTENTLY-POWERED COMPUTERS

Doctoral Dissertation of:
Andrea Maioli

Supervisor:
Prof. Luca Mottola

Tutor:
Prof. Raffaela Mirandola

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2023 – XXXV Cycle

To my grandparents.

To my family.

"Get ready for major remodel, fellas!
We're back in hardware mode."
Anthony E. Stark

Abstract

BATTERY-LESS devices represent a great opportunity to enable a sustainable Internet of Things: ambient energy harvesting replaces batteries, leading to zero-maintenance systems with a low environmental impact. However, despite potentially supplying unlimited free energy, ambient energy is irregular, unpredictable, and usually insufficient to power battery-less devices continuously. Therefore, battery-less devices experience frequent and unpredictable energy failures that lead to intermittent computation, as devices compute only when sufficient energy is available.

The presence of energy failures introduces several connected challenges. Unlike mainstream platforms, battery-less devices consist of highly constrained microcontroller units that run single non-concurrent programs and lack an operating system to manage energy failures. Therefore, when battery-less devices shut down due to energy failures, they lose the computational state and, in the next power cycle, they restart the computation all over again. To ensure battery-less devices progress in their programs, they periodically need to save their program state onto a non-volatile memory location, which is persistent across energy failures, to restore it when the energy returns. Although this ensures program forward progress across energy failures, battery-less devices may experience unexpected behaviors, producing results that differ from those of an equivalent continuous execution.

Ensuring program forward progress and avoiding unexpected behaviors introduce energy and computation overhead detrimental to battery-less devices' performance. Therefore, efficient energy management becomes essential to extract the most possible work from harvested energy, as it sup-

plies an unpredictable, limited, and scarce amount of energy.

The PhD research described in this thesis tackles these challenges and provides several contributions to the state of the art.

We first work on the first multi-year deployment of battery-less devices [4] that monitor the structural conditions of an archeological site. We devise three system design iterations, where we initially deploy battery-powered systems. Due to the high maintenance efforts of frequent battery replacements, we eventually switched to battery-less systems powered with kinetic and thermal energy. Our final design achieves zero-maintenance battery-less operations without compromising end-user requirements, as its sensed data provides comparable insights to battery-powered systems.

We then target intermittence anomalies [75], consisting of unexpected behaviors caused by energy failures. We classify intermittence anomalies and identify new types of anomalies previously overlooked by existing literature, which may happen whenever devices interact with the environment. We devise a set of techniques to analyze their occurrence, and we design `ScEpTIC` [69], an open-source tool to test intermittent programs.

Building on our work on intermittence anomalies, we devise intermittence awareness [72], a program design pattern that intentionally allows the occurrence of specific intermittence anomalies to gain new information regarding intermittent executions of programs. We show the potential of intermittence awareness by designing an intermittence-aware technique that reduces the energy overhead required to preserve the computation achieved inside loops. On average, our technique demonstrates a $35.2x$ lower energy consumption and a $48.4x$ faster workload completion time.

Next, we focus on improving the energy efficiency of mixed-volatile platforms, which feature a directly-addressable non-volatile memory location where developers can manually allocate portions of the program state. We design `ALFRED` [70, 73], a virtual memory abstraction and compilation pipeline for mixed-volatile platforms that automatically identifies the most efficient mapping of the program state across volatile and non-volatile memory. Our experiments show that `ALFRED` reduces programs' energy consumption by up to two orders of magnitude.

Finally, we focus on ensuring that battery-less devices always operate in the most efficient settings. We devise a system design to efficiently regulate supply voltage and clock frequency in highly resource-constrained battery-less devices. We then implement two hardware/software co-designs that capture these features. Our designs reduce battery-less devices' energy consumption by up to 170% and workload completion time by up to one order of magnitude.

Riassunto

I DISPOSITIVI senza batteria rappresentano una grande opportunitá per una nuova Internet of Things sostenibile. Le tecniche di energy harvesting permettono di utilizzare l'energia presente nell'ambiente come unica fonte di alimentazione per i dispositivi dell'Internet of Things, eliminando l'utilizzo di batterie. I dispositivi senza batteria risultanti hanno un basso impatto ambientale e potenzialmente non richiedono alcuna manutenzione.

Nonostante l'energia ambientale sia potenzialmente illimitata, essa é irregolare, imprevedibile e di solito non sufficiente per alimentare continuamente i dispositivi senza batteria. Conseguentemente, i dispositivi senza batteria subiscono frequenti ed imprevedibili interruzioni di energia, ed il loro modello di computazione diventa intermittente, dato che possono eseguire i loro programmi solo quando é disponibile energia sufficiente.

La presenza di interruzioni di energia introduce nuove sfide connesse tra di loro. A differenza dei dispositivi comuni alimentati con fonti energetiche stabili, i dispositivi senza batteria consistono in microcontrollori ad alto risparmio energetico e con bassa potenza di calcolo, eseguono singoli programmi e non dispongono di un sistema operativo che gestisce le interruzioni di energia. Conseguentemente, quando i dispositivi senza batteria si spengono a causa di un'interruzione di energia, perdono lo stato computazionale e devono poi ricominciare la computazione da capo quando l'energia torna disponibile. Per garantire che i dispositivi senza batteria procedano nei loro programmi e producano risultati utili, devono periodicamente salvare lo stato del loro programma in una memoria non volatile, che

non perde il suo contenuto quando il dispositivo si spegne. Questo permette ai dispositivi senza batteria di ripristinare lo stato salvato quando l'energia torna disponibile. Nonostante questo meccanismo garantisca l'avanzamento del programma in caso di interruzioni di energia, i dispositivi senza batteria potrebbero presentare comportamenti imprevisti, producendo risultati diversi da quelli di un'esecuzione continua equivalente.

Le operazioni richieste per garantire l'avanzamento del programma ed evitare comportamenti imprevisti incrementa il consumo energetico e la computazione effettuata dai dispositivi senza batteria, riducendone le loro prestazioni. Dato che queste operazioni sono necessarie per il corretto funzionamento dei dispositivi senza batteria, diventa necessaria una gestione efficiente dell'energia, in modo da poter estrarre il maggior quantitativo di computazione possibile dall'energia presente nell'ambiente.

La ricerca di dottorato descritta in questa tesi affronta queste sfide e fornisce diversi contributi allo stato dell'arte.

Per prima cosa, abbiamo lavorato alla prima implementazione pluriennale di dispositivi senza batteria che monitorano le condizioni strutturali di un sito archeologico [4]. Abbiamo inizialmente utilizzato dei sensori alimentati unicamente con le batterie. A causa degli sforzi elevati e periodici richiesti per la sostituzione di quest'ultime, abbiamo deciso di passare a sistemi senza batteria alimentati unicamente con energia cinetica e termica. I dispositivi risultanti sono in grado di operare senza batterie, e non richiedono alcuna manutenzione. Nei nostri risultati abbiamo appurato che i dispositivi senza batteria da noi ideati forniscono informazioni comparabili ai sistemi alimentati a batteria, rispettando quindi i requisiti dell'utente finale.

Successivamente, ci siamo focalizzati sulle anomalie da intermittenza [75], che corrispondono a comportamenti imprevisti dei dispositivi senza batterie, causati dalle interruzioni di energia. In questo lavoro, abbiamo classificato le anomalie da intermittenza e abbiamo identificato nuovi tipi di anomalie, precedentemente trascurati dalla letteratura esistente, che possono verificarsi quando i dispositivi senza batteria interagiscono con l'ambiente circostante. Abbiamo quindi ideato una serie di tecniche per verificare la presenza delle anomalie da intermittenza e abbiamo progettato `ScEpTIC` [69], uno strumento open source per testare i programmi intermittenti.

Partendo del nostro lavoro sulle anomalie da intermittenza, abbiamo ideato un nuovo modello di programmazione per progettare i programmi eseguiti sui dispositivi senza batterie [72]. Tale modello consente intenzionalmente il verificarsi di specifiche anomalie da intermittenza, al fine di ottenere nuove informazioni sull'esecuzione intermittente del programma.

Per dimostrare l'efficacia di questo modello, lo abbiamo utilizzato per creare una tecnica che riduce il consumo energetico e il carico computazionale richiesti per preservare il calcolo effettuato all'interno di cicli. In media, la nostra tecnica dimostra un consumo energetico inferiore di 35.2 volte ed riduce il tempo di computazione di 48.4 volte.

Successivamente, ci siamo concentrati sul miglioramento dell'efficienza energetica delle piattaforme a volatilitá mista, che dispongono di una memoria non volatile direttamente indirizzabile dal microcontrollore, dove é possibile allocare manualmente parti dello stato del programma. Abbiamo quindi progettato `ALFRED` [70, 73], che consiste in un'astrazione di programmazione basata sul concetto di memoria virtuale e in una pipeline di compilazione per piattaforme a volatilitá mista. `ALFRED` identifica automaticamente qual é la mappatura piú efficiente dello stato del programma tra la memoria volatile e non volatile. I nostri esperimenti mostrano che `ALFRED` riduce il consumo energetico dei programmi fino a due ordini di grandezza.

Infine, ci siamo concentrati sull'assicurare che i dispositivi senza batteria funzionino sempre nelle impostazioni operazionali piú efficienti. Abbiamo individuato un insieme di caratteristiche necessarie a regolare in modo efficiente la tensione di alimentazione e la frequenza di clock in dispositivi senza batteria con risorse limitate. Abbiamo poi progettato due sistemi che catturano queste caratteristiche. Nei nostri esperimenti, i nostri sistemi riducono il consumo energetico dei dispositivi senza batteria fino al 170% e il tempo di completamento dei fino a un ordine di grandezza.

Acknowledgements

The Ph.D. research described in this thesis was a lonely and tortuous path, full of stress, frustration, deadlines, satisfaction, laughs, and joy. This path was possible through the help of many people who supported me in various ways, both from academic and personal life standpoints.

Academic. I want to express my deepest gratitude to my advisor, Luca Mottola, for his support, dedication, invaluable guidance, and patience. Working with him was inspiring, and his feedback was essential to improve my writing and research skills and to complete this journey. I owe him most of what I know about doing research.

Second, I want to thank Josiah Hester and his research group for their help, support, and guidance during my stay at Northwestern University (IL, USA). It was a very inspiring experience that I will never forget.

Finally, I want to thank the thesis reviewers, Marco Zimmerling and Bashima Islam, for their valuable feedback.

Personal. The most enormous thank you goes to my family for their unconditional love and support throughout this journey and my entire life.

Second, I extend my appreciation to all my friends who provided me with a supportive network during these years. Listing you all would be impossible, but you know who you are!

Finally, throughout my life, many people thought and told me that I was not good enough to get things done, but here I am. To all these people, I just want to say thank you for the extra push and motivation, and I thank myself for getting here.

Thank you.

Andrea

Contents

CHAPTER 1

Introduction

Several billions of small-scale battery-powered embedded devices are deployed in the Internet of Things and enable a variety of smart applications such as environmental monitoring, industrial automation, and traffic control. However, batteries must be periodically recharged, replaced, and eventually disposed of, resulting in high maintenance costs and causing a significant environmental impact.

To address these problems, systems rely on ambient energy harvesting [15] to power battery-less devices, potentially yielding zero-maintenance IoT systems with greater long-term sustainability [4,22,47,92,93,101]. Figure 1.1 depicts an example of battery-less devices' architecture. The sensor node is parallel connected to an energy harvester and an energy buffer. The energy harvester is specific to the ambient energy source considered and converts ambient energy into electrical energy, powering the sensor node and recharging the energy buffer. The energy buffer consists of a capacitor or a super-capacitor and smooths the fluctuations of harvested energy. More complex designs exist, which use multiple energy harvesters [22], arrays of energy buffers [27, 42, 102], or use voltage converters to operate the energy harvester and the sensor node at different operating points [5, 10, 27, 39]. Throughout this thesis, we consider an architecture with a single energy

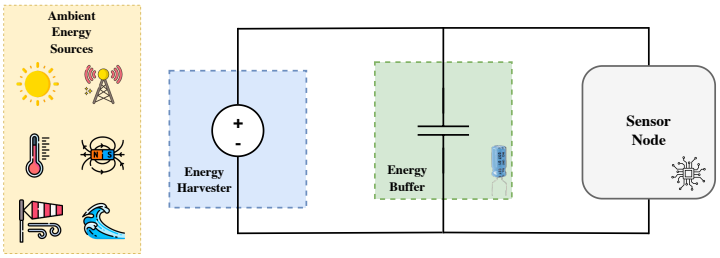
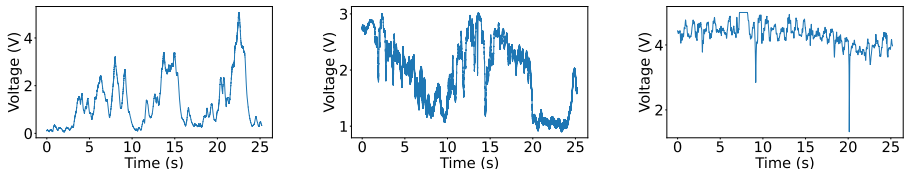


Figure 1.1: *Example of battery-less devices' architecture.*



(a) RF energy source (b) Solar energy source - indoor (c) Solar energy source - outdoor

Figure 1.2: *Voltage trace of various ambient energy sources measured in our lab.*

harvester, a single energy buffer, and no voltage regulator between the energy harvester and the sensor node.

Ambient energy and intermittent computation. Ambient energy sources vary from the most common solar energy to thermal energy, kinetic energy, and the energy carried by RF signals [15]. Figure 1.2 shows an example of the voltage trace of various ambient energy sources, which experience frequent and unpredictable fluctuations. Due to its irregular nature [15], harvested energy is unpredictable and usually insufficient to power these devices continuously.

Despite using energy buffers, battery-less devices experience frequent and unpredictable energy failures that cause an *intermittent computation* throughout their entire lifetime, as Figure 1.3 depicts. A battery-less device is initially powered off and harvests energy to recharge its energy buffer (*Charging*). When the energy buffer stores sufficient energy, the device powers on and starts computing (*On*). Note that the device energy consumption is usually significantly higher than the ambient energy intake. Consequently, throughout the computation, the device completely depletes its energy buffer and eventually shuts down, experiencing an energy failure. The device goes back to recharging its energy buffer, and this behavior repeats for the entire device's lifetime.

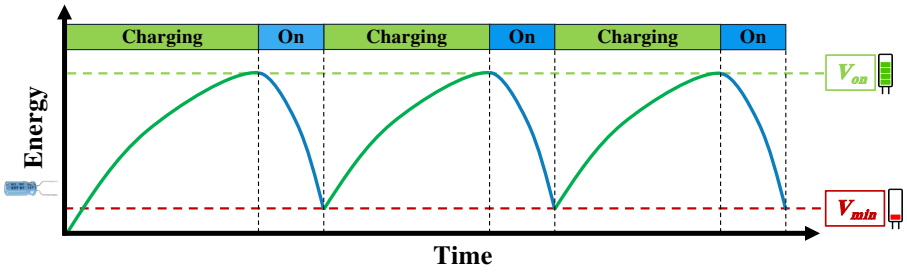


Figure 1.3: *Example of the intermittent computation of battery-less devices.*

1.1 Intermittent Computing Challenges

Ambient energy harvesting provides a limited energy throughput. Depending on the type of energy source, the supplied power ranges between hundreds of μW and tens of mW [15]. Therefore, to operate under such limited power throughput, battery-less devices consist of ultra-low-power and highly resource-constrained Microcontroller Units (MCU) with a main memory size in the order of tens of kilobytes, such as the ultra-low-power MCUs from the MSP430 MCU family from Texas Instruments [51]. Battery-less devices usually run single C applications, and, differently from mainstream computers, they lack an operating system that manages the occurrence of energy failures. These limitations, in combination with the presence of energy failures and the erratic nature of ambient energy, introduce several challenges that prevent using battery-less devices as mainstream sensors for the Internet of Things [43, 63]. These challenges include ensuring program forward progress, ensuring program consistency, program testing, enabling efficient operations, enabling communications, enabling peripheral accesses, keeping track of time, ensuring timely executions, and deploying battery-less devices. These challenges are connected and affect each other. We describe next some of the main challenges, whereas Section 1.2 describes the challenges we tackle in this thesis.

Program forward progress. Energy failures harm program forward progress. When devices shut down due to energy failures, they lose the computational state, as the content of volatile memories (i.e., registers and main memory) is preserved only when sufficient energy is available. Consequently, they lose the computational state and must restart the computation from scratch when the energy returns. This prevents devices from completing their programs or producing useful results.

As we describe in Chapter 3, ensuring program forward progress across

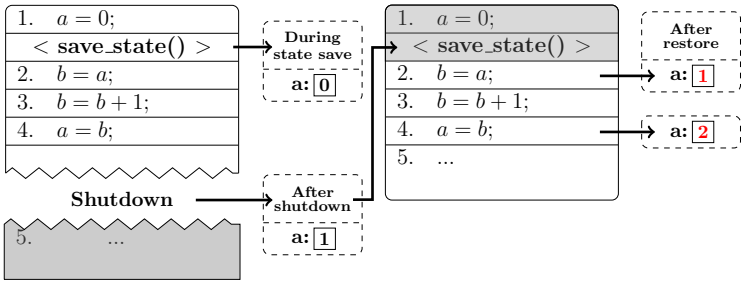


Figure 1.4: *Example of an intermittence bug [74].* An energy failure causes the re-execution of portions of a program, leading to an unexpected result.

energy failures requires devices to periodically save their state onto a non-volatile memory (NVM) location, which is persistent across energy failures. The saved state usually includes the content of the register file, special registers such as the program counter and the stack pointer, and the content of volatile main memory. Then, when the energy returns, restoring the saved state from non-volatile memory allows devices to resume the computation from where the state was saved.

Ensuring program forward progress across energy failures represents one of the main challenges of intermittent computing. Researchers are exploring various ways to preserve the device state across energy failures. Existing techniques [11, 12, 16, 52, 54, 64, 66, 68, 86, 100, 103] mainly differ on how devices save and manage their state, and at which point during the program execution the state is saved [8]. Devices can explicitly save their entire state onto non-volatile memory [11, 12, 16, 86] or rely on mixed-volatile platforms [49, 50] to directly allocate portions of the program state [54, 64, 66, 100] (e.g., single variables [64] or entire stack segments [54, 100]) onto a built-in non-volatile memory location. The latter eases persistent state management. The program state directly allocated onto non-volatile memory is excluded from state-saving operations, as it is automatically retained across energy failures and thus implicitly preserved

Program consistency. Energy failures may cause battery-less devices to produce results different than an equivalent continuous execution of the same program. This makes battery-less devices prone to experience unexpected behaviors unattainable in a continuous execution, recognized in the literature as *intermittence bugs* [64, 74, 85, 100].

Figure 1.4 shows an example. The program of Figure 1.4 runs on a mixed-volatile MCU, and variable a is allocated onto non-volatile memory. Line 1 sets a to 0, and then the device saves its volatile state onto

non-volatile memory. Note that the saved state does not include a, as it is already non-volatile. Next, the execution of lines 2-4 increments a to 1, and then an energy failure occurs. When there is sufficient energy, the device powers on, restores the saved state from non-volatile memory, and resumes the computation from line 2. Variable a is not restored, as it was not included in the saved state. As such, a retained the effects that line 4 produced during the previous power cycle, that is, a future operation with respect to where the computation resumes [85]. Here is where the intermittence bug happens. Lines 2-4 re-execute and further increment a to 2. This result differs from an equivalent continuous execution of the program of Figure 1.4, which would instead set a to 1.

Existing forward progress techniques avoid intermittence bugs by saving the program state more frequently [74, 100], or by including portions of non-volatile memory into state-save operations [64, 66].

Testing intermittence programs. The presence of intermittence bugs requires developers to test the intermittent executions of their programs to verify they are free of unexpected behaviors. However, energy failures create new requirements for testing the behaviors of programs. Existing testing techniques for mainstream computation are not suited to analyze intermittent executions, as they do not account for energy failures. Testing intermittent programs requires the possibility of reproducing energy failures [74], allowing developers to reproduce specific patterns of intermittent executions. Moreover, testing intermittent programs may require repeatable reproductions of energy harvesting sources, which is non-trivial and require custom hardware solutions [41].

Energy Optimization. Energy harvesting sources are unpredictable and supply limited energy [15]. Therefore, battery-less devices must efficiently manage harvested energy to maximize the work achieved in each power cycle. However, all the operations required to address the challenges introduced by energy failures introduce an energy overhead that may harm devices' performance. For example, the state-save operations required to ensure program forward progress introduce an energy overhead [8, 12, 16, 86], as the device pauses the program execution to save the program state. A similar case applies to the operations that ensure correct peripheral accesses [14, 17, 87]. Using mixed-volatile platforms [49, 50] may increase devices' energy consumption, as non-volatile memory is slower and less efficient than volatile memory [49, 50]. Moreover, avoiding intermittence bugs [74] requires additional state-save operations, further increasing the energy overhead of state-save operations. Therefore, extracting the maximum possible computation from harvested energy is non-trivial and may

require complex optimizations [55, 66, 72, 73, 103] or custom hardware designs [5].

Peripherals accesses. Embedded sensing devices periodically monitor and interact with the environment using peripheral devices, such as sensors and actuators. The presence of energy failures challenges peripheral accesses [14, 17, 67, 87].

Peripheral devices may have a volatile internal state that, similarly to the program state, requires to be preserved across energy failures. Otherwise, peripherals may be left in an inconsistent or non-initialized state when devices shut down due to energy failures. Consequently, when energy returns and the device resumes the computation, access to peripheral devices may fail or produce unexpected results [14,17,87]. Addressing this challenge requires representing peripherals' state inside the device's main memory [87], writing custom restore routine for peripheral states [14], postponing peripheral operations when sufficient energy is available [27], or using custom program abstractions that capture peripheral states [17, 67].

Network and communication. Embedded sensing devices may need to communicate sensed data to other devices. Being peripherals, radios and communication devices experience the same problems as peripheral accesses. Moreover, energy failures harm radio communications and networking. Energy failures cause nodes to disappear from networks, failing to transmit or receive data. Further, the energy cost of data transmission may be prohibitive using the severely limited energy budget of battery-less devices, as it has a higher energy consumption than normal computation [4]. Addressing these problems requires the design of dedicated protocols [62, 79] or custom communication devices [57, 89].

Tracking time. Energy failures cause battery-less devices to lose track of time. Being battery-less devices highly resource-limited, they usually lack real-time clocks. Therefore, battery-less devices can track time only when active, using timers and counters internal to the MCU [51]. Although counters can be preserved with the program state, devices are unable to track the time elapsed while they were powered off due to energy failures. Addressing this problem requires relying on physical phenomena, such as capacitor charge decay [45] or volatile memory cells decay [45], to estimate the time elapsed since the energy failure.

Timely Executions. Energy failures may prevent programs from executing tasks within their time-based deadlines [52, 59, 68]. Embedded sensing devices may need to periodically execute time-sensitive sensing operations within a given deadline [52, 59, 68]. Similarly, sensed data may expire

after a given period, requiring programs to process it within a given deadline [59, 97].

Energy failures during task execution prevent the task to complete. As energy failures can last for long periods, interrupted tasks may fail to meet their deadlines. Moreover, the inability of battery-less devices to track the time elapsed while powered off may prevent them from identifying when tasks cannot meet their deadlines. Addressing this problem requires dedicated scheduling strategies [52], reserving portions of energy buffers only for the execution of time-sensitive tasks [68], or custom program semantics for binding time requirements to energy failures [59].

Usability and deployments. The proposed solutions to the various challenges introduced by power failures mainly target researchers, and their application requires a broad knowledge of intermittent computing. This prevents the spread and adoption of this technology and results in the lack of significative examples of battery-less device deployments. Unlike the various examples of deployments of battery-powered sensors [13, 19–21, 30, 34, 46, 60, 77, 80, 82, 98], existing deployments of battery-less devices only demonstrate specific techniques [22, 37, 47, 84, 91] without accounting for actual end-user requirements and real-world scenarios. Deployments that meet end-user requirements are necessary to identify this technology's possible pitfalls and demonstrate its potential.

1.2 Research Challenges

This section describes the challenges we tackle in this thesis as directions of the PhD research. Note that, as we later show in this section, these challenges are all interconnected and the way we tackle one challenge affects the others.

1.2.1 Real Use Cases

As we anticipate in Section 1.1, the literature provides several examples of deployments of battery-powered sensors in various environments and at different scales [13, 19–21, 30, 34, 46, 60, 77, 80, 82, 98]. These deployments show how the need for battery replacement introduces significant maintenance costs and usually leads to unreliable operations. In contrast, battery-less systems require almost zero maintenance, as there is no longer the need to periodically recharge, replace, and eventually dispose of batteries. This potentially unlocks a deploy-and-forget scenario, where battery-less devices can be left untouched for years, while still providing useful data.

Although researchers actively tackle the various challenges created by the instabilities of harvested energy [43,63,85], the literature provides very few examples of long-term deployments [22,37,47,84,91] that demonstrate the potential of this research field. As we discuss in Chapter 2, existing deployments of battery-less devices are relatively short, where the longest-running one has just a 3 months lifetime [22]. Further, they aim only at demonstrating specific and isolated techniques, and do not account for real applications or end-user needs.

Long-lasting real-world deployments of battery-less devices targeting real application scenarios are necessary to understand and demonstrate the potential of this technology. The lessons learned from such deployments would provide valuable experiences from which researchers can identify new possible pitfalls and further improve this technology.

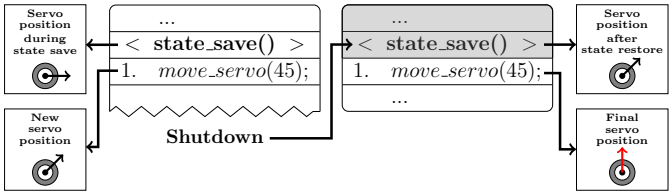
The absence of many deployments of battery-less devices also indicates that this technology is not ready for a mainstream audience. Deploying battery-less devices should be as easy as uploading the firmware to devices and placing them in the deployment zone. However, existing techniques required to ensure safe and reliable intermittent operations target researchers and assume skilled developers with a deep knowledge of this research field. Moreover, only a subset of the techniques available in the literature is open-source and available for end users or researchers to use. This prevents the widespread adoption of this technology and hardens the possibility of new long-lasting deployments.

For these reasons, deployments of battery-less devices are a great research opportunity that may open new research directions and provide useful experiences to the community. Therefore, researchers should focus on deploying battery-less devices and provide more accessible techniques to enable battery-free computations, improving the advancement and adoption of this technology.

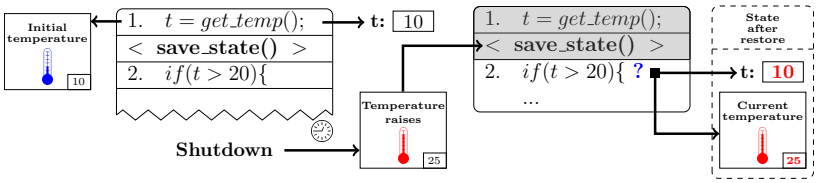
1.2.2 Intermittent Program Consistency

As we anticipate in Section 1.1, energy failures may cause battery-less devices experience unexpected behaviors unattainable in a continuous execution, recognized in the literature as *intermittence bugs* [64, 74, 85, 100].

As we later describe in Chapter 4, the cause of the intermittence bug described in Figure 1.4 stands in the non-idempotent re-execution of a sequence of non-volatile memory read (line 2) and write (line 4) operations caused by energy failures [64,74,100]. Existing literature avoids the occurrence of intermittence bugs by (i) saving the state more frequently [74,100]



(a) The device alters the environment



(b) The device senses the environment

Figure 1.5: *Examples of unexpected behaviors happening when battery-less devices interact with the environment [75].*

to break the sequence of hazardous non-volatile memory accesses, or (ii) including portions of non-volatile memory onto the saved state [64, 66] to ensure data consistency when resuming the computation after an energy failure.

Environment interactions. Although current literature identifies intermittence bugs characterizing mixed-volatile platforms [64,74,85,100], it overlooks cases of unexpected behaviors that may occur whenever battery-less devices interact with the environment. Figure 1.5 depicts two examples of such unexpected behaviors.

Figure 1.5(a) shows an example with a pattern similar to the intermittence bug described in Figure 1.4. The device saves its state onto non-volatile memory, moves a servo by $45°$ and then the device shuts down due to an energy failure. When there is sufficient energy, the device restores the saved state from non-volatile memory and resumes the computation from line 1, which is re-executed. Consequently, the servo is moved by an additional $45°$, reaching a position of $90°$. Such servo position differs from the one of an equivalent continuous execution of the program of Figure 1.5(a), where the servo would be at $45°$. This example has a pattern similar to the example of Figure 1.4, where an energy failure causes the re-execution of a non-idempotent sequence of instructions and leads to unexpected behavior. However, despite such a similar pattern, existing solutions to avoid intermittence bugs do not work in this case, as more frequent state saves would not prevent the re-execution of line 1.

Let us now focus on Figure 1.5(b). This example demonstrates a different and previously-unseen pattern that leads to unexpected behavior due to energy failures. The device measures the environment temperature, say $10°C$, and then the program state is saved onto non-volatile memory. The execution continues until there is no energy left. Here the device immediately shuts down, and a long energy failure occurs. While the device is powered off, the environment temperature rises to $25°C$. When there is sufficient energy, the device powers on, restores the saved state from non-volatile memory, and resumes the computation from line 2. Here the device considers the environment temperature to be $10°C$, that is, the environment temperature measured during the previous power cycle found in the saved state. Consequently, although the current environment temperature is higher than $20°C$, the *if* statement of line 2 evaluates to *false*.

The unexpected behavior described in Figure 1.5(b) occurs for different reasons than the ones shown in Figure 1.4 and Figure 1.5(a). In the example of Figure 1.5(b), an energy failure causes the device to execute portions of a program in different power cycles, leading to a desynchronization between the actual environment state and the environment state held in the device's main memory.

The literature lacks an analysis of unexpected behaviors that may happen when devices interact with the environment, such as the one described in Figure 1.5. Consequently, there is no technique that ensures battery-less devices avoid these cases of unexpected behaviors.

Testing intermittent programs. In general, battery-less devices must avoid intermittence bugs and guarantee that the intermittent executions of a program produce the same results of an equivalent continuous execution. Otherwise, their behavior may be unpredictable or incorrect with respect to program specifications. However, ensuring programs are free of intermittence bugs or other unexpected behaviors is still an open problem. Although the literature analyzes intermittence bugs happening in mixed-volatile platforms [64, 74, 100], it still lacks practical ways to test intermittent executions of programs and to verify whether the program behaves as intended.

As we describe in Chapter 4, ensuring intermittent programs are free of unexpected behaviors introduced by energy failures is non-trivial. Testing intermittent programs requires the ability to reproduce energy failures and trigger state-saving operations at arbitrary points during testing, allowing developers to reproduce specific patterns of intermittent executions. Therefore existing program testing techniques cannot be used to test intermittent programs, as they are designed for continuous programs and do not account for energy failures or state-saving operations.

The literature provides very few techniques [24,41] to debug battery-less devices while they operate. However, due to the complexity of intermittent program testing, hardware-based debugging is not practical.

Compared to continuous techniques, the complexity and magnitude of operations involved in testing intermittent programs are drastically higher. Energy failures and state-saving operations can happen at any instant during program execution. Therefore, to verify that intermittent programs are free of unexpected behaviors, developers should test all the possible combinations of energy failures and state-saving operations, comparing the results of each combination against an equivalent continuous execution [74]. For instance, given a program with n machine-code instructions, we need to simulate the occurrence of n different energy failures, one after the execution of every instruction, thus generating n^2 possible intermittent executions [74]. This complexity further increases [74] when state-saving operations are not statically-fixed in the program's code [11, 12, 54].

Therefore, due to the complexity of intermittent program testing, emulation testing [35, 38, 74] seems to be the only practical approach. However, existing tools that emulate intermittent executions [38, 74] are not suited to exhaustively test intermittence programs, as either they only reproduce specific energy sources [38] or limit their analysis to memory-based intermittence bugs [74].

This challenge introduced by energy failures is still open, as current literature lacks (i) an extensive analysis of unexpected behaviors introduced by energy failures, and (ii) program analysis and testing techniques for intermittent programs. This poses severe limitations to the adoption of battery-less devices, as developers cannot verify the absence of unexpected behaviors.

1.2.3 Energy Efficiency

As we anticipate in Section 1.1, battery-less devices must efficiently manage harvested energy to maximize the work achieved in each power cycle, as energy harvesting sources are unpredictable and supply limited energy [15]. This requires focusing on two different aspects: (i) the operations required to ensure safe and reliable intermittent computations, namely, ensuring forward progress and avoiding intermittence bugs, and (ii) the device operating setting.

Ensuring intermittent computations. The operations required to ensure program forward progress across energy failures and to avoid unexpected behaviors are detrimental to device performance due to the execution of

state-save and restore operations. As we describe in Section 1.1, ensuring program forward progress requires devices to save their state onto a non-volatile memory location periodically. Similarly, avoiding unexpected behaviors requires devices to save their state more frequently, as we point out in Section 1.2.2. Saving the device state introduces a significant computational and energy overhead, as the device pauses the program execution and saves its state onto non-volatile memory. Further, non-volatile memory accesses are significantly slower and less energy efficient than volatile memory [49, 50, 72, 73]. For example, in mixed-volatile platforms, non-volatile memory accesses may require up to 247% more energy than volatile memory accesses and twice the number of clock cycles [50, 72]. These numbers further increase when non-volatile memory is connected as a peripheral external to the MCU [2, 61].

To reduce the overhead of state-save and restore operations, existing techniques [54, 55, 64, 66, 100] rely on mixed-volatile platforms to allocate portions of main memory onto non-volatile memory. However, this approach has two main drawbacks. First, the program execution now accesses the slower and less energy-efficient non-volatile memory, potentially increasing the overall energy consumption. Second, as we describe in Section 1.2.2, mixed-volatile platforms may experience intermittence bugs, whose avoidance requires devices to save the state more frequently, further increasing the energy consumption.

For these reasons, identifying the optimal and most efficient trade-off between a reduced volatile state against an increased energy overhead with the possibility of unexpected behaviors due to non-volatile memory operations is non-trivial, as it also depends on multiple factors, including the execution flow, workload, memory accesses, and energy patterns. As we point out in Chapter 3 and Chapter 5, the literature includes several solutions [7, 16, 55, 59, 66] that aim at reducing battery-less devices energy consumption while ensuring forward progress and avoiding intermittence bugs.

The focus of these solutions varies from identifying the most energy-efficient placement and execution strategy of state-saving operations [16], enabling differential state saving to reduce the size of the state saved onto non-volatile memory [7], disabling unnecessary state-saving operations at runtime to reduce their overhead [66], and dynamically relocating program sections across volatile/non-volatile memory to optimize the energy consumption [55]. However, these solutions reduce the volatile device state at the expense of increased computational complexity of state-saving operations or their execution frequency, potentially leading to sub-optimal per-

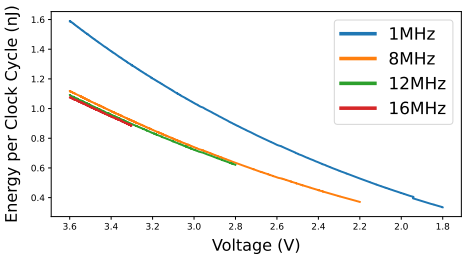


Figure 1.6: *Real measures [6] of the energy consumption of the MSP430-G2553 [48] factory-calibrated frequencies.*

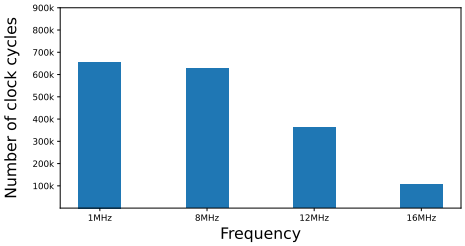


Figure 1.7: *Number of clock cycles executed by a MSP430-G2553 [48] in a single discharge of a* $100\mu F$ *capacitor from* $3.6V$ *to the minimum operating voltage of a given frequency.*

formance.

Device operating settings. Another factor affecting devices' energy consumption is their operating setting, including the operating voltage and the MCU operating clock frequency, which affect the energy consumed to execute each clock cycle. The energy cost e_{cc} of executing a single clock cycle can be calculated as $e_{cc} = \frac{I_{mcu} \cdot V_{op}}{f_{mcu}}$ [72], where I_{mcu} is the current draw of the MCU, V_{op} is the operating voltage of the MCU, and f_{mcu} is the operating frequency. Hence, to minimize e_{cc}, we need to operate the highest possible frequency supported by the MCU at the minimum possible operating voltage.

Battery-less devices usually lack input voltage regulation and are connected parallel to their energy sources and energy buffers. Therefore, battery-less devices operating voltage depends on the energy buffer level and harvested energy. As such, to minimize e_{cc}, we can mainly act on the operating clock frequency f_{mcu}, selecting the highest possible one, as higher operating clock frequencies demonstrate a lower energy consumption per clock cycle than lower operating clock frequencies [6]. However, f_{mcu} limits the minimum operating voltage V_{op} at which devices can operate; the higher the operating clock frequency, the higher the minimum voltage required to operate such frequency. For example, let us focus on Figure 1.6, which depicts the energy consumption per clock cycle and the operating voltage range of four factory-calibrated frequencies of the MSP430-G2553 [48], a MCU from the popularly used [6, 11, 54, 55, 65, 66] MSP430 family [51] of ultra-low-power MCUs. Here we notice that the operating clock frequency of $16MHz$ leads to a 68% lower energy consumption than $1MHz$. However, $16MHz$ requires a minimum operating voltage of $3.3V$, whereas the less efficient $1MHz$ requires a minimum operating voltage of $1.8V$.

Due to the reduced voltage range of higher operating frequencies, iden-

tifying the most efficient settings of battery-less devices is non-trivial. The most efficient operating clock frequency constantly changes throughout the computation, as it strictly depends on the voltage of the energy buffer, which limits the usable operating clock frequencies. When energy sources exceed or provide sufficient energy to match the device energy computation, the highest operating clock frequency represents the most efficient setting due to its lower energy consumption per clock cycle. In contrast, when devices' energy consumption exceeds harvested energy, the reduced operating voltage range of higher operating clock frequencies pones a severe disadvantage against lower frequencies.

In this last scenario, V_{op} eventually decreases throughout the consumption, as the device also draws the energy previously stored in the energy buffer. Consequently, higher operating clock frequencies execute fewer clock cycles before experiencing an energy failure due to their reduced operating voltage range. In contrast, less efficient lower operating clock frequencies can sustain the computation for longer periods. For example, let us focus on Figure 1.7, which depicts the number of clock cycles executed by the MSP430-G2553 [48] in a single discharge of a $100\mu F$ capacitor with no new incoming harvested energy. Despite a higher energy consumption, when the MCU operates at a frequency of $1MHz$, it executes $6.1x$ more clock cycles than at $16MHz$.

Therefore, due to the lower number of executed clock cycles in their active periods, higher operating clock frequencies may require more power cycles to complete a given workload than lower operating frequencies. This increases the workload completion time and the number of energy failures required by higher operating frequencies. Moreover, the higher number of energy failures experienced by higher operating clock frequencies further increases the workload completion time and the energy consumed due to state-save and state-restore operations [5], resulting in a worse performance than slower and less-efficient operating clock frequencies [5].

For these reasons, statically configuring battery-less devices with a fixed operating clock frequency leads to sub-optimal performance. To extract the best possible performance from harvested energy, battery-less devices must dynamically adapt their operating clock frequency to ensure they always select the most efficient operating clock frequency among the ones sustainable at a given energy buffer level. This technique is similar to the Dynamic Voltage and Frequency Scaling (DVFS) technique commonly used in mainstream processors.

However, applying DVFS to battery-less devices is challenging. Unlike mainstream processors, battery-less devices lack hardware support to

dynamically adapt the system operating voltage and frequency, and an operating system that manages such operations. Adding such capabilities may increase energy consumption, resulting in worse performance than a static frequency configuration. Further, battery-less devices' power supply is unstable and outputs an operating voltage subject to frequent changes. This requires constantly checking the energy buffer level to identify the available operating frequencies, which may further increase devices' energy consumption.

As we point out in Chapter 5, very little research is available that explores the application of DVFS to battery-less systems [5]. The literature mainly focuses on applying DVFS to achieve power-neutral operations [9, 10, 36], where devices' energy consumption matches harvested energy within a given time period. However, these works target multi-core processors that already have the required hardware capabilities. Their energy consumption usually exceeds the energy harvestable from the environment, making these devices unsuited for intermittent applications.

For the reasons we identified in this section, the challenge of improving the energy efficiency of battery-less devices is still open and provides a broad range of opportunities as a research direction.

1.3 Contributions

The PhD research focuses on various faces of the challenges we describe in Section 1.2. This section summarizes the contributions of the PhD research.

Figure 1.8 depicts how our contributions map to the research challenges, and shows the connection among our contributions. We mainly focus on intermittent program consistency and energy efficiency. Throughout the PhD research, we explore new designs to interface hardware and software, aiming to increase system efficiency while transparently ensuring program forward progress and avoiding unexpected behaviors.

1.3.1 Deployment of Battery-less Devices

This work tackles the real use case challenges described in Section 1.2.1. We work on the first multi-year deployment of battery-less devices that sense the structural and environmental conditions of an underground archaeological site [40, 99] in Rome (Italy).

The unique and extreme conditions of this site require unattended operations. Accesses to the site are strictly regulated and need to be as minimum as possible to avoid the deterioration of environmental conditions. Further, the site lacks a stable power source, and only kinetic and thermal energy

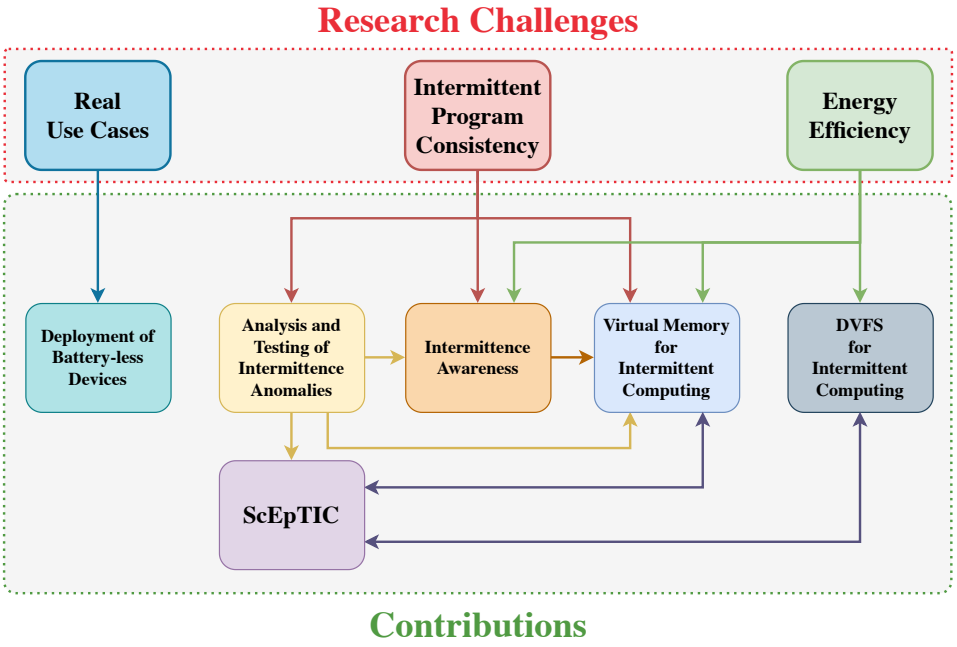


Figure 1.8: *Connections among challenges and research contributions.*

is available. These requirements represent a perfect use case example for battery-less devices.

We design three different iterations of battery-less systems, evaluating the use of multiple existing techniques for intermittent computing. The archaeologists using the sensed data confirmed that our deployment provides insight into environmental conditions comparable to one of battery-powered systems, despite providing fewer data due to energy failures. Moreover, our deployment is still been running with zero maintenance for over 4 years, demonstrating the potential of battery-less devices.

We published a paper summarizing the experiences and lessons learned from this deployment to the ACM Conference on Embedded Networked Sensor Systems (SenSys 2020) [4]. We describe the contributions of this work to the state of the art in Chapter 6, and we attach the paper in Chapter 12.

1.3.2 Analysis and Testing of Intermittence Anomalies

This work tackles the intermittent program consistency challenge described in Section 1.2.2. We provide an in-depth analysis of *intermittence anomalies*, consisting of unexpected behaviors characterizing battery-less devices,

and we devise a set of techniques to test all the possible intermittent executions of a program.

We expand the concept of intermittence bugs and extend our previously published work [74] on this topic with multiple new contributions. In particular:

- we identify new types of unexpected behaviors that energy failures may cause when devices interact with the environment;
- we provide a set of techniques to identify and analyze environment-related intermittence anomalies, and we provide a set of guidelines that allow developers to avoid their occurrence;
- we design a new technique to identify memory-related intermittence anomalies, which demonstrates a higher efficiency than our previous technique

We design `ScEpTIC`, an emulation environment to test intermittence programs, and we implement a prototype of our analysis techniques for intermittence anomalies. `ScEpTIC` code and documentation are available as open-source [69]. Note that we constantly update `ScEpTIC` throughout the PhD research, and we used it as an evaluation environment for our works.

We published this work at the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021) [75]. We describe the contributions of this work to the state of the art in Chapter 7, and we attach the paper in Chapter 12.

Finally, as Figure 1.8 shows, the knowledge we build from this contribution leads to precious insights that we later use to devise intermittence awareness design pattern [72] and our virtual memory abstraction [70, 73]. Further, throughout the research, we constantly update `ScEpTIC` with new features and we use it in multiple contributions to explore and evaluate our techniques design.

1.3.3 Intermittence-awareness Program Design Pattern

This work tackles the intermittent program consistency and energy efficiency challenges described in Section 1.2.2 and in Section 1.2.3, respectively. Note that this work builds on the knowledge we have developed during our work on intermittent anomalies [75], as Figure 1.8 shows.

We consider a new perspective on intermittence anomalies: we intentionally allow the occurrence of specific intermittence anomalies to gain new information regarding energy failures and intermittent executions of

programs without introducing unexpected behaviors. We call this concept *intermittence awareness*.

Intermittence awareness allows developers to consider intermittence as a new input for their programs. Developers can use this previously-unavailable information to make their programs react to energy failures, unlocking new program design patterns.

To demonstrate one of the many possibilities that intermittence-awareness unlocks, we design an intermittence-aware technique that reduces the energy overhead required to preserve the computation achieved inside loops. Our experiments show that, compared to existing forward progress techniques, our intermittence-aware technique shows, on average, a $32.5x$ lower energy consumption and a $48.4x$ lower workload completion time.

We published this work at the International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys '20) [72], where it received the Best Paper award. We describe the contributions of this work to the state of the art in Chapter 8, and we attach the paper in Chapter 12.

Finally, as Figure 1.8 shows, the knowledge we build from this contribution leads to precious insights that we later use to avoid intermittence anomalies in our virtual memory abstraction [70, 73].

1.3.4 Virtual Memory for Intermittent Computing

This work tackles the intermittent program consistency and energy efficiency challenges described in Section 1.2.2 and in Section 1.2.3, respectively. Note that this work builds on the knowledge we have developed during our works on intermittent anomalies [75] and intermittence awareness [72], as Figure 1.8 shows.

We design `ALFRED`, a virtual memory abstraction and compilation pipeline for mixed-volatile platforms that automatically identify the most efficient mapping of the program state across volatile and non-volatile memory. Our technique is completely transparent to developers and works on top of existing forward progress techniques.

`ALFRED` virtual memory abstraction relieves developers from mapping the program state across volatile and non-volatile memory, as they need not to specify a mapping for their programs. `ALFRED` compilation pipeline analyzes the input program and automatically maps virtual memory to volatile and non-volatile memory, identifying the most efficient mapping for each memory slice and code region.

The key behind `ALFRED` compilation pipeline is a set of program transformation techniques that decide what slice of the memory must be allo-

cated onto non-volatile memory and at what point in the program execution. The resulting mapping is not fixed, and it is automatically adjusted at different places in the program based on read/write patterns and program structure.

We implement a prototype of ALFRED in ScEpTIC [69, 70, 75], which we use as an evaluation environment. Moreover, we update ScEpTIC enabling simulations of devices' energy consumption and energy sources. Our experiments show that ALFRED improves the energy consumption by up to *two orders of magnitude*, with a comparable reduction of the workload completion time due to a significant decrease in the number of experienced energy failures.

We published this work at the ACM Conference on Embedded Networked Sensor Systems (SenSys 2021) [73]. A prototype of ALFRED code and documentation are available as open-source release [70] along with an extended technical report of ALFRED program transformation techniques [71]. We describe the contributions of this work to the state of the art in Chapter 9, and we attach the paper in Chapter 12.

1.3.5 DVFS for Intermittent Computing

This work tackles the energy efficiency challenges described in Section 1.2.3. Following early-stage research on DVFS for battery-less devices [5], we identify key functionalities required to efficiently enable DVFS in resource-constrained embedded MCUs that lack a dedicated DVFS controller.

The key functionality of our system design stands in identifying available MCU performance windows, consisting of the most efficient combinations of voltage and frequency settings. Low-power MCUs feature several factory-calibrated frequencies, where the MCU datasheet reports their corresponding minimum operating voltage. We then partition the energy buffer voltage into discrete energy levels and map each level to the most efficient performance windows supported at that level.

The system runtime identifies changes in the discrete energy level and applies the corresponding frequency and voltage. This removes the overhead introduced by periodic measurement of the energy buffer voltage, as the system tunes its operating settings only upon changes in the discrete energy level.

We use our system design rationale to implement two hardware/software co-designs for the MSP430-G2553 [48], one of which we materially fabricated.

As Figure 1.8, we rely on ScEpTIC to evaluate our system designs.

For doing so, we extend ScEpTIC with the ability of simulating devices' energy consumption, energy buffers, energy sources, circuitry external to the MCU, and custom hardware designs. Compared to static frequency configurations, our evaluation shows a reduction of the device energy consumption by up to 170% and a reduction of the workload completion time by up to one order of magnitude.

A paper summarizing this work was submitted to the ACM Transactions on Sensor Networks, and we are awaiting the reviews. We describe the contributions of this work to the state of the art in Chapter 10, and we attach the paper in Chapter 13.

Part I

Background

CHAPTER 2

Deployments

The literature provides several examples of long-term deployments of battery-powered systems [3, 19–21, 23, 28, 30, 31, 33, 60, 77, 78, 80, 82, 98], whereas very few deployments exist of battery-less systems [22, 37, 47, 53, 84, 91]. Section 2.1 briefly introduces various deployments for battery-powered systems, showing common pitfalls and various techniques to improve battery lifetime. Instead, Section 2.2 describes existing deployments of battery-less devices.

2.1 Battery-powered Systems

The literature demonstrates several deployments of battery-powered embedded sensing systems in various environmental scenarios and at different scales [3,19–21,23,28,30,31,33,60,77,78,80,82,98]. Common to these deployments is the requirement of periodic user intervention due to frequent battery replacements and limited performance due to battery depletion.

Battery-powered systems. Navarro et al. [82] demonstrate a two-year-long deployment of a 42-nodes wireless sensors network that monitors the temperature and humidity of a forested nature reserve. The average lifetime of batteries was 38 days, requiring periodic user intervention for their

replacement.

Szewczyk et al. [98] show a four-month-long deployment of a 150-nodes wireless sensor network that monitors seabirds' habitats during their breeding season. They try to design sensing nodes whose battery lifetime equals the required deployment time by designing a firmware that manages tasks execution, ensuring that batteries last for an entire breeding season. However, due to varying environmental conditions, deployed nodes show a 50% lower lifetime than expected, demonstrating that estimating battery depletion is non-trivial and may be erratic. Further, systems designed to last for a limited time still require user intervention to recollect the nodes for new deployments or to replace their batteries to extend the deployment period.

Marfievici et al. [77] report on a 17-month-long deployment of 30 nodes that monitor a data center's temperature, humidity, and airflow. This deployment demonstrates that battery replacement is required to ensure system operations and to grand a required level of performance. Despite achieving an extended battery lifetime of 15 months, the authors identify that nodes' performance depends on the remaining battery charge, as the system experienced a decrease in packet rate that returned to normal after replacing the batteries.

Other deployments [19–21, 30, 60, 80] show similar experiences, requiring periodic user interventions to keep nodes active and maintain their performance level.

Energy-harvesting and rechargeable batteries. Several deployments extend battery life and reduce user intervention using ambient energy harvesting and rechargeable batteries [3, 23, 28, 31, 33, 78] for their sensing nodes. Common to these deployments is the requirement of large batteries to sustain the computation for periods longer than systems with no rechargeable batteries. Therefore, these deployments demonstrate that energy harvesting alone does not extend batteries' lifetime enough to ensure nodes can sustain the computation for long periods.

Adkins et al. [3] show a city-scale deployment of six months where they deploy 12 nodes that monitor weather conditions, TV whitespace spectrum usage, and vehicular traffic at a university campus. Each node harvests solar energy through a solar panel and has a large $100Wh$ Li-ion battery pack. Although energy harvesting increases sensor nodes' battery life, this deployment shows that battery replacement is still required for long-term deployment scenarios.

Other deployments [23, 78] improve nodes' lifetime by using energy sources correlated with the monitored events. Martin et al. [78] report on a four-week deployment of 6 nodes monitoring showers' water flow and

usage. Sensing nodes are equipped with rechargeable batteries and harvest thermal energy from hot water pipes. The authors design each node to wake up only when hot water flows, leading to a 10-year estimated battery lifetime. Chiang et al. [23] target a similar application scenario and report on a two-week-long deployment of 23 nodes equipped with rechargeable batteries and electromagnetic generators that harvest kinetic energy from the water flow. The authors estimate that a single shower increases the device battery lifetime by 23 hours.

Battery alternatives. Deployments exist [28, 31] that validate system designs that consider battery alternatives to improve nodes' lifetime further.

Dutta et al. [31] use a four-month-long deployment of 557 sensor nodes to demonstrate a hybrid system design that combines batteries and super-capacitors for improved battery life. The authors equip each node with a small-sized solar cell to harvest solar energy, a $22F$ super-capacitor, and a $200mAh$ lithium polymer battery. The battery cannot sustain a 100% duty cycle, and each node uses the super-capacitors as a secondary energy buffer to ensure system operation when the battery is depleted, improving system lifetime.

Corke et al. [28] show a two-year-long deployment of 15 nodes powered with solar energy, comparing two different system designs: one using a rechargeable battery as an energy buffer and one using a super-capacitor. Although super-capacitors store $250x$ less energy than batteries, the system design using a super-capacitor lasted up to 27 hours without solar energy. This result demonstrates that super-capacitors can be a battery replacement for solar-powered sensor nodes.

2.2 Deployments of Battery-less Devices

The literature provides fewer examples of deployments for battery-less devices [22, 37, 47, 84, 91], where only a small fraction of them are real-world deployments that account for end-user needs [22, 47]. These deployments use single (super-)capacitors as energy buffers. Although the literature provides several architectures [27, 42, 102] that use arrays of (super-)capacitors as energy buffers, no real-world deployment exploits such architectures.

Super-capacitors. Following the examples of previous super-capacitor investigations [28, 31], other deployments [22, 37, 84] demonstrate system designs that rely on super-capacitors as the only energy buffer.

Fraternali et al. [37] report on a 15-day-long deployment of 20 sensor nodes for indoor smart building applications. Each node harvests solar energy and stores it into a $1F$ super-capacitor. During the computation, a

power-management algorithm increases the node lifetime by changing the BLE advertising rate and scaling the number of used sensors, depending on available energy. The results show that this system design maintained continual operations at the cost of a significantly reduced throughput when no solar energy was available. The authors suggest using bigger energy buffers and solar panels for an improved lifetime when no energy can be harvested from the environment.

Petrariu et al. [84] demonstrate a node design that harvests solar energy and stores it into a $4F$ super-capacitor using a 4-month-long deployment of a single node. The node periodically monitors the environment's temperature and humidity and uses the energy stored in the super-capacitor for data transmission over night-time when no solar energy is available. The authors report that the proposed system power supply is sufficient to achieve the expected level of performance.

Chen et al. [22] report on a 3-month-long deployment to monitor the water quality of a river. They deploy a gateway and 5 wireless sensor nodes that periodically measure the water temperature, dissolved oxygen concentration, and pH. The gateway harvests solar energy using a $20W$ solar panel, whereas each sensor node uses a microbial fuel cell (MFC) to harvest bioenergy from the river. MFCs supply a severely limited amount of energy, preventing the system from sustaining the lowest possible duty cycle that yields the data rate requirements of water quality measurement applications. Moreover, MFCs output voltage is up to $0.33V$, well below the system operating voltage of $1.8V$. To ensure system operations and periodic measurements of water quality, the authors equip each node with a custom power module, which features (i) a charge pump that boosts the MFC output voltage, (ii) a $200mF$ super-capacitor that acts as an energy buffer and stores harvested energy, and (iii) an RF switch that turns on the system by harvesting the energy carried by specific RF signals. The gateway transmits the RF signal, deciding when and for how long the sensor nodes collect and transmit data. The deployment shows that the achieved data rate meets the need of regular water quality monitoring applications, demonstrating unattended operations and better performance than manually-operated systems that require manually collected samples to measure water quality.

Capacitors. The literature provides very few examples of deployments that use capacitors as energy buffers [47, 91].

Saoda et al. [91] report on a week-long deployment of 38 sensor nodes that provide indoor tracking through BLE, harvest solar energy and use a capacitor as an energy buffer. The author's main objective is to evaluate how physical and design variables, such as node placement and energy

buffer size, affect system performance. From this deployment, the authors identify that a capacitor of $200uF$ stores sufficient energy for transmitting a single packet containing indoor tracking information. Further, the authors argue that prioritizing light intensity for node placement does not lead to higher system availability, which instead requires deployment schemes and runtime optimization strategies specifically designed for battery-less devices.

Ikeda et al. [47] report on three two-month-long deployments of a single node that monitors farm fields' temperature and soil moisture in Japan and India. The node uses a thermoelectric generator (TEG) to harvest energy from the temperature difference between air and underground soil and uses a $500\mu F$ capacitor as an energy buffer. These deployments allow the authors to evaluate the efficacy of the TEG and the system design, demonstrating that the TEG supplies more than sufficient energy to drive periodic temperature and soil moisture sensing.

CHAPTER 3

State Retention and Forward Progress

The literature provides several forward progress mechanisms [11, 12, 16, 25, 54, 59, 64–66, 76, 86, 88, 100, 103] that allow battery-less devices to achieve program forward progress across energy failures. All these techniques share the same underlying idea: they ensure devices periodically save their program state onto a non-volatile memory location, which is persistent across energy failures. When restarting after energy failures, devices restore their program state from non-volatile memory and resume the computation from where the state was saved. Note that the saved state must include all the information required to resume the computation, including the content of the main memory, register file, and special registers, such as the program counter and the stack pointer. These elements are usually volatile and need to be preserved across energy failures. Moreover, non-volatile memory can be either a peripheral external to the MCU and used only as program state storage [11, 12, 86], or a directly-addressable location internal to the MCU also used to allocate slices of main memory onto it [54, 64, 100], as in mixed-volatile platforms [51].

Forward progress, program consistency, and energy efficiency challenges are connected, and how we tackle one challenge affects the others, as we point out in Chapter 1.2. The existing forward progress mechanisms ex-

plore various approaches to ensure program forward progress across energy failures while reducing the energy and computational overhead introduced by state-saving operations and the ones introduced to avoid unexpected behaviors caused by energy failures, such as intermittence bugs [64, 74, 85, 100]. In this chapter, we provide basic knowledge on various forward progress mechanisms, whereas we later describe in Chapter 4 and Chapter 5 how they avoid intermittence bugs and how they improve the energy consumption, respectively.

The differences between the forward progress mechanisms available in the literature reside in four key aspects:

1. how the program state is saved onto non-volatile memory
2. when the program state is saved onto non-volatile memory
3. what slices of the program state are saved onto non-volatile memory
4. the mapping of program state across volatile and non-volatile memory

Considering these aspects, we can differentiate forward progress mechanisms into *checkpoint-based* [11,12,16,54,66,86,100] and *task-based* [25, 64, 65, 76, 88, 103] mechanisms. Checkpoint-based mechanisms rely on a special routine to save the program state, namely, a *checkpoint*. Instead, task-based mechanisms require developers to partition their program onto tasks and save the program state on task completion. We describe next the differences and characteristics of these two classes of forward progress mechanisms, along with examples of existing techniques.

3.1 Checkpoint-based Forward Progress Mechanisms

Checkpoint-based mechanisms ensure program forward progress with the help of two routines: (i) a *checkpoint* routine that saves the program state onto non-volatile memory and (ii) a *restore* routine that restores a previously-saved program state from non-volatile memory.

Checkpoint routines can execute in fixed places inside the program or at any instant during the program execution. The former case identifies a *static* checkpoint mechanism [16,66,86,100] whereas the latter identifies a *dynamic* or *just-in-time* checkpoint mechanism [11, 12, 54].

3.1.1 Static Checkpoint Mechanisms

Static checkpoint mechanisms place calls to checkpoint routines inside the program's code during compile-time, accordingly to a pre-defined placement strategy. Hence, checkpoint routines execute at fixed program locations decided at compile-time.

Checkpoint routines can either directly save the state or verify device runtime information, such as the energy buffer level, and consequently decide whether to save the state. In the former case, the checkpoint mechanism is in a *direct* configuration [100], whereas the latter is in a *conditional* configuration [16, 66, 86].

We describe next some static checkpoint mechanisms that introduce relevant concepts necessary to understand the contributions of the PhD.

Direct - Ratchet. Ratchet [100] is a static checkpoint mechanism with a direct configuration. Ratchet instruments the program code with function calls to the checkpoint routine, whose execution immediately saves the program state.

Ratchet checkpoint placement strategy is specifically designed to avoid intermittence bugs, as we later describe in Chapter 4. Ratchet analyzes the program's code and places function calls targeting the checkpoint routine to partition the program into idempotent code sections. An idempotent code section consists of sequences of operations that alter the program state only during their first execution. Further re-executions due to energy failures do not produce a different program state.

Ratchet targets mixed-volatile platforms, and its default configuration maps the entire content of main memory onto non-volatile memory, including global variables and the stack segment. As such, Ratchet checkpoint routine needs to save only the register file and special registers onto non-volatile memory, as the program's main memory is already persistent. This reduces the energy and computation overhead of the checkpoint routine.

However, as we later describe in Chapter 5, such overhead decrease comes at the cost of an increased program energy consumption due to accesses to the slower and less efficient non-volatile memory, which is used as main memory. Further, Ratchet checkpoint placement strategy produces frequent execution of the checkpoint routine, increasing the energy and computation overhead due to the executions of the checkpoint routine. Note that Ratchet also supports other memory mapping configurations, which, however, increase the checkpoint routine overhead.

Conditional - Mementos. Mementos [86] is a static checkpoint mechanism with a conditional configuration. Instead of inserting calls to checkpoint routines, Mementos instruments the program code with trigger calls that verify when the program state needs to be saved. When the execution reaches a trigger call, Mementos probes the energy buffer level by measuring its voltage through an ADC. If the measured voltage falls below a pre-defined threshold, Mementos saves the program state onto non-volatile memory by executing its checkpoint routine.

Mementos provides two different placement strategies to automatically instrument programs: *loop-latch* and *function-return*. The loop-latch placement strategy inserts a trigger call at the back edge of each loop, that is, the loop latch. This strategy ensures the execution of trigger calls at the end of each loop iteration. Instead, the function-return placement strategy inserts a trigger call after each function call inside the program. This strategy ensures the execution of trigger calls after the return of every function. Finally, Mementos also allows developers to insert trigger calls inside their programs manually.

Mementos maps main memory to volatile memory and uses non-volatile memory only to store the saved program state. As such, Mementos checkpoint routine saves the entire content of the main memory, the register file, and special registers, resulting in a significant energy and computation overhead.

To avoid unnecessary state saves and reduce checkpoints overhead, Mementos relies on probing the energy buffer level to decide when to save the state. However, as we later argue in Chapter 5, ADC probing may introduce a significant energy overhead.

Conditional - HarvOS. HarvOS [16] is a static checkpoint mechanism with a conditional configuration. Similarly to Mementos [86], HarvOS automatically places trigger calls inside programs to verify when the program state needs to be saved.

As we later describe in Chapter 5, HarvOS trigger calls placement strategy aims at minimizing the overhead of checkpoints and trigger calls. HarvOS splits the program into blocks of instructions and places a single trigger call within each block, where the allocated memory is minimum. This minimizes checkpoints overhead, as they execute where the amount of data to save onto non-volatile memory is minimum. Further, HarvOS sizes these blocks to minimize the number of trigger calls while ensuring the instructions between two trigger calls do not consume more energy than the device can buffer. Note that this prevents devices from being stuck at re-executing the same portion of a program due to insufficient energy to reach the next trigger call that saves a checkpoint [26].

Differently from Mementos [86], which uses a fixed voltage threshold to save the program state, HarvOS uses an adaptive threshold based on the energy remaining in the energy buffer. Whenever a trigger call executes, HarvOS probes the energy buffer level and executes the checkpoint routine only if there is insufficient energy to reach the next trigger call and to save the state next. Note that HarvOS identifies this information at compile-time, where it associates with each trigger call the energy necessary to reach the

next trigger call and save the state.

Finally, HarvOS maps main memory to volatile memory and uses non-volatile memory only to store the saved program state. As such, HarvOS checkpoint routine saves the entire content of the main memory, the register file, and special registers.

Conditional - Chinchilla. Chinchilla [66] is a static checkpoint mechanism with a conditional configuration. Similarly to Mementos [86] and HarvOS [16], Chinchilla aims at reducing checkpoints overhead. However, unlike Mementos [86] and HarvOS [16], which probe the energy buffer level, Chinchilla uses a timer to decide when to execute a checkpoint routine.

Chinchilla automatically places function calls to the checkpoint routine at the end of each basic block of a program. To minimize the execution of checkpoints, Chinchilla considers each checkpoint initially disabled and skips its execution when encountered. On system startup, Chinchilla sets a timer whose expiration dynamically enables the execution of checkpoint routines. When a checkpoint routine terminates its execution, Chinchilla dynamically disables checkpoint routines and resets the timer. Note that the timer interval is initially set at compile time and dynamically adjusted during runtime.

Chinchilla targets mixed-volatile platforms and maps main memory to both volatile and non-volatile memory. Global variables are mapped onto non-volatile memory, whereas the stack segment is shared among volatile and non-volatile memory. Chinchilla relies on program analysis techniques to map stack elements across volatile and non-volatile memory. Chinchilla maps to volatile memory the stack elements whose read and write operations do not cross any checkpoint, consisting in the elements whose write and read operations happen within the same basic block. All the other stack elements are mapped to non-volatile memory.

Chinchilla checkpoint routine saves the register file and special registers. However, it does not save the volatile portion of main memory, as it contains data that need not to be preserved across energy failures. Alongside with checkpoints, Chinchilla logs the write operations targeting non-volatile memory and avoids intermittence bugs [74, 85] by reverting these operations when restoring a checkpoint. We further describe this technique in Chapter 4.

3.1.2 Dynamic / Just-in-time

Dynamic or just-in-time checkpoint mechanisms rely on hardware interrupts to trigger the execution of checkpoint routines whenever the energy buffer level drops below a certain threshold. Hence, checkpoint routines can execute at any instant during the program computation.

We describe next some dynamic checkpoint mechanisms that introduce relevant concepts necessary to understand the contributions of the PhD.

Hibernus. Hibernus [11, 12] is a system design of a dynamic checkpoint mechanism.

Hibernus maps main memory to volatile memory and uses non-volatile memory only to store the saved program state. Hence, state-saving operations need to save the entire content of main memory, the register file, and special registers.

Hibernus targets the MSP430 platform [51] and relies on its dedicated hardware capabilities, consisting in (i) an on-chip comparator to generate interrupts and (ii) an on-chip variable voltage generator to set the comparator reference voltage (iii) the MCU low-power mode, which enters a deep-sleep state that keeps the content of volatile memory alive.

Hibernus considers two voltage thresholds to trigger state-save and state-restore operations: the hibernation voltage V_h and the resume voltage V_r. On system startup, Hibernus sets the on-chip comparator reference voltage to V_h. When the energy buffer voltage drops below V_h, the on-chip comparator triggers an interrupt that causes the execution of Hibernus *hibernate* routine. Here Hibernus saves the program state onto non-volatile memory, sets the on-chip comparator reference voltage to V_r, and sets the MCU into hardware deep sleep. The system is now in a low-power mode, waiting for new harvested energy, and retains its volatile state, including the content of main memory and registers.

If the energy buffer voltage raises to V_r, the on-chip comparator triggers an interrupt that wakes up the MCU. Hibernus sets the on-chip comparator reference voltage back to V_h, and the computation resumes without restoring the saved state. In fact, the MCU volatile state was not lost, as no energy failure happened. This allows Hibernus to reduce the overhead due to state-restoring operations.

Instead, if an energy failure occurs while Hibernus is in hibernation mode, the MCU volatile state is lost. Consequently, when sufficient energy is available, Hibernus restores the saved state from non-volatile memory, and the computation resumes.

Hibernus functionality relies on the voltage threshold values V_h and

V_r, whose optimal value changes during runtime, as it depends on energy harvesting and workload patterns [11]. Hibernus calculates and statically sets V_h and V_r at compile-time, potentially resulting in sub-optimal performance. To account for this limitation, Balsamo et al. [11, 12] propose an updated system design, Hibernus++ [11], which dynamically updates V_h and V_r accordingly to energy failure patterns and runtime system behavior.

QuickRecall. QuickRecall [54] is a system design of a dynamic checkpoint mechanism that shares some similarities with Hibernus [11, 12]. The main difference among these systems is that QuickRecall targets mixed-volatile platforms and maps the entire program's main memory onto non-volatile memory. Hence, QuickRecall checkpoints need only to save the content of the register file and special registers.

Similarly to Hibernus [11, 12], QuickRecall relies on an on-chip comparator to trigger interrupts that cause the execution of checkpoint and restore routines. However, unlike Hibernus [11, 12], QuickRecall considers a single voltage threshold V_{tr}, called trigger voltage.

When the energy buffer voltage drops below V_{tr}, the on-chip comparator triggers an interrupt that causes the execution of the checkpoint routine. Similarly to Hibernus [11, 12], QuickRecall saves the volatile portion of the program state and puts the MCU into hardware deep sleep. The system is now in a low-power mode, waiting for new harvested energy, and retains its volatile state, consisting of registers.

If the energy buffer voltage raises back to V_{tr}, the on-chip comparator triggers an interrupt that wakes up the MCU. Here QuickRecall resumes the computation without restoring the saved state, as no energy failure happened, and the volatile state was not lost. Otherwise, the MCU will eventually power off due to an energy failure. When sufficient energy is available, QuickRecall restores the saved state from non-volatile memory and resumes the computation.

QuickRecall uses non-volatile memory as main memory. Therefore, the computation must not continue after the program state is saved. Otherwise, an intermittence anomaly [74, 85] may happen.

3.2 Task-based Forward Progress Mechanisms

Task-based mechanisms require developers to design their programs as sequences of tasks, where each task consists of an atomic unit of computation. A runtime scheduler ensures program forward progress by automatically saving the state on task completion and ensures tasks always execute in the specified order, even in the presence of energy failures.

The literature provides several system supports and programming abstractions to partition programs into tasks [25, 64, 65, 76, 88, 103]. We describe next some task-based mechanisms that introduce relevant concepts necessary to understand the contribution of the PhD.

DINO. DINO [64] represents one of the first attempts to task-based forward progress mechanisms. DINO provides a novel view of checkpoints, considering them as task boundaries that specify the end of a task and the beginning of a new one.

DINO defines a programming model that requires developers to partition their programs into tasks by placing task boundaries into their programs, consisting of function calls to a routine of the DINO runtime library. DINO considers each sequence of instructions executed between two task boundaries as a task, granting atomicity and data consistency for each task. Task boundaries are statically defined at compile-time, whereas tasks are formed dynamically at runtime, depending on the placement of task boundaries and program control flow.

DINO ensures program forward progress across energy failures using the same technique of checkpoint mechanisms [11, 12, 16, 54, 86, 100]. When a task boundary executes, DINO saves the program state onto non-volatile memory, The saved program state includes the content of the main memory, the register file, and special registers, such as the program counter and the stack pointer. When resuming after energy failures, DINO restores the saved state from non-volatile memory and resumes the computation from the latest task boundary.

DINO targets mixed-volatile platforms and allows developers to allocate global variables onto non-volatile memory. Non-volatile variables may experience intermittence bugs [64, 74, 85, 100], which DINO avoids by ensuring a consistent program state through the inclusion of non-volatile variables into the saved state. We describe this aspect in Chapter 4.

Chain. Chain [25] is a task-based forward progress mechanism for mixed-volatile platforms. Unlike DINO [64], Chain provides a programming abstraction to design programs as collections of tasks, where developers define each task as a C function. Developers define task execution order directly in each task function, specifying the task to execute after the current task completion.

Tasks exchange data through dedicated non-volatile memory locations called channels, which developers explicitly define. Each channel is shared among two tasks, and Chain ensures an access pattern that avoids intermittence bugs [64, 74, 85, 100]. In general, a task can access a channel either in write mode to store its results or in read mode to access other tasks' results,

but not in both modes. This exclusive read-only or write-only access pattern ensures a consistent program state. We focus on this aspect in Chapter 4.

Chain considers tasks atomic units of computation and preserves forward progress at task granularity. On task completion, Chain saves a reference of the next task to execute onto a non-volatile memory location, called global execution context. Tasks explicitly save their results onto channels. On system startup, Chain resumes the computation by starting the execution of the task whose reference is saved in the global execution context. This ensures forward progress as long as each task can execute within a single charge of a capacitor [26].

Chain maps tasks runtime state onto volatile main memory and needs not to preserve task state across energy failures, as it considers tasks execution atomic. Either a task completes its execution before an energy failure, or Chain re-executes the task from the beginning. Therefore, Chain needs not to preserve the content of volatile memory, as task results are saved onto channels, and the global execution context holds the necessary information to resume the computation.

Alpaca. Alpaca [65] is a task-based forward progress mechanism for mixed-volatile platforms that simplifies data sharing and communication across tasks. Similarly to Chain [25], Alpaca provides a programming abstraction that allows developers to define tasks as C functions. However, unlike Chain [25], Alpaca allows tasks to communicate directly through global variables allocated onto non-volatile memory.

Alpaca memory model divides variables into task-local and task-shared variables. Task-local variables contain private task information and cannot be accessed outside the task where they are created. Instead, task-shared variables contain task results and data shared among tasks.

Similarly to Chain [25], Alpaca ensures program forward progress at task granularity. Either a task completes its execution before an energy failure, or Alpaca re-executes it from the beginning. Consequently, Alpaca maps task-local variables onto volatile main memory, as they contain intermediate task results that need not to be preserved across energy failures. Instead, Alpaca maps task-shared variables onto non-volatile main memory, as they contain data accessible by any task.

Alpaca ensures task atomicity and avoids intermittence bugs [64, 74, 85, 100] by creating task-specific private local copies of task-shared variables. Alpaca allocates these copies onto non-volatile memory. Tasks access task-shared variables through their private local copies, and Alpaca updates task-shared variables on task completion. This ensures that energy failures cannot cause unexpected task-shared values, as tasks cannot mod-

ify task-shared variables unless their execution is complete. We focus on this aspect in Chapter 4.

Alpaca preserves program forward progress with a two-phase commit mechanism, consisting of a pre-commit and commit phase. Alpaca starts the pre-commit phase on task completion and saves the references of task-shared variables' local copies onto a non-volatile commit buffer. Next, Alpaca saves the next task identifier onto a dedicated non-volatile memory location and sets the pre-commit bit, which specifies that the pre-commit phase is completed. Note that an energy failure happening before the completion of the pre-commit phase causes the re-execution of the task from the beginning when energy returns. Alpaca now starts the commit phase. For each reference in the commit buffer, Alpaca updates the corresponding task-shared variable and removes the reference of its private local copy from the commit buffer. Note that an energy failure happening during this phase does not cause the re-execution of the task from the beginning, as Alpaca checks the pre-commit bit on system startup. If the commit bit is set, Alpaca resumes the commit phase from where it stopped. When the commit buffer is empty, Alpaca unsets the pre-commit bit and executes the next task, whose reference was saved into a dedicated non-volatile memory location during the pre-commit phase.

Other systems. The literature provides other system designs that enrich the capabilities of existing task-based forward progress mechanisms.

Coati [88] extends Alpaca [65] programming and execution models, adding support for interrupts. Interrupts may happen at any instant during task execution, potentially violating task atomicity and causing inconsistent program states [88]. Coati requires developers to define the interrupt service routine (ISR) associated with interrupts as two distinct functions: a top function and a bottom function. We can consider the top function as the ISR that executes when the associated interrupt fires. The top function preempts task execution, collects time-sensitive input data, schedule the execution of the bottom function, and resumes the interrupted task. Instead, the bottom function consists of a task that processes the data collected by the top function and only executes after the interrupted task terminates.

InK [103] enriches the semantics of task-based systems by adding support for event-driven and time-sensitive tasks, unlocking new multi-application program designs. InK allows developers to group tasks into task threads, which essentially specify an application. Developers can specify a priority level for each task thread, which InK then uses to schedule task execution. Further, InK allows developers to conditionally enable task threads depending on specific events, such as energy levels, interrupts, and timers. InK

interrupts handling is conceptually similar to the one of Coati [88], where the ISR only gathers time-sensitive data and then schedules the execution of a task that processes such data. Interrupts can preempt task threads at task granularity, whereas single tasks cannot be interrupted.

Coala [76] improves task-based systems efficiency by providing an adaptive task execution model that allows tasks coalescing and downscaling. Coala coalescing strategy postpones state-saving operations and executes them after completing multiple tasks, effectively reducing the overall energy consumption of state-saving operations and tasks transition. Similarly, Coala downscaling strategy uses a timer to save the partial progress of large tasks that are unable to complete due to multiple energy failures, effectively avoiding being stuck at executing non-terminating tasks [26]. We focus on this aspect in Chapter 5. Coala memory handling technique avoids intermittence bugs [64,74,85,100] and ensures that coalescing and downscaling operations do not invalidate tasks' atomicity. Coala allocates tasks state onto a working buffer mapped onto volatile memory. Coala splits the address space of non-volatile memory into (i) a private partition that holds tasks committed data and (ii) a shadow partition that holds non-committed data modified by tasks. Coala takes the same approach as Alpaca [65] to ensure task atomicity and to avoid intermittence bugs [64, 74, 85, 100]. Coala creates copies of variables from the private partition of non-volatile memory onto tasks working buffers. Coala saves the modified copies of the working buffer into the shadow partition of non-volatile memory when each task completes. Then, Coala uses a two-phase commit mechanism to update the private partition from the shadow partition when coalesced tasks commit.

CHAPTER 4

Analysis of Intermittent Program Behaviors

Energy failures may cause battery-less devices to experience unexpected behaviors that lead to results unattainable in a continuous execution [26,64, 74,85,97,100]. The literature analyzes two types of unexpected behaviors specific to intermittent computing: intermittence bugs [64,74,85,97,100] and non-terminating path bugs [26].

Intermittence bugs [64,74,85,97,100] cause an inconsistent program state and lead devices to produce results unattainable in a continuous execution. Section 4.1 provides basic knowledge of intermittence bugs, describing current analysis techniques and ways to avoid their occurrence.

Non-terminating path bugs [26] cause devices to be forever stuck at re-executing the same portion of the program, leading to starvation and preventing program forward progress. Section 4.2 provides basic knowledge of non-terminating path bugs, describing current analysis techniques and ways to prevent their occurrence.

Verifying the absence of intermittence bugs and non-terminating path bugs from programs requires new techniques and tools specifically designed for intermittent computing. Testing techniques for mainstream computation cannot reproduce specific patterns of intermittent executions [24, 26,39,41,74]. Section 4.3 provides basic knowledge of the available testing

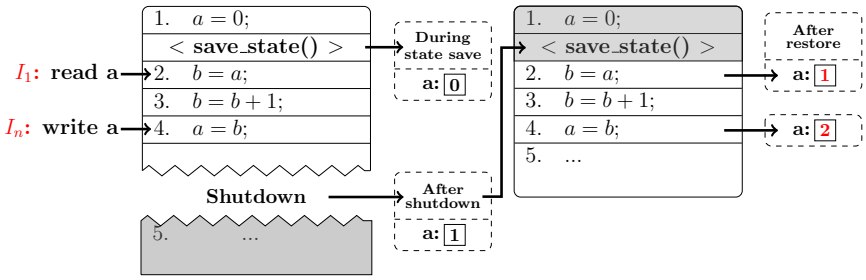


Figure 4.1: *Example of a data access bug [74, 75].* An energy failure causes the re-execution of program portions, causing an unexpected result.

techniques and tools for intermittent programs.

4.1 Intermittence Bugs

This section describes the literature targeting intermittence bugs, consisting of unexpected behaviors specific to mixed-volatile platforms [49–51] and caused by energy failures. Section 4.1.1 describes available analysis of causes and effects of intermittence bugs [64, 74, 85, 100], whereas Section 4.1.2 describes existing techniques to avoid their occurrence [25, 64–66, 74, 100].

4.1.1 Intermittence Bugs Characterization

Mixed-volatile platforms [49–51] simplify persistent state management by allowing developers to directly map portions of program state onto non-volatile memory. However, instructions directly reading or writing non-volatile memory are non-idempotent [100], as their re-execution produces different program states. Consequently, when such instructions are re-executed due to energy failures, the device may produce results different than an equivalent continuous execution of the same program. This unexpected behavior is recognized in the literature as non-volatile memory inconsistency [85] or intermittence bug [64, 74, 100]. Further, the literature [74, 97] classifies intermittence bugs into data access bugs [74], activation record bugs [74], memory map bugs [74], and I/O-dependent idempotence bugs [97]. They all share the same underlying causes but have different consequences.

Data access bugs. Data access bugs [74] are intermittece bugs that affect generic non-volatile memory locations, such as global or local variables, causing unexpected memory results. When a data access bug occurs, the

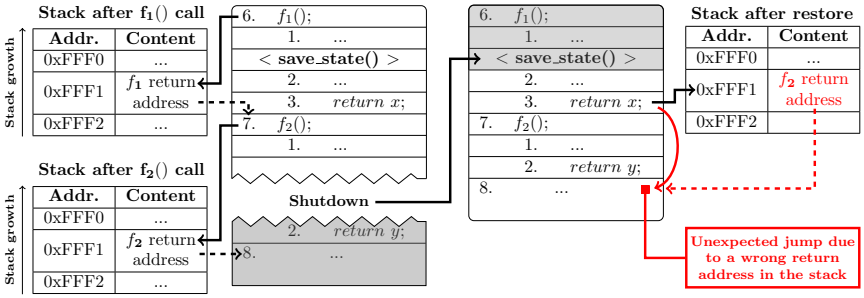


Figure 4.2: *Example of an activation record bug [74, 75].* An energy failure causes the re-execution of portions of a program, causing an unexpected jump.

program produces results different than the ones of an equivalent continuous execution.

Figure 4.1 depicts an example of a data access bug, which we previously describe in Chapter 1.2.2. Variable a is allocated onto non-volatile memory. After the execution of line 1, the device saves a snapshot of its program state onto non-volatile memory. Note that variable a is not included in the saved program state, as it is already persistent. The execution continues, line 4 modifies a, and then an energy failure eventually occurs. When there is sufficient energy, the device restores the saved state and resumes the computation from line 2. The restored program state is inconsistent, as the value of variable a differs from when the program state was saved in the previous power cycle. In fact, being a non-volatile, it retained the value that line 4 produced in the previous power cycle, that is, a *future* instruction compared to where execution resumes after the energy failure [85]. Consequently, the re-execution of lines 2-4 leads to a result that differs from an equivalent continuous execution.

Current literature [85] compares non-volatile memory to a broken time machine that travels back in time while maintaining the effects of changes done in the future. In general, the literature recognizes *write-after-read* (WAR) hazards [64, 74, 97, 100] as the cause of all types of intermittence bugs. WAR hazards consist of sequences of non-volatile memory read and write operations whose re-execution due to energy failures is non-idempotent. In the example of Figure 4.1, such operations are the ones of lines 2 and 4: line 2 reads the non-volatile variable a, and line 4 writes the same non-volatile variable a. The re-execution of lines 2 and 4 due to energy failures is non-idempotent, as each re-execution produces different results.

Activation record bugs. Activation record bugs [74] are particular cases

of data access bugs that may happen whenever the stack segment is non-volatile and affect functions activation record. The activation record of a function is created onto the stack upon function calls. It contains all the information necessary to execute the function, including its parameters, the return address, and local variables. When an activation record bug occurs, a function accesses information from the activation record of a function to be executed in the future, leading to the computation of wrong results, unexpected jumps, or program crashes.

Figure 4.2 depicts an example of an activation record bug. Line 6 calls function f_1, and f_1 activation record is created on the stack. During the execution of f_1, the device saves its state onto non-volatile memory. Note that the saved state does not include the stack, as it is already persistent. f_1 eventually returns and pops its activation record from the stack. Note that this operation updates the stack pointer register and does not delete the stack content. The execution continues from line 7, which calls function f_2. Here f_2 activation record is created on the stack, overwriting the memory cells where the activation record of f_1 was. During the execution of f_2, an energy failure happens. When sufficient energy is available, the device restores the saved state and resumes the computation from line 2, inside the context of the function f_1. Note that the activation record in the stack is the one of f_2, and not the one of f_1. Line 3 executes a return from f_1, reading a wrong return address from the stack. Consequently, the device executes line 8, skipping the execution of line 7.

In general, the effects of activation record bugs depend on the activation record size of involved functions, and may also affect the values of local variables and function parameters [74]. Moreover, in the example of Figure 4.2, if f_2 overwrites f_1 return address with an invalid address due to a larger activation record, the program would crash [74].

Similarly to data access bugs, activation record bugs happen due to WAR hazards involving memory read and write operations sequences that target functions' activation records. In the example of Figure 4.2, the *return* operation of line 3 reads f_1 activation record, and the function call of line 7 writes the same memory cells where f_1 activation record resides.

Memory map bugs. Memory map bugs [74] are intermittence bugs that may happen whenever the heap segment is non-volatile and affect accesses to dynamically-allocated memory blocks. When a memory map bug occurs, the program accesses memory blocks that were dynamically allocated or de-allocated by future operations, leading to memory access failures, program crashes, or memory leaks.

Figure 4.3 depicts an example of a memory map bug. The *malloc* in-

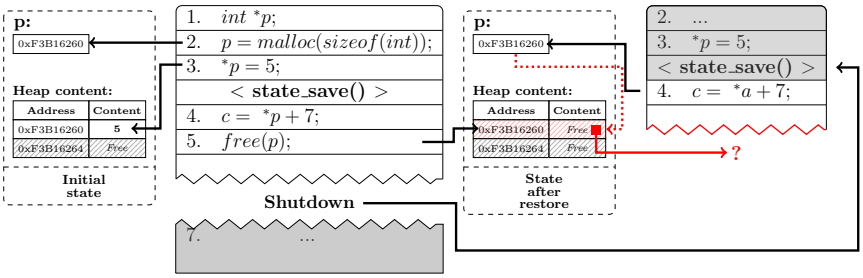


Figure 4.3: *Example of a memory map bug [74, 75].* An energy failure causes the re-execution of portions of a program, causing access to an unavailable memory location.

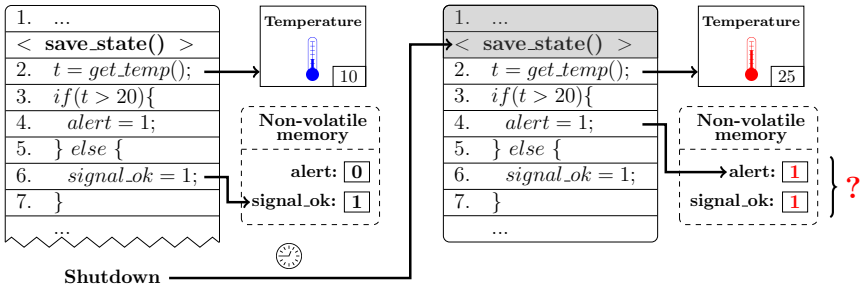


Figure 4.4: *Example of an I/O-dependent idempotence bug [97].* An energy failure causes the re-execution of portions of a program, causing a wrong memory state.

struction of line 2 allocates a new memory block in the heap and saves its address in the pointer p. After the execution of line 3, the device saves its state onto non-volatile memory. Note that the saved state does not include the heap, as it is already persistent. The execution continues, the *free* instruction of line 5 de-allocates the previously-allocated memory block, and then an energy failure happens. When sufficient energy is available, the device restores the saved state and resumes the computation from line 4. Line 4 tries to read the memory block pointed in p. However, this memory block was de-allocated by line 5 during the previous power cycle, and the memory access fails, potentially leading to a program crash.

Similarly to data access bugs and activation record bugs, memory map bugs happen due to WAR hazards involving sequences of operations that access dynamically allocated memory blocks and then alter the block state. In the example of Figure 4.3, line 4 reads a dynamically-allocated memory block that the *free* instruction of line 5 de-allocates. A similar problem happens with instructions that dynamically relocate memory blocks, such as *realloc*. Moreover, multiple re-executions of instructions that dynamically allocate new memory blocks lead to memory leaks.

I/O-dependent idempotence bugs. I/O-dependent idempotence bugs [97] are intermittence bugs that happen whenever external input data controls the execution of memory write operations that target non-volatile memory. When an I/O-dependent idempotence bug occurs, the program state reflects a control flow where multiple branches of the same conditional statement simultaneously execute, leading to unexpected behaviors or wrong results.

Figure 4.4 depicts an example of an I/O-dependent idempotence bug. Variables $alert$ and $signal\_ok$ are non-volatile. The device saves the program state onto non-volatile memory. Note that the saved state does not include $alert$ and $signal\_ok$, as they are already non-volatile. Line 2 measures the environment temperature and sets t to 10. The condition of the *if* statement of line 3 evaluates to *false*, as t is 10, and line 6 executes next, setting $signal\_ok$ to 1. The execution continues until an energy failure occurs. While the device is powered off, the environment temperature rises to $25°C$. When sufficient energy is available, the device restores its state, and the execution resumes from line 2, which measures the environment temperature and sets t to 25. The condition of the *if* statement of line 3 now evaluates to *true*, as t is 25, and line 4 executes next, setting $alert$ to 1. Although the program now entered the *true* branch of the *if* statement of line 3, the memory also has the effects that the execution of the *false* branch produced during the previous power cycle. Consequently, the execution continues with an inconsistent program state, reflecting a case where both branches were taken simultaneously. This may lead to unexpected and unpredictable results that are not possible in a continuous execution of the same program.

Similarly to other intermittence bugs, I/O-dependent idempotence bugs happen due to WAR hazards involving sequences of operations that read external input data and, depending on such data, conditionally write non-volatile memory. In the example of Figure 4.3, line 2 reads external input data and controls the execution of lines 4 and 6, which write non-volatile memory.

4.1.2 Avoiding Intermittence Bugs

The literature identifies WAR hazards [64, 74, 85, 100] as the cause of the intermittence bugs described in Section 4.1.1, as they all share the same underlying read-write patterns [74, 97]. Consequently, avoiding intermittence bugs requires ensuring that programs are free of WAR hazards [74].

In our previous research [74], we point out that a program contains a WAR hazard if it executes a sequence of operations $I_1, ..., I_n$ such that:

1. I_1 reads a non-volatile memory location at address x
2. I_n writes the same non-volatile memory location at address x
3. state-saving operations may not execute in the sequence $I_1, ..., I_n$

In the example of Figure 4.1, I_1 and I_n are the instructions of lines 2 and 4, respectively. We refer to these conditions as *WAR hazard condition*.

Note that this sequence of operations mainly targets data access bugs, but it can be easily adapted to target any other type of intermittence bug. For example, WAR hazards that lead to I/O-dependent idempotence bugs happen if I_1 is an instruction that reads an external input and I_n is a conditionally-executed instruction that writes non-volatile memory. For simplicity, this section refers to this formalization as WAR hazards.

There are several techniques [25, 64–66, 74, 100] to ensure programs are free of WAR hazards. They share similar principles and ensure a consistent program state through specific checkpoint placement strategies [100], versioning of non-volatile memory locations [65], specific memory access patterns [25], or restoring specific portions of non-volatile memory [64,66].We describe next the differences between these approaches.

Checkpoint placement strategy. Ratchet [100] avoids intermittence bugs with its checkpoint placement strategy, which ensures that the execution of the instructions between two checkpoints is idempotent. We recall that Ratchet is a checkpoint-based forward progress mechanism that automatically instruments programs with checkpoints. Ratchet analyzes the program code and identifies all the sequences of instructions that satisfy the WAR hazard condition. For each identified sequence, Ratchet places a checkpoint between instructions I_1 and I_n of the WAR hazard condition, consisting respectively in the instructions that read and write the same non-volatile memory location. Such placement ensures that if I_n writes a non-volatile memory location and an energy failure occurs, the execution resumes after I_1. This removes the WAR hazard and prevents the occurrence of intermittence bugs, as no instruction can read the value modified by I_n during the previous power cycle.

Ratchet checkpoint placement ensures that the instructions between two checkpoints cannot simultaneously read and write the same non-volatile memory location. This makes the sequence of instructions between two checkpoints idempotent, as re-executions due to energy failures always produce the same non-volatile memory state.

Versioning. Alpaca [65] avoids intermittence bugs by creating task-specific private local copies of task-shared variables. We recall that Alpaca is a task-based forward progress mechanism that allows task communication through non-volatile task-shared variables. Alpaca initializes private local

copies of task-shared variables at the beginning of task execution and allocates them onto non-volatile memory. Tasks access task-shared variables through their private local copies, and Alpaca updates task-shared variables only on task completion. This ensures that, if power fails during task execution, task-shared variables retain their initial state, as tasks only read and write their private local copies. Consequently, energy failures cannot cause tasks to access values of task-shared variables produced by partial executions of tasks, as tasks cannot modify task-shared variables unless their execution completes. Moreover, Alpaca's technique ensures that private local copies of task-shared variables are not involved in any WAR hazard. In fact, if a task modifies its private local copy of a task-shared variable and an energy failure occurs, Alpaca reinitializes the private local copy when the execution resumes, and the task cannot access the value produced during the previous power cycle. Finally, the two-phase commit mechanism of Alpaca described in Chapter 3.2 ensures that task-shared variables are atomically updated only after task completion.

Memory access pattern. Chain [25] avoids intermittence bugs by ensuring that each task can exclusively read or write any non-volatile memory location. We recall that Chain is a task-based mechanism that allows tasks to exchange data through channels, consisting of developers-specified non-volatile memory locations. Each channel is shared among two tasks: one task can only read data from the channel, whereas the other can only write data onto the channel. This ensures no WAR hazard is present, as tasks cannot read and write data onto the same non-volatile memory location.

However, tasks may need to read and write data from the same channel. This may introduce WAR hazards. To address this problem, Chain allows tasks to read and write data from the same channel using self-channels. A self-channels is a channel with two buffers: one used as a read-only buffer and the other as a write-only buffer. Chain manages accesses to self-channels as normal channels: a task can only read data from the read-only buffer of the channel and can only write data onto the write-only buffer of the channel. Similarly to normal channels, this ensures no WAR hazard is present, as tasks cannot read and write data onto the same non-volatile memory location. Upon task completion, Chain swaps the read-only buffer with the write-only buffer, ensuring that tasks re-executions due to normal control flow access correct self-channel information.

Restoring non-volatile memory. DINO [64] and Chinchilla [66] avoid intermittence bugs by restoring the content of non-volatile memory when resuming after energy failures.

DINO [64] avoids intermittence bugs by restoring the state of global

variables involved in WAR hazards when resuming after energy failures. We recall that DINO is a task-based forward progress mechanism that allows tasks to communicate through non-volatile global variables and requires developers to define tasks by placing task boundaries in their programs. DINO identifies the global variables involved in WAR hazards for each task using the WAR hazard condition. Then, DINO instruments the task boundary at the beginning of the considered task with instructions that copy the identified global variables onto volatile memory. This ensures that state-save operations also save the state of global variables that task execution modifies. When resuming after energy failures, DINO restores the saved global variables along with the program state. This ensures that if a task alters the value of a global variable and an energy failure occurs before task completion, no instruction can read the modified value, as DINO restores the memory to a consistent state when the computation resumes.

Chinchilla [66] avoids intermittence bugs with an undo-logging technique that logs changes to non-volatile memory and restores it to a consistent state when resuming after energy failures. We recall that Chinchilla is a checkpoint-based forward progress mechanism that maps global variables onto non-volatile memory. Chinchilla uses a log buffer to keep track of all the changes to non-volatile memory since the last checkpoint. When Chinchilla saves a checkpoint, it invalidates the entry of the log buffer. Before each instruction that writes a global variable x, Chinchilla inserts a call to *uLog*, a special function that takes as a parameter the address of the variable x. *uLog* saves the current value and the address of x onto a log buffer allocated onto non-volatile memory. Note that if a valid entry already exists for x, *uLog* skips this operation, as a previous execution of *uLog* already saved the value that x had before any change since the last checkpoint. When Chinchilla resumes the computation after an energy failure, it executes the undo-logging. Chinchilla iterates over the log buffer and restores the value of each global variable saved in it. After restoring a global variable, Chinchilla removes the corresponding entry from the log buffer, ensuring that undo-logging happens only once for each global variable. Undo-logging ensures that changes to global variables are rolled back to a previous and consistent state as if no operation modified them. This ensures that, when devices resume the computation, non-volatile memory is in the same state as when a checkpoint was saved. Consequently, undo-logging prevents intermittence bugs, as no instruction can access results produced by future operations during previous power cycles.

4.2 Non-terminating Path Bugs

Non-terminating path bugs [26] consist in program paths whose instructions consume more energy than the device can buffer and do not include any state-save operation. The execution of such paths requires all the energy stored in the energy buffer and additional energy from the energy source. During the execution of such paths, the device may be unable to harvest sufficient energy, and an energy failure may occur. When the energy buffer stores sufficient energy, the device restarts the computation from the first instruction of the non-terminating path. However, if the device is unable to harvest energy fast enough to sustain the computation until the next state-save operation, it will experience other energy failures. This same process may be repeated indefinitely, leading to starvation: devices are forever stuck at re-executing the same portion of a program, as they cannot reach the next state-save operation before experiencing an energy failure.

This bug can happen with both static checkpoint-based forward progress mechanisms [16,66,86,100] and task-based forward progress mechanisms [25, 44, 52, 64, 65, 68, 76, 88, 103]. The instructions between two checkpoints or inside tasks may contain a non-terminating path.

The presence of non-terminating paths does not necessarily lead to starvation, as energy bursts from the harvesting source may eventually allow the execution of the next state-save operation. However, programs should be free of non-terminating paths, as they may prevent program forward progress despite the use of a forward progress mechanism.

Avoiding non-terminating paths. Identifying non-terminating paths requires modeling the energy consumption of program instructions and identifying the cost of each possible path in each task or each sequence of instructions between two checkpoints [26]. If the energy consumption of a path exceeds the maximum energy that the energy buffer can store, the path is non-terminating.

To remove non-terminating paths, developers need to split the tasks containing non-terminating paths into smaller sub-tasks or place additional checkpoints in the sequences of instructions that contain non-terminating paths [26].

4.3 Tools for Intermittent Computing

This section describes the tools [24, 26, 38, 39, 41, 97] available in the literature to test battery-less device behavior. Section 4.3.1 describes tools [24, 39, 41] that enable real-hardware testing, whereas Section 4.3.2 describes

tools [26, 38, 97] that simulate battery-less devices behavior and analyze intermittent programs.

4.3.1 Real-hardware testing tools

The literature provides very few tools that enable repeatable in-lab experiments on real hardware: EDB [24], EKHO [41], and Shepherd [39]. EDB [24] unlocks devices debugging without interfering with their energy consumption. EKHO [41] allows recording and reproducing the energy characteristics of ambient energy sources. Shepherd [39] enables testing networks of distributed battery-less devices through synchronous recording and reproducing of ambient energy sources across multiple points in space and time.

EDB. Debugging battery-less devices is non-trivial. Debugging operations may alter devices' energy consumption, interfering with the device energy buffer and intermittent behavior, or may be unable to retrieve data before devices shut down due to energy failures.

EDB [24] tackles these problems, providing an energy-interference-free debugging system with energy-oriented debugging capabilities specific to intermittent computing. EDB consists of two components, a hardware debugger and a runtime library.

The hardware debugger is electrically isolated from the device under test (DUT) and connects to the developer workstation. EDB hardware debugger supports two modes of operations, passive and active. In passive mode, EDB passively monitors the DUT, collecting its energy level, I/O events, and program events marked using watchpoints of EDB runtime library. In active mode, EDB allows developers to interact with the DUT by manipulating its energy level, executing specific debugging operations through the operations of EBD runtime library, or by providing an interactive debugging console.

EDB runtime library allows developers to monitor or interact with the DUT at specific points of the program execution using programming primitives. EDB compensates for the energy consumed by the execution of these programming primitives by measuring the energy buffer level and supplying constant energy to the DUT during their execution, ensuring the energy buffer level remains constant. EDB runtime library provides breakpoints, assertions, watchpoints, and energy guards.

EDB supports three types of breakpoints that pause the program execution and open an interactive debugging console on the developer workstation, switching EDB to active mode. Program breakpoints trigger whenever

reached during the program execution; energy breakpoints trigger whenever the energy buffer level drops below a specified threshold; combined breakpoints combine program and energy breakpoints and trigger whenever reached, and the energy buffer level meets a specified condition. Similarly, assertions verify developer-specified conditions and open an interactive debugging console on the developer workstation if they fail.

EDB watchpoints signal specific program events while maintaining EDB in passive mode.

Finally, EDB energy guards indicate sections of programs that EDB needs to run under constant energy to keep the energy buffer level constant.

EKHO. Reproducing energy harvesting sources is non-trivial, as harvested energy has an unpredictable pattern and depends on environmental conditions and device runtime behavior. EKHO [41] tackles this problem and enables repeatable experiments of energy-harvesting scenarios, allowing developers to record and reproduce energy-harvesting sources.

Harvested energy depends on the energy buffer voltage, which affects the energy buffer charge current. EKHO uses I-V curves to express this relationship between the energy buffer voltage and its charge current. An I-V curve captures the variation of the energy buffer charge current (I) at a given supply voltage (V). Note that different programs have different loads, which results in different energy consumptions that occupy different areas of an I-V curve. EKHO represents energy-harvesting sources using I-V surfaces, which capture the variation of I-V curves over time.

EKHO comprises a surface manager, an I-V curve controller, and a front-end device. The surface manager and I-V curve controller run on a workstation, whereas the front-end device is a custom system that provides current and voltage sensing capabilities.

EKHO records I-V surfaces using the I-V curve controller and the surface manager. The I-V curve controller measures an I-V curve by varying a $100k\omega$ potentiometer with a high frequency of up to $1kHz$. The fast variations of the potentiometer resistance simulate multiple program loads, allowing EKHO to capture an entire I-V curve at a specific time instant. The surface manager then aggregates multiple I-V curve measures to create the I-V surface characterizing the energy harvesting source.

EKHO front-end device can reproduce energy-harvesting sources using I-V surfaces as lookup tables. At any instant, EKHO measures the energy buffer voltage, loads the current I-V curve, and outputs the charge current corresponding to the measured energy buffer voltage.

Shepherd. Reproducing the same ambient energy source across multiple distributed battery-less devices is non-trivial, as energy harvesting patterns

change throughout time and depend on the devices' deployment locations. Shepherd [39] tackles this problem and enables synchronized recording and emulation of ambient energy sources across multiple battery-less devices deployed at different locations.

Shepherd assumes an architecture where a DC/DC converter connects the energy harvester to the MCU, allowing them to operate at different voltages. Such decoupling of operating voltages allows converter-based architectures to operate energy harvesters at fixed points independently from the load of the MCU. Therefore, Shepherd represents energy-harvesting sources only using I-V curves, as changes in the load do not affect the harvester voltage. This results in a faster sampling time and a lower collected data size than EKHO [41], which instead assumes no DC/DC converter and must collect entire I-V surfaces by varying the load multiple times.

Shepherd consists of multiple identical nodes attached to each distributed battery-less device we consider. Shepherd nodes have a modular architecture consisting of an observer and multiple capelets. Shepherd's observer consists of a single-board computer that drives an analog front-end device and a DC/DC converter. The analog front-end device comprises an ADC to record I-V data of energy-harvesting sources and a DAC to reproduce it. Shepherd allows developers to reproduce various battery-less devices hardware configurations using three types of capelets, which can be stacked together: (i) the harvester capelet, which contains the energy harvester, (ii) the storage capelet, which contains the energy buffer, and (iii) the sensor node capelet, which contains the sensor node. Note that the harvester capelet connects to Shepherd's observer DC/DC converter, which regulates the voltage between the energy harvester and the sensor node. In contrast, the storage capelet connects in parallel to the sensor node.

Shepherd supports two operating modes: recording and emulation. During recording, Shepherd's observers sample the output voltage and current of the energy harvesters at a sampling frequency of $100kHz$. Each Shepherd observer timestamps the recorded data and sends it to a central database. During emulation, Shepherd's observers emulate the output voltage and current of the energy harvesters by reproducing previously-recorded voltage and current data, setting them as input of the DC/DC converter. When emulating energy sources, Shepherd also collects GPIO and serial data of each node. Shepherd synchronizes each node's clock using GPS and PTP, ensuring that all its nodes target the same time-instant when recording and reproducing energy sources.

4.3.2 Simulation and analysis tools

The literature provides multiple tools [26, 38, 97] that allows developers to simulate battery-less devices behaviour [38], and verify the presence of intermittence bugs [97] and non-terminating path bugs [26].

Siren. SIREN [38] is a simulator of intermittently-powered devices built on top of MSPSim [35], a timing-accurate instruction-level simulator for the MSP430 platform [51], which runs unmodified target platform firmware. SIREN extends MSPSim with support for realistic energy simulations and new debugging capabilities.

SIREN energy simulation relies on (i) EKHO I-V surfaces to simulate the pattern of energy-harvesting sources and (ii) a capacitor model to simulate energy buffers and energy failures. During the simulation of program execution, SIREN simulates the device energy consumption and updates the capacitor voltage. Then, SIREN loads the current I-V curve from the I-V surface of the simulated energy-harvesting source, identifies the capacitor charge current corresponding to the updated capacitor voltage, and recharges the capacitor accordingly.

SIREN enables debugging capabilities through a special C function that developers can insert into their programs, called $siren\_command$, which takes a string specifying a SIREN debugging command as a parameter. SIREN supports two debugging commands: watchpoints and print. Similarly to EDB watchpoints, SIREN watchpoints signal program events to developers. Instead, SIREN print provides $printf$ capabilities, outputting the content of given variables.

IBIS. IBIS [97] is a tool that allows developers to identify I/O-dependent idempotence bugs [97] in their programs. IBIS identifies an I/O-dependent idempotence bug whenever a branch condition is input-dependent and the branches following the condition include divergent updates to non-volatile memory.

IBIS consists of two analysis techniques, IBIS-S and IBIS-D. Both analysis requires developers to annotate variables that contain input-dependent data. IBIS refers to these variables as tainted variables.

IBIS-S relies on static compile-time analysis to identify I/O-dependent idempotence bugs. IBIS-S analyzes a program and propagates the uses of tainted variables. When it identifies a branch condition that includes a tainted variable, it analyzes all the memory write operations included in the branches following the condition. If the write operations in the branches do not target the same non-volatile memory locations, IBIS-S finds an I/O-dependent idempotence bug and signals it to the developer.

IBIS-S analysis is unaware of information available only at runtime, such as non-volatile memory addresses and pointer locations. To overcome this limitation, IBIS-D executes a given program and identifies I/O-dependent idempotence bugs at runtime. IBIS-D instruments the program with bookkeeping information that tracks the propagation of tainted variables, applying a similar analysis strategy to IBIS-S.

IBIS-S and IBIS-D analysis returns a report that includes the variables involved in the I/O-dependent idempotence bugs, the tainted branches, and the involved non-volatile memory locations. IBIS provides a validator that allows developers to evaluate the effects of the I/O-dependent idempotence bugs found by executing programs and simulating energy failures accordingly to IBIS-S or IBIS-D reports.

CleanCut. CleanCut [26] is a tool that allows developers to verify the presence of non-terminating path bugs [26]. CleanCut comprises two components, a checker that verifies the presence of non-terminating path bugs and a placer that instruments programs with checkpoints [64] to prevent non-terminating path bugs.

CleanCut analysis relies on a statistical model of the device energy consumption to estimate the energy consumption of program instructions. Cleancut shows how to build such a model using the debugging capabilities of EDB [24]. Developers need to insert EDB watchpoints inside their programs and measure the level of the energy buffer whenever a watchpoint executes during program execution. CleanCut then uses these measurements to generate a statistical model of the program energy consumption, which then uses to estimate the energy consumption of program paths.

CleanCut checker takes as input (i) a program already instrumented with a forward progress mechanism, (ii) the size of the energy buffer, and (iii) the statistical model of the device energy consumption. CleanCut supports static checkpoint-based mechanisms or task-based mechanisms. Note that CleanCut mainly targets DINO, whose task boundaries can be considered both checkpoints and boundaries for task formation. CleanCut checker considers each possible path in the instructions executed between two state-saving operations and verifies the energy consumption of each path. Whenever a path energy consumption exceeds the maximum level of the energy buffer, CleanCut finds a non-terminating path bug and signals it to the developer.

CleanCut placer uses DINO [64] task boundaries to place state-saving operations and avoid non-terminating path bugs. CleanCut placer analyses the program energy consumption and uses an iterative algorithm to place DINO [64] task boundaries. The iterative algorithm starts analyzing each

program path and estimates its energy consumption. Whenever a program path exceeds the maximum level of the energy buffer, CleanCut placer splits the path in half with respect to the energy consumption of the instructions included in the path. CleanCut placer repeats this step until no path exceeds the maximum level of the energy buffer.

CHAPTER 5

Energy Efficiency

Battery-less devices must operate efficiently to extract the most possible work out of harvested energy. In this chapter, we describe existing strategies to improve devices' energy efficiency. Two key factors affect the energy consumption of battery-less devices: the used forward progress mechanism and the device operating settings.

Forward progress mechanisms periodically save the device state onto a non-volatile memory location to ensure program forward progress across energy failures. As we anticipate in Chapter 1.2.3, this introduces a computation and energy overhead proportional (i) to the frequency of state-save operations, and (ii) to the size of the saved program state.. The literature provides several strategies [7, 11, 12, 16, 25, 54, 55, 59, 64–66, 76, 86] to reduce the frequency and size of state-saving operations. These strategies save only differential updates [7], save the state only when there is no sufficient energy left to continue the computation [11, 12, 16, 54, 66, 86], place the least possible amount of state-saving operations [16] at program points where the state to be saved is minimum [16, 86], or map slices of program state onto non-volatile memory [25, 54, 55, 59, 64–66, 76]. We discuss these strategies in Section 5.1.

Devices operating settings, such as the operating frequency and volt-

age, directly affect the energy consumption. The literature provides several approaches to improve devices' energy consumption by tuning their operating setting. These approaches vary devices duty cycles [83, 94, 96], use maximum power point tracking techniques to maximize harvested energy [10, 90], or dynamically tune devices operating voltage and frequency to ensure optimal performance [5, 9, 36]. We discuss these approaches in Section 5.2.

5.1 Improving Forward Progress Efficiency

This section describes how forward progress mechanisms reduce the energy overhead of state-save operations. Section 5.1.1 describes existing approaches to reduce the frequency of state-save operations, whereas Section 5.1.2 describes existing approaches to reduce the size of the saved program state.

5.1.1 Reducing State-save Operations Frequency

The literature provides several approaches to reduce the frequency of state-save operations. There is a limit in reducing the frequency of state-save operations, as a too aggressive reduction of state-save operations frequency may result in non-terminating path bugs [26], where devices are unable to reach the next state-save operation and are forever stuck at re-executing the same portion of programs. Forward progress techniques save the program state only when the level of the energy buffer drops below a given threshold [11, 12, 16, 54, 86], use specific checkpoint placement strategies to place the minimum possible number of checkpoints inside programs [16], disable checkpoints at runtime [66], or skip saving the state after task completion [76].

Probe-before-save. Mementos [86] and HarvOS [16] postpone the execution of state-save operations, saving the program state only when the energy buffer does not store sufficient energy to continue the computation. As we describe in Chapter 3, they instrument programs with trigger calls that probe the energy buffer level and proceed to save the program state only when it is below a given threshold.

Mementos trigger calls probe the energy buffer voltage through an ADC and save the program state only if the measured voltage falls below a given threshold. However, ADC accesses are expensive both in terms of energy consumption and access latency [50], introducing an energy overhead that may worse performance [16, 50, 73]. Instead, HarvOS trigger calls decide to save the program state by comparing the energy buffer level against an

energy-based threshold, which indicates the energy cost to reach the next trigger call and save the state. This allows HarvOS trigger calls to identify the remaining energy in the energy buffer using software [18, 95] or hardware solutions [29, 81] that introduce very little overhead compared to Mementos ADC probing. Unlike Mementos, HarvOS compile-time analysis computes a threshold specific to each trigger call, providing better control on the execution of state-save operations that lead to better forward progress than Mementos [16].

Compared to forward progress mechanisms that directly execute state-save operations [25,64,65,100], Mementos and HarvOS reduce the number of states save, as not all trigger calls end up saving the program state [16, 86].

Save at specific voltage threshold. Hibernus [11,12] and QuickRecall [54] save the program state only when the energy buffer voltage drops below a given threshold. Although this approach is in principle similar to Mementos [86] and HarvOS [16], Hibernus and QuickRecall rely on hardware interrupts to trigger state-save operations instead of trigger calls statically placed inside programs. This allows Hibernus and QuickRecall to save the program state at any instant during the program execution instead of specific program locations.

As we describe in Chapter 3, Hibernus and QuickRecall use voltage comparators external to the MCU [11, 12, 51, 54] to compare the energy buffer voltage against a pre-defined voltage threshold. Whenever the voltage comparator detects that the energy buffer voltage drops below the given threshold, it triggers an interrupt whose handler saves the program state.

This interrupt-based solution introduces a lower overhead than statically placed trigger calls, as no additional computation is necessary to detect when to save the program state. Further, it allows the device to save the state at the latest possible instant, potentially leading to higher progress in the program execution than solutions with statically placed trigger calls [16, 86]. However, this came at the cost of additional hardware components, which may not be available in the target system and may increase the overall device energy consumption.

Disabling state-save operations. Unlike previous approaches that save the state depending on the energy buffer level [11,12,16,54,86], Chinchilla [66] uses a timer to decide when to save the program state.

Chinchilla instruments programs with calls to a checkpoint routine at the end of every basic block of the program. At runtime, Chinchilla considers state-save operations disabled by default and relies on a timer to dynamically enable them. When state-save operations are disabled and Chinchilla

executes a call to a checkpoint routine, the device continues the computation. When the timer fires, Chinchilla enables state-save operations. Then, when Chinchilla executes a checkpoint, it saves the program state, disables state-save operations, and resets the timer.

Similarly to forward progress mechanisms that save the state depending on the energy buffer level [11,12,16,54,86], Chinchilla strategy reduces the frequency of state-save operations, providing better forward progress [66] than mechanisms that instrument programs with direct calls to state-save operations [25, 64, 65, 100].

Checkpoints placement strategy. HarvOS [16] placement strategy minimizes the number of trigger calls placed inside programs. Combined with the probe-before-save approach, this allows HarvOS to reduce the frequency of state-save operations.

The key behind HarvOS placement is the C_{use} parameter, which specifies the maximum number of clock cycles a device can execute within a single power cycle. C_{use} refers only to program instructions and does not include the cost of state-save and stat-restore operations: HarvOS computes C_{use} considering as available energy the one stored in a fully charged energy buffer, at which it removes the cost of executing one state-restore and one state-save. Note that C_{use} also represents the maximum distance between two state-save operations. Otherwise, a non-terminating path bug arises [26].

HarvOS compile-time analysis splits the program into chunks that execute $\frac{C_{use}}{2}$ instructions and place one trigger call inside each chunk, where the allocated program state is the lowest. This ensures that the distance between state-save operations does not exceed C_{use} while ensuring that the number of trigger calls placed inside programs is minimum.

HarvOS placement reduces the number of trigger calls placed in programs, resulting in a lower energy overhead than more frequent placements, such as the one of Mementos [86] or Ratchet [100].

Task coalescing. Task-based forward progress mechanisms [25, 64, 65] save the program state on task completion as if state-save operations are placed at the end of each task. To reduce the frequency of state-save operations in task-based mechanisms, Coala [76] coalesces the execution of tasks, skipping state-save operations at the end of single tasks and saving the state only after the completion of a group of tasks. This approach shares a similar principle with the one of Chinchilla [66]: Coala considers state-save operations at the end of tasks to be initially disabled and dynamically enables them depending on the number of executed tasks.

On startup, Coala sets the initial number of coalesced tasks, that is, the

number of tasks to execute before saving the program state. Whenever Coala saves the program state, it also decrements the number of coalesced tasks. The rationale behind such decrement is that the more tasks execute, the more likely an energy failure will occur. Note that Coala supports various strategies to set and decrement the number of coalesced tasks [76].

As we previously mention, a too-aggressive reduction of state-save operations frequency may result in non-terminating path bugs [26]. To void this situation, Coala keeps track of the number of tasks completed during a power cycle and uses such a number to update the number of coalesced tasks on startup. This ensures that Coala does not set the coalesced tasks parameter too high and that such a parameter will eventually converge to a number of tasks that the device is able to execute within a single power cycle. Moreover, Coala detects if a task cannot complete within a single power cycle, verifying if multiple energy failures occur with no task completed in previous power cycles. Whenever this situation happens, Coala applies a checkpoint-like strategy: Coala sets a timer whose expiration saves the task intermediate results, allowing the failing task to complete over multiple power cycles. The combination of these two strategies allows Coala to always ensure program forward progress across energy failures [26, 76].

The coalescing strategy of Coala results in a lower number of executed state-save operations than task-based mechanisms that save the program state on task completion [65, 76].

5.1.2 Reducing Saved State Size

The literature provides several approaches to reduce the size of the program state that needs to be saved onto non-volatile memory. Forward progress techniques execute state-save operations where the program state is minimum [16, 25, 65, 76, 86], use differential state-save operations that update the saved state with only modified portions of the program state [7, 59], or directly allocate slices of main memory onto non-volatile memory [54, 55, 66, 100].

Executing State-save Operations at Specific Locations

Various forward progress techniques [16, 25, 65, 76, 86] execute state-save operations at points where the program state is minimum.

Mementos. Mementos [86] supports two placement strategies, loop-latch and function-return. The loop-latch placement strategy inserts a trigger call at the back edge of each loop, ensuring that state-save operations execute

at the end of loop iterations. This strategy allows state-save operations to ignore temporary variables used in single loop iterations, as they are iteration-specific and need not to be preserved [73]. Note that Mementos does not take full advantage of this strategy, as it saves the entire program state [73, 86]. The function-return placement strategy inserts a trigger call after each function calls inside the program, ensuring that state-save operations only execute after functions return. This reduces the size of the program state to be saved: when a function returns, it pops its stack frame from the stack, reducing the size of the allocated main memory.

HarvOS. HarvOS [16] ensure state-save operations execute at points where the allocated memory size is minimum. As we describe in Section 5.1.1, HarvOS splits the program into chunks and inserts a trigger call inside each chunk. When doing so, HarvOS analyses the memory footprint of each basic block in the chunk and inserts a trigger call at the end of the basic block whose allocated memory is minimum. This ensures that state-save operations execute where the size of the program state is the minimum possible.

Tasks. Task-based forward progress mechanisms [25, 65, 76] save the program state only after task completion. Tasks are atomic by design [25, 65], which means that tasks are either completed in a single power cycle or are entirely re-executed after energy failures. Therefore, tasks' intermediate results, such as tasks' local variables, need not to be preserved across energy failures, and state-save operations need not to save them. Consequently, the size of the program state saved by state-save operations of task-based mechanisms is always the minimum possible, as they need only to save executed task results and a reference to the next task to execute [25, 65, 76].

Differential Updates of Saved State

The literature provides two systems that save differential updates of the program state, reducing the saved state size: DICE [7] and TICS [59].

DICE. DICE [7] is a technique that uses differential updates to save only the portion of the program state that was modified since the latest state-save operation. DICE instruments programs to track write operations to main memory and works on top of existing checkpoint-based forward progress mechanisms. When checkpoints execute, they save only the slices of main memory modified since the latest checkpoint, overwriting the corresponding non-volatile memory location that holds the saved program state. DICE relies on three elements to track changes to main memory: a *stack tracker*, a $record()$ function, and a $record\_p()$ function.

The stack tracker allows DICE to identify the stack portion that was

modified since the latest checkpoint. We recall that the stack grows from high to low memory addresses. Moreover, programs use two special registers to track the stack growth: the stack pointer, which contains a reference to the first free memory cell on top of the stack, and the base pointer, which contains a reference to the first memory cell of the currently active stack frame, that is, the base of the stack frame used by the executing function. Function calls set the base pointer equal to the stack pointer, whereas function returns revert the base pointer to its previous value. The memory region between the base pointer and the stack pointer indicates the active stack frame.

DICE sets the stack tracker equal to the base pointer whenever a checkpoint executes or after startup. Similarly, whenever a function returns and increases the base pointer to a value higher than the stack tracker, DICE updates the stack tracker to the base pointer's new value. This allows the stack tracker to track the base of the modified portion of the stack. When a checkpoint executes, DICE saves the stack portion between the stack tracker and the stack pointer, overwriting the corresponding non-volatile memory location that holds the saved program state.

DICE tracks change to global variables using a *modification record*, a data structure that tracks the global variables modified since the latest checkpoint. DICE updates the modification record using a $record()$ function, which takes as inputs the address and size of global variables and inserts them into the modification record. DICE pre-compiler inserts calls to $record()$ before each instruction that modifies a global variable. When checkpoints execute, they save the global variables inside the modification record, overwriting the corresponding non-volatile memory location that holds the saved program state.

DICE handles memory accesses through pointers using a $record\_p()$ function, which takes as inputs the address x and size of the memory location referenced by a pointer, and verifies x location. If x is the address of a global variable, $record\_p()$ follows the logic of $record()$. Otherwise, x targets a memory cell in the stack, and $record\_p()$ verifies where x resides. If x is lower than the stack tracker, checkpoints operations already save x and $record\_p()$ returns, as x resides between the stack tracker and the stack pointer. Otherwise, $record\_p()$ updates the stack tracker to x, ensuring that checkpoints will save its content. DICE pre-compiler inserts calls to $record\_p()$ before any instruction that modifies a memory location by reference.

TICS. TICS [59] is a checkpoint-based forward progress mechanism that reduces the size of the program saved state using a differential strategy

similar to DICE [7]. TICS partitions the stack into fixed-size segments and identifies the segments used as a working stack. TICS compile-time analysis instruments the program to support stack segmentation and to track the working stack segments. Upon function calls, TICS allocates the function frame into the first free stack segment and considers it as working stack. When a checkpoint executes, TICS achieves differential state saves by saving only the stack segments of the working stack onto a dedicated non-volatile memory location, called segment checkpoint.

Unlike DICE [7], TICS also achieves differential state restoration. TICS maps main memory onto non-volatile memory. This enables TICS to restore only the stack segments saved in the segment checkpoint instead of the entire stack content.

Non-volatile Main Memory Allocation

The literature provides several forward progress techniques [25, 54, 55, 65, 66, 100] that exploit mixed-volatile platforms [49, 50] to directly map slices of main memory onto non-volatile memory. State-save operations need only to save volatile main memory, as slices allocated onto non-volatile memory are already persistent. This reduces the size of the saved program state at the expense of an increased program energy consumption and intermittence bugs [64, 74, 85, 100], as the computation now directly accesses the slower and less energy-efficient [49, 50, 72, 73] non-voltile memory. We describe next how existing forward progress techniques use mixed-volatile platforms to reduce the size of the saved program state.

Task-based mechanisms. Task-based forward progress mechanisms [25, 65, 76] require developers to partition programs into tasks, which are atomic by design [25, 65, 76]. As we describe in Chapter 3.2, tasks are either completed in a single power cycle or re-executed from the beginning. Therefore, state-save operations of task-based forward progress mechanisms need only to save task results and shared variables, as they hold results that other tasks may need to access. Instead, tasks' intermediate results and tasks' local variables need not to be preserved across energy failures, as tasks re-execute from the beginning after energy failures and recompute such data.

Task-based forward progress mechanisms map tasks' intermediate results and local variables onto volatile memory. In contrast, variables holding task results and variables shared among tasks are mapped onto non-volatile memory. This memory allocation minimizes the size of the saved program state: state-save operations need only to save the reference to the next task to execute, as tasks result are non-volatile and already persistent.

Further, this mapping has a minimal impact on the program energy consumption due to non-volatile memory accesses. Tasks access non-volatile memory only to write their results or to read shared variables.

Checkpoints-based mechanisms. Checkpoint-based forward progress mechanisms [11, 12, 16, 54, 66, 86, 100] do not take advantage of developers-partitioned tasks and need to preserve the content of volatile main memory, the register file, and special registers, such as the program counter and the stack pointer.

Ratchet [100] maps the entire content of main memory onto non-volatile memory, allowing checkpoints to save only the register file and special registers, as main memory is already non-volatile. However, using a non-volatile main memory introduces intermittence bugs, which Ratchet avoids by frequently saving the program state. As a result, the high frequency of state-save operations, in combination with the increased program energy consumption due to non-volatile main memory, lead to poor performance [55, 66].

Similarly to Ratchet [100], QuickRecall [54] maps the entire content of main memory onto non-volatile memory, allowing checkpoints to save only the register file and special registers. Unlike Ratchet, QuickRecall needs not to save the program state frequently to avoid intermittence bugs. As we describe in Chapter 3, QuickRecall uses hardware interrupts to save the program's volatile state when the level of the energy buffer drops below a pre-defined threshold. QuickRecall avoids intermittence bugs by pausing the program computation after saving the program state and waiting for new harvested energy. Despite improving system performance due to a lower checkpoint frequency than Ratchet, QuickRecall uses a non-volatile main memory that increases programs energy consumption, lowering the system performance [55].

Chinchilla [66] uses an approach similar to task-based mechanisms [25, 65]. As we describe in Chapter 3, Chinchilla maps onto volatile memory all the memory locations whose accesses do not cross any checkpoint. Similarly to intermediate task results, these memory locations hold intermediate results computed in the instructions between two subsequent checkpoints and not accessed elsewhere. Chinchilla mapping demonstrates better performance than an entirely non-volatile main memory [66]. Further, as we describe in Chapter 4.1, Chinchilla efficiently avoids intermittence bugs using an undo-logging technique that introduces a lower overhead than Ratchet frequent checkpoints placement [66, 100].

Optimal Program Sections Mapping. Jayakumar et al. [55] propose a system that identifies the optimal memory mapping of program segments

at the granularity of individual functions. They consider the program code (*.text* segment), global variables (*.data* segment), and stack frames (*.stack* segment).

The system has a warm-up phase, where it tries every possible memory configuration for every function and saves the most efficient configuration onto a lookup table that associates to each function a memory configuration and its energy consumption. A memory configuration is an ordered triplet $\{text, data, stack\}$, where each element indicates the allocation of the corresponding program section. S stands for SRAM and indicates that the segment is allocated onto volatile memory; F stands for FRAM and indicates that the segment is allocated onto non-volatile memory. Note that every function has a default configuration of $\{F, F, F\}$, where every segment is allocated onto non-volatile memory.

Before executing a function f, the system selects a memory configuration c that was never applied to the function f. Then, it measures the initial voltage V_i, applies the memory configuration c by copying the necessary segments onto volatile memory, and executes the function f. Once the function terminates, the system saves the volatile state onto non-volatile memory, measures the final voltage V_f, and calculates the configuration c energy consumption as $V_i^2 - V_f^2$. The system updates f record in the lookup table if c energy consumption is lower than one of the configurations already present in the lookup table. The system repeats the previous operations for every function and memory configuration, populating the lookup table for every function.

Note that the system discards the configuration c if an energy failure occurs during the execution of the function f, as f is unable to complete it within a single power cycle using the configuration c. Functions that fail to execute with any configuration fall back to a configuration where all the segments are allocated onto non-volatile memory, that is, $\{F, F, F\}$.

When the warm-up phase completes, the system starts the normal program execution. Before executing a function, the system loads the optimal memory configuration c and the corresponding energy consumption from the lookup table. Then, it measures the energy buffer voltage, and if it stores sufficient energy to execute the function, the system applies the memory configuration c and executes the function. Otherwise, the system shuts down and waits for new harvested energy. Note that this execution strategy ensures programs are free of intermittence bugs, as functions either complete their execution or stops and wait for new harvested energy after saving the program state.

For functions whose configuration is $\{F, F, F\}$, the system falls back

to a mode similar to QuickRecall [54]. The system configures a hardware interrupt that triggers whenever the energy buffer voltage drops below a pre-defined threshold and save the program state, consisting of the register file and special registers.

This system identifies the optimal mapping of program sections for each function, increasing programs' efficiency. When the mapping allows functions to complete within a single power cycle, the system demonstrates up to a 20% lower energy consumption and up to a $2x$ faster execution time than QuickRecall [54]. Otherwise, it follows the same operating mode of QuickRecall [54], providing similar performance.

5.2 Device Operating Setting

This section describes various approaches that improve battery-less devices' efficiency by tuning their operating settings. We focus on techniques that introduce relevant concepts necessary to understand the contributions of the PhD. Most of the available techniques [9, 10, 36, 83, 94] tune devices operating settings to reach energy-neutrality or power-neutrality, where devices operate perpetually, without any power interruption, using only the energy harvested from the environment. Energy-neutral systems tune their operating settings to ensure their energy consumption matches harvested energy at any time instant, while power-neutral systems match harvested energy over a time period. Section 5.2.1 describes techniques that tune devices duty cycles whereas Section 5.2.2 describes dynamic voltage and frequency scaling techniques.

5.2.1 Duty Cycling

The literature provides various techniques that tune devices' duty cycle to ensure efficient operations [11, 12, 54, 55, 96].

Kinetisee [96] provides a system design of a battery-less wearable camera that operates perpetually, harvesting kinetic energy and varying the MCU duty cycle. When the device does not collect images, Kinetisee puts the MCU into a low-power mode to save energy.

Similarly, dynamic checkpoint-based forward progress mechanisms [11, 12, 54] enter a low-power mode after saving the program state due to a low energy buffer and wait for new harvested energy. This allows devices to avoid executing state-restore operations if they are able to harvest sufficient energy while waiting in low-power mode.

Jayakumar et al. [55] propose a system design that verifies if the energy buffer stores sufficient energy to execute a computation unit. Whenever this

condition is not verified, the system enters a low-power mode and waits until the energy buffer stores sufficient energy to execute the computation unit.

Other techniques [56, 83, 94] improve systems throughput and ensure devices achieve energy-neutrality by tuning sensors sampling rate [83] and data transmit rate [94]. However, they do not target battery-less devices and require batteries to operate.

5.2.2 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) techniques tune the device's operating voltage and frequency to make it operate in the most efficient condition. We recall that higher operating clock frequencies have a higher power consumption but also demonstrate a higher efficiency [48–50], as they have a faster execution speed and a lower energy consumption per clock cycle than lower operating clock frequencies. Further, devices' energy consumption directly depends on their operating voltage: the lower the operating voltage, the lower the energy consumption. We describe next the DVFS techniques applied to battery-less devices.

DFS for Power Neutrality. Balsamo et al. [9] propose a system design that uses dynamic frequency scaling (DFS) to achieve power-neutrality in battery-less MCUs. The proposed system runs a DFS algorithm that keeps the level of the energy buffer constant by dynamically adjusting the operating frequency whenever the energy buffer level changes.

The DFS control algorithm uses two voltage thresholds to decide when to decrease or increase the MCU operating frequency, V_{dec} and V_{inc}, respectively. These thresholds are fixed at compile-time and depend on capacitor size and current draws of available operating clock frequencies. When V_{cap} falls below V_{dec}, the system raises a hardware interrupt, and the DFS algorithm sets a timer that expires after a pre-defined number of clock cycles. When the timer expires, the DFS algorithm decreases the operating clock frequency. Then, if V_{cap} is still below V_{dec}, the DFS algorithm resets the timer and repeats these last steps until either V_{cap} exceeds V_{dec} or the MCU is at the lowest operating frequency. The DFS algorithm follows a similar behavior for the case when V_{cap} exceeds V_{inc}, increasing the operating frequency until V_{cap} no longer exceeds V_{inc}.

Compared to a configuration that uses a static clock frequency, this system design extends the computation achieved in a single power cycle by up to 88% [9] and shows a 21% faster execution time.

DVFS for Power Neutrality. Balsamo et al. [36] propose a similar sys-

tem design for multi-core processors with big and LITTLE cores, which adapts its instantaneous power consumption to match instantaneous harvested power. The proposed system achieves power neutrality using DFS to scale the system operating frequency and core hot-plugging to alter the number of active CPU cores. DFS allows the system to react to micro-variations of harvested energy, whereas core hot-plugging allows the system to react to macro-variations of harvested energy.

Similarly to their power-neutral system design for MCUs [9], the system detects changes to the energy buffer voltage V_{cap} using two voltage thresholds, V_{high} and V_{low}, which are equally distant from V_{cap}. Note that $V_{high} > V_{cap} > V_{low}$. Whenever V_{cap} falls below V_{low} or raises above V_{high}, the system fires a hardware interrupt that runs the DFS algorithm first and the core hot-plugging algorithm next.

The DFS algorithm uses a linear control similar to the authors' previous system design [36]. When $V_{cap} < V_{low}$ ($V_{cap} > V_{high}$), the DFS algorithm decreases (increases) the operating clock frequency and then decreases (increases) V_{high} and V_{low} by a parameter V_q.

The core hot-plugging algorithm uses a derivative control that compares the V_{cap} gradient variation against uses two gradient thresholds, α and β. Whenever V_{cap} gradient exceeds α (β), the core hot-plugging algorithm adjusts the number of active big (LITTLE) cores accordingly to the event that triggered the interrupt.

Compared to other static approaches, this system design allows devices to execute up to 69% more instructions in a single power cycle.

DVFS and MPPT. Balsamo et al. [9, 36] use their system designs to conceive Momentum [10], a system that uses software-based maximum power point tracking to ensure devices operate at the maximum power point of the energy harvesting source, further improving system efficiency and performance. Momentum identifies the voltage V_{opt} that maximizes harvested power, that is, the maximum power point of the energy harvesting source. Momentum sets V_{low} and V_{high} of previous system designs [9, 36] to be equally distant from V_{opt}. Similar to the previous system designs [9, 36], the system adjusts its operating frequency and the active number of cores to achieve power-neutrality while maintaining V_{opt} as operating voltage.

D2VFS. D2VFS [5] is a system design for battery-less devices that enable dynamic voltage and frequency scaling in ultra-low-power MCUs without dedicated hardware support for DVFS operations. D2VFS targets the MSP430-G2553 [48], an ultra-low-power MCU from Texas Instruments. D2VFS considers four discrete system configurations, consisting of four factory-calibrated clock frequencies and their minimum operating voltage:

$1MHz$ at $1.8V$, $8MHz$ at $2.2V$, $12Mhz$ at $2.7V$, and $16MHz$ at $3.3V$. High operating clock frequencies have a higher efficiency but a reduced operating voltage range. In contrast, lower operating clock frequencies have higher energy consumption but a broader operating voltage range.

During runtime, D2VFS sets the MCU at the maximum possible operating clock frequency available at the current level of the energy buffer while keeping the MCU operating voltage equal to the minimum possible operating voltage of the set frequency. This ensures that the MCU always operates using the most efficient configuration.

When the energy buffer voltage drops below the minimum operating voltage of the current operating frequency, that is, a changepoint, D2VFS reduces the operating clock frequency and sets the MCU operating voltage equal to the minimum operating voltage supported by set frequency, extending the operations executed in a power cycle. D2VFS behaves similarly when the energy buffer voltage increases above the minimum operating voltage of a higher operating clock frequency. However, in such a case, D2VFS offsets the increase of the operating clock frequency by one changepoint to avoid instabilities around changepoints.

D2VFS consists of a hardware-software co-design. D2VFS hardware components consist of (i) a circuit that tracks changes in the energy buffer and fires an interrupt whenever a new changepoint is reached, and (ii) a step-down voltage regulator placed between the energy buffer and the MCU, which allows the regulation of the MCU operating voltage. Note that D2VFS tracks change in the energy buffer with a dedicated circuit to keep the overhead of periodic probing of the energy buffer level at a minimum through an ADC. D2VFS software components react to interrupts and adjust the MCU operating configuration accordingly.

Compared to static frequency configurations, D2VFS reduces the workload completion time by up to 300% and requires a smaller capacitor size to achieve comparable performance.

Part II

Contribution

CHAPTER 6

Deployment of Battery-less Devices

This chapter describes our contribution to battery-less devices deployments [4]. We eventually achieve the first multi-year deployment of battery-less devices that sense the structural and environmental conditions of an underground archaeological site [40, 99] in Rome (Italy). Structural engineers and archaeologists need to monitor the environmental and structural conditions of the site, requiring periodic measurements of ambient temperature, humidity, and vibrations. We published a paper summarizing the experiences and lessons learned from this deployment at the ACM Conference on Embedded Networked Sensor Systems (SenSys 2020) [4]. The paper is attached in Chapter 12.

Deployment challenges. As we describe in Chapter 2, the literature is rich in experiences of deployments for battery-powered devices. However, the target site has unique conditions that pose severe limitations to the use of battery-powered devices. Access to the site is regulated, must be booked in advance, and must be kept as minimum as possible to avoid deteriorating the site conditions. This increases the difficulty of battery replacement logistics.

Further, the site lacks high-power ambient energy sources, such as solar energy, and only thermal and kinetic energy are available, preventing

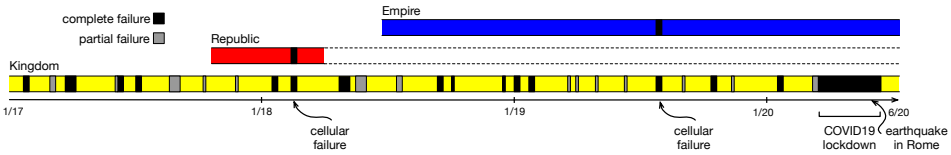


Figure 6.1: *Time evolution of the deployment of our three design iterations [4].*

on-site battery recharge during device operations. This also poses severe limitations to the possible use of battery-less devices, as ambient energy is severely limited and may not suffice to provide periodic and reliable measures of environmental conditions. As we describe in Chapter 2, the literature lacks experiences of deployments for battery-less devices that are long-running, meet end-user requirements, and do not use high-power solar energy. Note that, to the best of our knowledge, the duration of the longest deployment of battery-less devices that does not rely on solar energy is three months [22].

Our deployment. To address end-user requirements, we design two types of sensors to measure environmental and structural conditions: T/H sensors and I/A sensors. T/H sensors periodically record environmental temperature and humidity, whereas I/A sensors also record vibrations through an inclinometer and accelerometer. We deploy 18 T/H sensors and 6 I/A sensors at specific site locations.

Our deployment undergoes three different system design iterations, as Figure 6.1 shows, which we name `KINGDOM`, `REPUBLIC`, and `EMPIRE`. Each iteration aims at improving system reliability and uptime, using the experience from previous system designs. In the remainder of this chapter, we report on the main aspects of each design iteration, whereas a more detailed analysis is available in the published paper [4].

First design iteration. `KINGDOM` is our first design iteration and uses battery-powered sensors. We ensure that T/H sensors and I/A sensors efficiently manage the energy available in their batteries. T/H sensors activate every 20 minutes to sense the environment, and every hour send a radio signal to a gateway containing the average and standard deviation of these measurements. Instead, I/A sensors activate every minute to collect vibration data, and every hour send a radio signal to a gateway containing compressed information of these measurements.

Despite the efforts to ensure efficient battery management, `KINGDOM` experiences several failures due to battery depletion and undergoes long downtime periods waiting for battery replacements, leading to a 71% up-

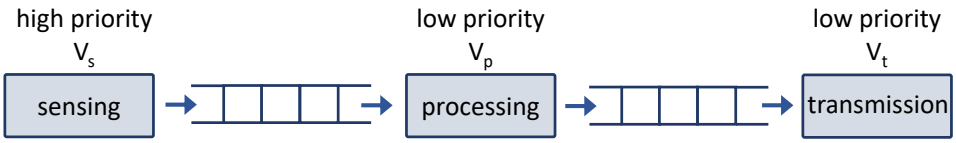


Figure 6.2: *Task-based software design of T/H and I/A sensors in* `EMPIRE` *[4].*

time. Therefore, we devise `REPUBLIC`, a new system design that uses battery-less sensors. Note that we decide to keep `KINGDOM` active to collect baseline data for comparing the performance of battery-less devices against battery-powered ones.

Second design iteration. `REPUBLIC` is our second design iteration, where we swap batteries with an energy harvester and energy buffers. T/H sensors use thermoelectric generators to harvest thermal energy from the temperature difference between air and ceiling, whereas I/A sensors use a piezoelectric transducer to harvest kinetic energy from vibrations. T/H and I/A sensors use a $20\mu F$ capacitor as an energy buffer, which we carefully size to achieve an optimal balance between charging time and available energy. We rely on the same code base of `KINGDOM` and ensure program forward progress across energy failures by instrumenting sensors programs with HarvOS [16].

The severely limited energy supply from ambient energy sources causes `REPUBLIC` sensors to collect 22% of the data that `KINGDOM` sensors collect. Despite such lower data yield compared to `KINGDOM`, `REPUBLIC` provides sufficient information to analyze ambient temperature and humidity. However, `REPUBLIC` fails to capture relevant vibration events due to insufficient energy while the events are occurring. `REPUBLIC` I/A sensors periodically activate to randomly sense vibration data instead of waiting for events of interest. This results in a waste of energy that makes `REPUBLIC` I/A sensors unable to provide relevant information to analyze the site's structural integrity.

Third design iteration. `REPUBLIC` poor performance is a consequence of using design techniques that (i) are conceived for systems with a stable power source, and (ii) assume there is always sufficient energy to capture events of interest. To overcome these limitations, we design `EMPIRE`, a new system design iteration where we co-design sensors' software and hardware to work with the unstable energy sources of the archaeological site.

We design T/H and I/A sensing programs using a task-based approach [25, 64, 65], and we divide tasks into sensing tasks, local processing tasks, and

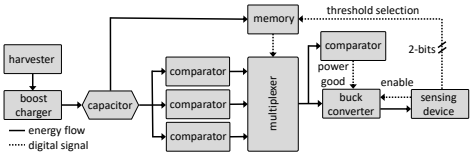


Figure 6.3: *Hardware design of T/H sensors in* EMPIRE *[4].*

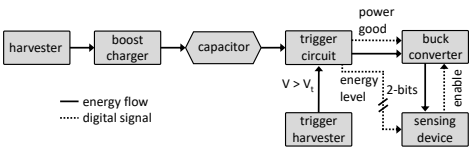


Figure 6.4: *Hardware design of I/A sensors in* EMPIRE *[4].*

data transmission tasks. Figure 6.2 decpits the control flow of EMPIRE sensors. Sensing tasks have the highest priority, as they collect important environmental information and must execute periodically.

Figure 6.3 depicts the key aspects of the hardware design of T/H sensors. We design T/H sensors hardware with a dedicated circuit that enables software-driven system shutdown and turns on the MCU when the energy buffer level exceeds a software-selectable activation threshold. T/H sensors support three activation thresholds: for the execution of sensing tasks, one for the execution of sensing and local processing tasks, and one for the execution of sensing, local processing, and data transmission. At runtime, the MCU decides the tasks to execute in the next power cycle by selecting the corresponding activation threshold. This ensures that tasks are always able to complete, as they execute only when sufficient energy is available.

Figure 6.4 depicts the key aspects of the hardware design of I/A sensors. We design I/A sensors hardware with an additional piezoelectric transducer that turns on the MCU only when vibrations occur. This ensures that the MCU does not waste energy randomly sensing vibration data and instead powers on only when an event of interest occurs. Further, I/A sensors hardware features a 2-bit line that, on startup, signals the MCU with the energy buffer level. After executing the sensing task, the MCU decides the tasks to execute next using such information.

Key results. Similarly to REPUBLIC, EMPIRE sensors collect up to 30% of the data that KINGDOM sensors collect. However, EMPIRE design provides better energy management and ensures that events of interest are recorded.

Figure 6.5 shows an example of vibrations spectral density of the data sensed by I/A sensors of KINGDOM and EMPIRE. Despite a lower number of collected samples, EMPIRE data provide similar quality to the one of KINGDOM, allowing engineers to analyze the structural integrity of the archaeological site. Moreover, the information provided by EMPIRE I/A sensors was sufficient to estimate the magnitude of an earthquake that hit north of Rome on May 11th, 2020, at 3.03 AM UTC. Using the data sensed by EMPIRE I/A sensors, for which Figure 6.6 shows the spectral density,

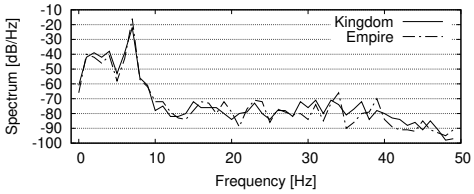


Figure 6.5: *Comparison of vibrations spectral density at I/A sensors of* KINGDOM *and* EMPIRE *[4].*

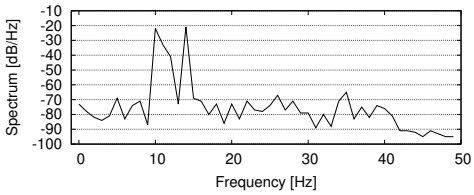


Figure 6.6: *Vibrations spectral density of an earthquake captured by* EMPIRE *[4].*

we calculated a Richter magnitude of 3.14 using existing computational methods [58]. Our estimate is close to the official report of a Richter magnitude in the interval $(3.2, 3.7)$ from the Italian Institute of Geophysics [32], which uses several professional seismographs deployed around Rome.

EMPIRE is still running without any maintenance since June 2018. Instead, KINGDOM is deployed since January 2017 and the overall maintenance efforts required to replace KINGDOM batteries were double the development effort of EMPIRE.

A more detailed analysis of the three deployed systems is available in the published paper [4]. These results demonstrate that battery-less sensors are a viable alternative to battery-powered ones.

CHAPTER 7

Testing and Analyzing Intermittent Programs

This chapter describes our contribution to intermittent program consistency and testing [69, 75]. We provide an in-depth analysis of the unexpected behaviors characterizing battery-less devices, which we call *intermittence anomalies*. We extend the concepts of intermittence bugs and WAR hazards [64,74,85,97,100], and we identify new types of intermittence anomalies previously overlooked by current literature that may happen whenever battery-less devices interact with the environment. Next, we devise a set of techniques to test intermittent execution, and we implement them into `ScEpTIC`, a tool that enables developers to test their intermittent programs. We published this work at the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021) [75] and we release `ScEpTIC` as an open-source project [69].

Open problems. As we describe in Chapter 4, energy failures cause battery-less devices to re-execute portions of programs, potentially leading to the computation of unexpected results unattainable in a continuous execution, which we recognize as *intermittence anomalies*. The literature recognizes only intermittence anomalies happening in non-volatile memory locations

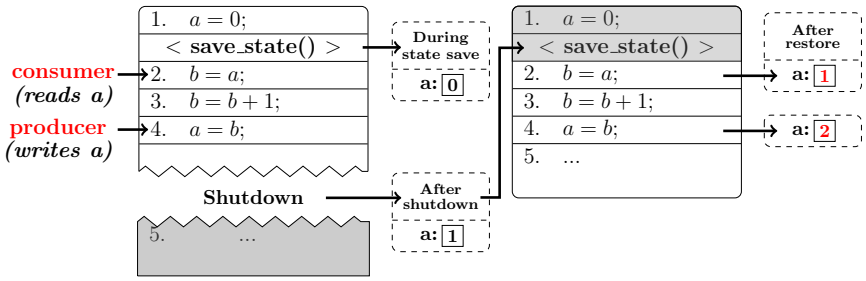


Figure 7.1: *Example of a data access anomaly [75].* An energy failure causes the re-execution of portions of a program, causing an unexpected result.

of mixed-volatile platforms [64, 74, 85, 100], such as the one of Figure 1.5. We recall that, in the example of Figure 1.5, variable a is non-volatile and an energy failure causes the re-execution of lines 2-4, leading to a result that differs from the one of an equivalent continuous execution.

As we point out in Chapter 1.2.2, the literature overlooks intermittence anomalies that may happen whenever devices interact with the environment, failing to provide concepts to analyze this new type of intermittence anomaly, which is not specific to mixed-volatile platforms. Further, the literature lacks tools to verify the presence of intermittence anomalies and to test intermittent executions of programs.

To address these open problems, we extend our early work on intermittence bugs [74], dividing intermittence anomalies into *memory-related* anomalies and *environment-related* anomalies. We then provide (i) new techniques to test memory-related intermittence anomalies, which demonstrate a higher efficiency than existing techniques [74], and (ii) a set of techniques to identify and analyze environment-related intermittence anomalies. In the remainder of this chapter, we report on the main aspects of our contribution, whereas a more detailed analysis is available in the published paper [75].

Memory-related anomalies. Following our previous classification of intermittence bugs [74], we classify memory-related anomalies into *data access anomalies*, *activation record anomalies*, and *memory map anomalies*. We already discuss these anomalies in Chapter 4.1.1. We recall that data access anomalies cause the computation of a wrong result, as Figure 7.1 shows. Activation record anomalies and memory map anomalies follow a similar pattern to data access anomalies. However, they respectively affect functions activation records and dynamic memory mappings leading to unexpected program jumps, crashes, or memory leaks.

We determine a general pattern among the three types of memory-related anomalies, which allows us to identify the minimum amount of information necessary to verify their occurrence. We call *producer* a machine-code instruction that alters a non-volatile memory location and *consumer* a machine-code instruction that accesses a non-volatile memory location. In the example of Figure 7.1, the instruction at line 4 is a producer for variable a, and the instruction at line 2 is a consumer for variable a.

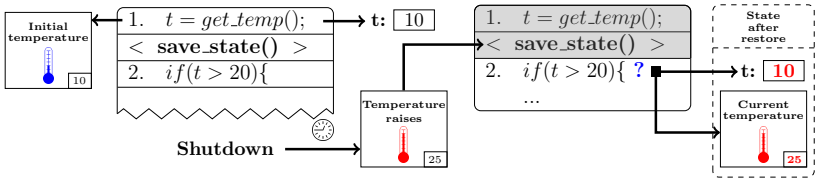
A program contains a memory-related anomaly if it executes a sequence of instructions where a producer executes after a consumer of the same memory location. An energy failure after the execution of the producer causes the consumer to read data that a future instruction wrote in the previous power cycle, that is, the producer. This is the case in the example of Figure 7.1, where the consumer at line 2 eventually reads data that the producer at line 4 wrote during the previous power cycle. This generalizes the existing concept of WAR hazards [64, 74, 85, 100], applying it to any memory-related anomaly. A more detailed description of this consumer-producer pattern is available in the published paper [75].

We use this producer-consumer pattern to devise a technique that locates where memory-related anomalies may happen inside programs. Unlike existing literature [74], we locate memory-related anomalies without simulating energy failures. We search for consumer-producer patterns inside a trace of non-volatile memory accesses collected from a continuous execution of a program. This reduces the time required to analyze programs from several minutes to a few seconds. We provide further details of our technique and its evaluation in the published paper [75].

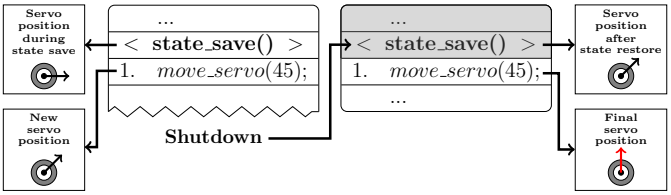
Environment-related anomalies. We analyze unexpected behaviors that may happen whenever battery-less devices interact with the environment, classifying them as *input-related anomalies* and *output-related anomalies*. As we argue in Chapter 1.2.2, these cases of unexpected behaviors were previously overlooked by existing literature.

Figure 7.2 recalls the examples of environment interactions that we describe in Chapter 1.2.2. Figure 7.2(a) shows an example of an input-related anomaly. An energy failure causes the program to access temperature data sensed in a previous power cycle, which continues the execution considering an old environment state. Deciding if this behavior is unexpected depends on the application requirements. For example, programs that record long-term trends may still value data sensed in previous power cycles.

We define two access models that identify how programs access previously-sensed environment states. A *most-recent* access model indicates that programs must access and sense environment data during the same power cy-



(a) The device senses the environment



(b) The device alters the environment

Figure 7.2: *Examples of unexpected behaviors happening when battery-less devices interact with the environment [75].*

cle. A *long-term* access model indicates that programs can access environment data sensed in previous power cycles. The program in the example of Figure 7.2(a) accesses the environment temperature using a *long-term* access model.

The correct access model depends on application requirements and is application-specific. Therefore, an input-related anomaly occurs whenever the program behavior does not reflect the required access model.

We devise a program analysis technique that identifies the access model of accesses to input-related data. Our technique sequentially executes programs and keeps track of access to input-related data by propagating variable-specific metadata, which we then use to identify access models. We signal an input-related anomaly if the identified access model does not reflect developer requirements. A more detailed description of this technique is available in the published paper [75].

Output-related and input-related anomalies follow a similar pattern. Figure 7.2(b) shows an example of an output-related anomaly. An energy failure causes the program to execute duplicate operations, altering the environment state multiple times.

Similarly to input-relate anomalies, deciding if this behavior is unexpected depends on the application requirements. Therefore, our analysis of output-related anomalies is dual to input-related anomalies. We provide further details in the published paper [75].

ScEpTIC. We design and implement `ScEpTIC`, a tool written in Python

that simulates and analyses programs' intermittent executions. ScEpTIC emulates the execution of LLVM IR [1], an intermediate representation of programs' source code close to machine code. This makes ScEpTIC independent from devices' target architecture. ScEpTIC implements our techniques to analyze memory-related and environment-related anomalies and provides developers with an emulation environment to test intermittent programs. ScEpTIC code and documentation are available as an open-source release [69].

We use ScEpTIC to evaluate our techniques to analyze intermittence anomalies. Our experiment results show that our techniques run up to ten orders of magnitude faster than the baselines we consider, allowing developers to analyze intermittence programs in a reasonable time. A more detailed description of our results is available in the published paper [75].

Research connections. ScEpTIC demonstrated to be a fundamental tool for the PhD research, as it enabled the exploration and evaluation of various techniques and system designs. Throughout the PhD research, we extend and update ScEpTIC with numerous simulation features, including accurate simulations of devices' energy consumption and simulations of energy buffers, energy sources, circuitry external to the MCU, and custom hardware designs. We use ScEpTIC to (i) implement and evaluate ALFRED [73] and (ii) implement, explore, and evaluate our DVFS hardware/software co-designs of Chapter 10.

Finally, this research direction gave us essential knowledge and intuitions on intermittence anomalies and non-volatile memory access patterns. As we describe in Figure 1.8, we later use this knowledge to devise intermittence awareness [72] and ALFRED [73], respectively described in Chapter 8 and Chapter 9.

CHAPTER 8

Intermittence Awareness Program Design Pattern

This chapter describes our contribution to intermittent program consistency and energy efficiency [72]. We consider a new perspective on intermittence anomalies, a new type of bug characterizing intermittent computing devices. We introduce *intermittence awareness*, a new program design pattern that intentionally allows the occurrence of specific intermittence anomalies to gain new information regarding intermittent executions of programs. We show one of the many design possibilities that intermittence awareness unlocks by designing an intermittence-aware technique that reduces the overhead required to preserve the computation achieved inside loops. We published this work at the International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys '20) [72], where it received the Best Paper award. The paper is attached in Chapter 12.

We recall that the knowledge gained from our previous work on intermittence anomalies [75], described in Chapter 7, gave us precious insights that we exploit to conceive the concept of intermittence awareness and to design our intermittence-aware technique for loops.

Intermittence awareness. As we describe in Chapter 4.1, mixed-volatile

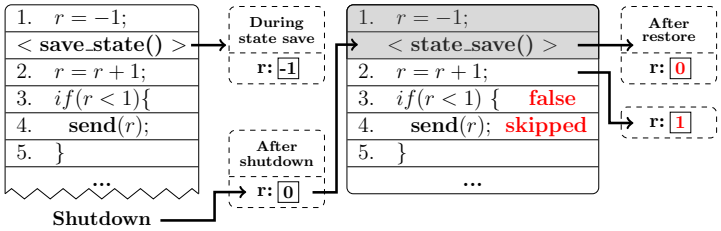


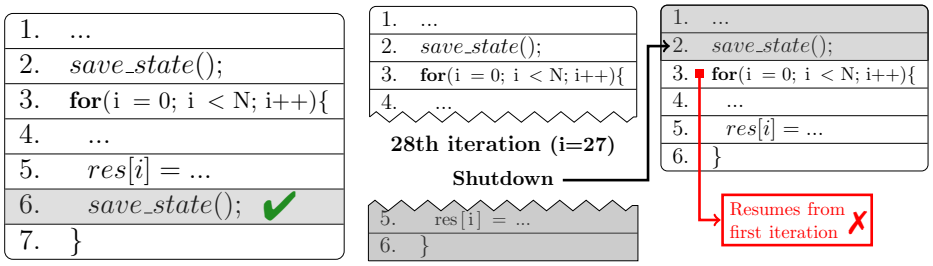
Figure 8.1: *Example of an intermittence-aware program [72].* We allow the occurrence of the intermittence anomaly at line 2, making variable r track the number of energy failures.

platforms [49–51] may experience intermittence anomalies [64, 74, 75, 85, 100], consisting in unexpected behaviors caused by non-idempotent code re-executions after energy failures. Existing forward progress techniques always avoid intermittence anomalies and enforce a computation equivalent to a continuous one [25, 64, 65, 74, 75, 100] executing additional state-save operations at specific program locations [74, 75, 100] or re-executing portions of programs when resuming after energy failures [25, 64, 65].

We take a different approach. Differently from existing literature, we deliberately allow the occurrence of specific intermittence anomalies to make programs aware of their intermittent executions. We call this concept *intermittence awareness*. *intermittence-aware* programs consider intermittence as a new program input and can alter their behavior accordingly to where and when energy failures occur. This allows developers to make their programs react to energy failures, unlocking new program designs.

Figure 8.1 shows an example of an intermittence-aware program. Variable r is non-volatile. Line 1 initializes r to -1, and the device saves the program's volatile state onto non-volatile memory. Next, line 2 increments r to 0, the *if* statement of line 3 evaluates to *true*, and line 4 executes. The computation continues, and an energy failure eventually occurs. When sufficient energy is available, the device restores the program state from non-volatile memory and resumes the computation from line 2. Here an intermittence anomaly happens, as r retained the effects that line 2 produced during the previous power cycle.

Instead of preventing the intermittence anomaly, we deliberately allow its occurrence. This makes r track the number of energy failures since the last checkpoint, as line 2 increments r every time the computation resumes after energy failures. We then rely on this information to prevent re-executions of line 4. The *if* statement of line 3 evaluates to *false* after the first energy failure, and line 4 does not re-execute.



(a) State-save inside loop body

(b) No state-save inside loop body

Figure 8.2: *Preserving loop iterations progress [72].* Fig. (a) saves the state at the end of each loop iteration, preserving loop progress across energy failures. Fig. (b) lacks a state-save operation inside the loop body and energy failures cause to re-execute the loop from the beginning.

Note that this behavior is not possible with existing forward progress techniques [16,64,86,100]. They enforce results equivalent to a continuous execution, ensuring that r is equal to 0 every time the computation resumes after energy failures. Consequently, r would not track energy failures, and line 4 would re-execute after every energy failure.

Figure 8.1 shows only one of the many possible program designs that intermittence-awareness unlocks.

Intermittence-aware loops. We demonstrate one of the possible applications of intermittence awareness by designing an intermittence-aware technique that reduces the overhead required to preserve the computation achieved inside loops. We report here the main aspects of our technique, whereas a detailed description is available in the published paper [72].

As we describe in Chapter 3, ensuring forward progress across energy failures requires devices to periodically save the program state onto a non-volatile memory location. When resuming after energy failures, devices restore the saved state from non-volatile memory and resume the computation from where the state was saved.

Consequently, to preserve the progress achieved inside loop iterations, existing forward progress techniques [16, 64, 86, 100] need to save the program state at the end of each loop iteration, as Figure 8.2(a) shows. In the event of an energy failure, this ensures that the results of executed loop iterations are preserved, as the program resumes from the latest-executed loop iteration. Otherwise, the computation would resume before the loop, and the results of loop iterations executed during the previous power cycle are lost, as Figure 8.2(b) shows. However, saving the program state at the end of each loop iteration introduces a significant overhead detrimental to

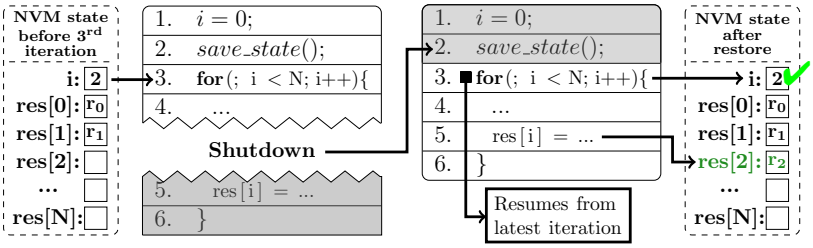


Figure 8.3: *Example of an intermittence-aware loop [72].* We allow the occurrence of specific intermittence anomalies to preserve the computation achieved inside loop iterations, without saving the state at the end of each iteration.

devices' performance [72].

Unlike existing forward progress techniques [16, 64, 86, 100], we do not execute state-save operations at the end of each loop iteration. Instead, we preserve the computation achieved inside loops by allowing the occurrence of specific intermittence anomalies, reducing the overhead of state-save operations.

Figure 8.3 shows an example of our technique, applied to the example of Figure 8.2. We allocate the loop iterator i and the result vector res onto non-volatile memory. Say that the program of Figure 8.3 reaches the third loop iteration, and then an energy failure occurs before the execution of line 5. When sufficient energy is available, the device restores the saved program state and resumes the computation from line 2, outside the loop.

Both i and res retained the effects of loop iterations executed during the previous power cycle. Consequently, the instructions of the loop of line 3 access an inconsistent value of i, as i was 0 when the state was saved, and not 2. A similar case stands for res. However, we deliberately allow the occurrence of intermittence anomalies involving i and res. Line 3 executes and the loop resumes from the third iteration, as i is 2. Moreover, res contains the results of previous iterations, executed in the previous power cycle. Consequently, the program resumes from the third loop iteration as if the state was saved at the end of the second loop iteration. Note that this behavior is not possible if we prevent the intermittence anomalies on i and res, as i would be 0 and the loop would restart from the beginning.

We provide a more detailed description of our technique in the published paper [72], where we also provide a programming abstraction to ease its application.

Results. To correctly preserve the computation achieved inside loops, our technique requires the execution of state-save operations at precise pro-

gram locations during its execution. Consequently, our technique does not apply to dynamic/just-in-time checkpoint-based forward progress mechanisms [11, 12, 54], as they may execute state-save operations at any instant during programs' execution, depending on harvested energy patterns. Therefore, we evaluate our technique only against static checkpoint-based forward progress mechanisms [16,86], as they execute state-save operations at fixed program locations.

We use a heterogeneous set of benchmarks representing typical workloads in intermittent computing. Our experiment results show that, on average, our technique demonstrates a $35.2x$ lower energy consumption and a 48.4 lower workload completion time than existing forward progress techniques. A more detailed description of our experiments is available in the published paper [72].

Research connection. As we describe in Figure 1.8, the knowledge gained from our previous work on intermittence anomalies [75], described in Chapter 7, gave us precious insights that we exploit to conceive the concept of intermittence awareness and to design our intermittence-aware technique for loops. Moreover, this work on intermittence awareness gave us precious insights that we use in `ALFRED` [73]. We design a versioning technique to avoid unwanted anomalies in intermittence-aware loops, which we describe in the published paper [72]. We later use the concept behind this technique as a basis for designing a more complex technique that avoids intermittence anomalies in `ALFRED` [73], as we point out in Chapter 9.

CHAPTER 9

Virtual Memory for Intermittent Computing

This chapter describes our contribution to forward progress, energy efficiency, and intermittent programs consistency [73]. We design ALFRED, a virtual memory abstraction and compilation pipeline for mixed-volatile platforms that automatically identifies the most efficient mapping of the program state across volatile and non-volatile memory. We published this work at the ACM Conference on Embedded Networked Sensor Systems (SenSys 2021) [73]. We release an open-source prototype of ALFRED code and documentation [70] along with a comprehensive technical report of ALFRED compilation pipeline techniques [71].

We recall that, as Figure 1.8 depicts, the knowledge gained from our previous works on intermittence anomalies [72,75], described in Chapter 7 and Chapter 8, gave us precious insights that we exploit to conceive ALFRED techniques.

Program state mapping challenges. As we describe in Chapter 3, to ensure program forward progress across energy failures, devices must periodically save their program state onto a persistent non-volatile memory location and then restore it when energy returns. State-save operations introduce a computation and energy overhead proportional to the size of the saved program state as devices pause the program execution and copy the

program state onto non-volatile memory. State-restore operations entail a similar case.

Mixed-volatile platforms [49, 50] allow developers to directly allocate portions of the program state onto non-volatile memory. Being these portions non-volatile, they are automatically preserved across energy failures and can be excluded from state-save and state-restore operations, reducing their overhead. However, this memory allocation increases programs energy consumption and workloads completion time, as now a subset of program instructions directly target non-volatile memory, whose accesses require up to 247% of the energy and twice the number of clock cycles of volatile memory accesses [49, 50, 75], Moreover, as Chapter 4.1 describes, programs that directly access non-volatile memory may experience intermittence anomalies [75], whose avoidance requires saving the program state more frequently [74, 75, 100] or re-executing portions of programs when resuming after energy failures [25, 64, 65], further increasing programs energy consumption and workloads completion time.

For these reasons, identifying the optimal mapping of the program state across volatile and non-volatile memory is non-trivial, as it also depends on multiple factors, including program execution flow and device energy consumption.

Existing mapping approaches. As we describe in Chapter 5.1.2, existing forward progress techniques allocate the entire program state onto non-volatile memory [54, 59, 66, 100] or require developers to manually specify the mapping of the program state across volatile and non-volatile memory [25, 64, 65, 103], leading to sub-optimal performance. Unlike these works, we automatically identify the optimal mapping of the program state across volatile and non-volatile memory, without requiring user intervention.

The only work that automatically explores various mapping of the program state is the one of Jayakumar et al. [55]. They evaluate all the possible mappings of entire program segments, such as global variables, program code, and stack frames, at single function granularity and apply the most energy-efficient one. The mapping is applied at runtime, introducing an energy overhead that may decrease device performance. Moreover, such high granularity level limits the extracted performance, as the access pattern to single data items may change multiple times across the execution of a function. Unlike this work, our mapping is resolved at compile-time and considers the granularity of single instructions, providing a higher optimization opportunity without introducing runtime overhead.

ALFRED. We design a virtual memory abstraction and compilation pipeline

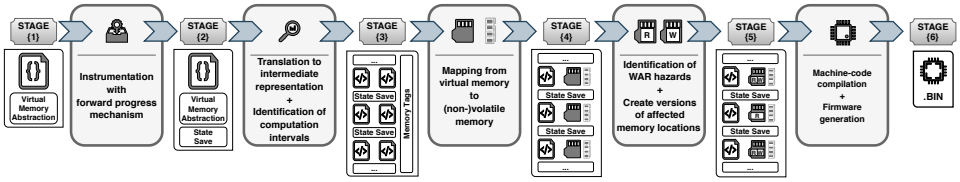


Figure 9.1: *ALFRED compilation pipeline [73].*

for mixed-volatile volatile platforms, called ALFRED. ALFRED automatically identifies the optimal mapping of the program state across volatile and non-volatile memory.

Unlike existing works [25, 54, 55, 59, 64–66, 100, 103], ALFRED operates at the granularity of single data items and provides a virtual memory abstraction that relieves developers from specifying any memory mapping for their programs. ALFRED uses a series of program transformation techniques to automatically resolve the mapping of virtual memory at compile-time, deciding what slices of the program state must be allocated onto non-volatile memory and at what point during the program execution.

The general idea behind ALFRED technique consists in using the more energy-efficient volatile memory to store intermediate results that need not to survive energy failures, whereas we allocate data that requires persistence onto non-volatile memory. Therefore, the mapping of each data item is not fixed, and it is automatically adjusted during the program execution based on memory read/write access patterns and program structure.

In the remainder of this chapter, we report on the main aspects of ALFRED compilation pipeline, whereas a more detailed description is available in the published paper [73] and companion technical report [71].

ALFRED compilation pipeline. Figure 9.1 depicts ALFRED compilation pipeline. At each stage, ALFRED addresses compile-time uncertainty due to unresolved memory addresses, memory aliases, loops, and conditional executions with specific program transformation techniques that remove sources of ambiguity. Further details are available in the published paper [73] and companion technical report [71].

Developers write their programs using ALFRED virtual memory abstraction, which simply requires developers not to specify any mapping for the program state. At stage ⟨1⟩, ALFRED applies a developer-specified forward progress mechanism to the input program. ALFRED works on top of any checkpoint-based [11, 12, 16, 54, 66, 86, 100] or task-based forward progress technique [25, 64, 65, 76, 88, 103].

ALFRED program transformations work at the machine-code level. There-

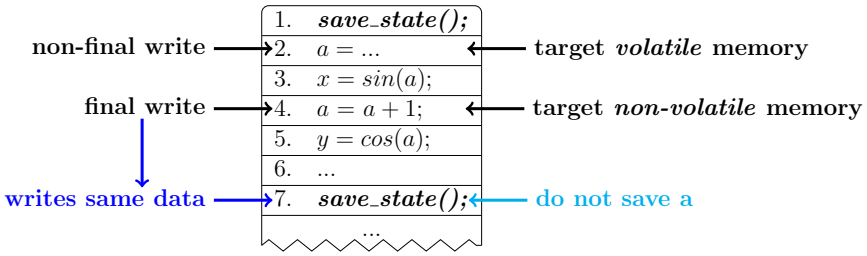


Figure 9.2: *Example of* ALFRED *mapping.* Final memory writes target non-volatile memory and state-save operations need not to save memory locations.

fore, in stage ⟨2⟩, ALFRED translates the program into an intermediate source code representation. Here ALFRED also partitions the program code into *computation intervals*, consisting of sequences of machine-code instructions executed between two state-save operations.

At stage ⟨3⟩, ALFRED analyzes memory access patterns and maps virtual memory to volatile/non-volatile memory accordingly, modifying the target location of memory read and write operations. Figure 9.2 shows an example of ALFRED mapping.

The key behind ALFRED mapping stands in the identification of *final* results, consisting in the same results that state-save operations save. We say a memory write operation is *final* and writes final results if no other operation alters the same memory location before the next state-save operation. In the example of Figure 9.2, line 4 is a final memory write operation.

ALFRED makes every final memory write operation target non-volatile memory, whereas all the other memory write operations target volatile memory. In the example of Figure 9.2, line 4 is a final memory write operation and targets non-volatile memory, whereas line 2 is non-final and targets volatile memory.

ALFRED mapping exploits existing program instructions to save the program state, as final memory write operations target non-volatile memory. Therefore, state-save operations need only to save the register file and special registers, such as the program counter and the stack pointer. This reduces the energy and computation overhead of state-save operations. Moreover, ALFRED achieves differential state saves [7] with zero run-time overhead, as only modified memory locations are saved onto non-volatile memory without requiring runtime memory tracking [7].

ALFRED applies a dual case to memory read operations. A more detailed description of ALFRED mapping is available in the published paper [73] and companion technical report [71].

At the end of stage ⟨3⟩ of Figure 9.1, ALFRED identified the most efficient mapping of virtual memory across volatile and non-volatile memory. However, now a portion of the program directly accesses non-volatile memory and may experience intermittence anomalies [74, 75, 85]. Therefore, at stage ⟨4⟩, ALFRED identifies where intermittence anomalies may happen and avoids their occurrence using a memory versioning technique that tightly integrates with the applied program transformations. Further details are available in the published paper [73]. As we describe in Figure 1.8, the underlying idea behind this versioning technique originates from our works on intermittence anomalies [75] and intermittence awareness [72], described in Chapter 7 and Chapter 8, respectively.

Finally, at stage ⟨5⟩, ALFRED compiles the transformed intermediate representation of the program for the target architecture, producing the final firmware and reaching stage ⟨6⟩, where developers can upload the firmware to the target device.

Results. As we point out in Figure 1.8, we rely on ScEpTIC to evaluate ALFRED performance. We implement a prototype of ALFRED techniques into ScEpTIC [69, 75], which is available as an open-source release [70]. Moreover, we extend ScEpTIC with new features, enabling simulations of devices' energy consumption and energy sources.

We compare ALFRED against state-of-the-art checkpoint-based techniques [16, 59, 66, 86, 100] using a heterogeneous set of benchmarks representing typical workloads in intermittent computing. Our experiments show that ALFRED maps up to 90% of memory accesses to volatile memory and reduces the energy consumption by up to two orders of magnitude. Further, ALFRED more efficient memory mapping reduces the energy consumption, removing non-terminating path bugs [26] from experiment configurations that initially presented them. A more detailed description of our experiments is available in the published paper [73].

CHAPTER 10

Dynamic Voltage and Frequency Scaling for Battery-less Devices

This chapter describes our contribution to energy efficiency for battery-less devices. We identify key features to enable efficient regulation of supply voltage and clock frequency in highly resource-constrained battery-less devices. We implement two hardware/software co-designs that capture these features and expose different trade-offs and functionality, one of which we fabricated. We submitted this work to the ACM Transactions on Sensor Networks, and we are awaiting the reviews.

Efficient operating settings. As we anticipate in Chapter 1.2.3, the energy consumption of low-power MCUs is a function of their operating settings, including operating voltage and clock frequency. The higher the operating frequency, the faster the execution speed and the lower the energy consumption per clock cycle. For example, in the MSP430-G2553 [48], $16MHz$ is $16x$ faster and 47% more energy efficient than $1MHz$, as Figure 10.1 shows.

In principle, the most efficient operating setting corresponds to running the highest possible operating frequency supported by the MCU at the minimum possible supply voltage. However, the selected operating frequency

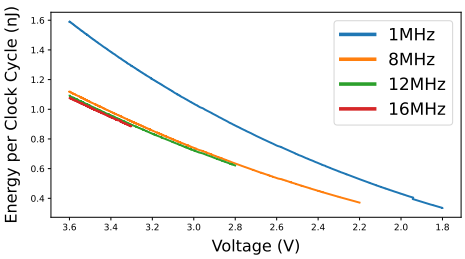


Figure 10.1: *Real measures [6] of the energy consumption of the MSP430-G2553 [48] factory-calibrated frequencies.*

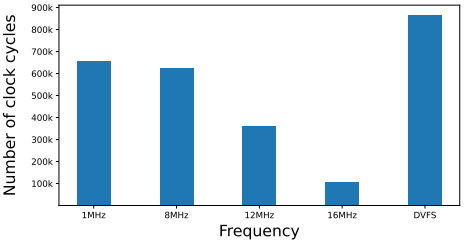


Figure 10.2: *Number of clock cycles executed by a MSP430-G2553 [48] in a single discharge of a* $100\mu F$ *capacitor from* $3.6V$ *to the minimum operating voltage of a given frequency.*

limits the minimum supply voltage at which devices can operate: the higher the operating frequency, the higher the minimum voltage required to operate such frequency. For example, in the MSP430-G2553 [48], $16MHz$ minimum operating voltage is $3.3V$, whereas $1MHz$ minimum operating voltage is $1.8V$.

We recall that, in the absence of voltage regulators between the energy harvester and the sensor node, the input voltage of the MCU depends on harvested energy and energy buffer level, and it is subject to periodic fluctuations. In such a scenario, the reduced operating voltage range of higher operating frequencies pone a severe disadvantage, as it forces devices to shut down at higher voltages, limiting the computation achieved in each power cycle. Although lower operating frequencies are less energy-efficient than higher operating frequencies, their lower minimum operating voltage allows devices to execute more operations in each power cycle. For example, in the MSP430-G2553 [48], the reduced operating voltage range of $16MHz$ yields devices to execute $3.75x$ lower clock cycles than $1MHz$ in each power cycle, as Figure 10.2 shows.

DVFS challenges. Statically selecting an operating frequency leads to sub-optimal performance, as the most efficient frequency changes throughout the computation with the energy buffer voltage. Therefore, battery-less devices should dynamically adapt their operating voltage and frequency at runtime, constantly selecting the most efficient configuration. In the MSP430-G2553 [48], this increases the number of clock cycles executed in a single power cycle by up to $8.1x$, as Figure 10.2. However, applying such a Dynamic Voltage and Frequency Scaling (DVFS) technique to battery-less devices is challenging.

Simple battery-less devices' architectures lack input voltage regulators. Moreover, unlike mainstream processors, the ultra-low-power MCUs used

in battery-less devices lack hardware support to dynamically adapt the system operating voltage and frequency, as well as an operating system that manages such operations. Finally, applying DVFS under harvested energy supply requires constant monitoring of the energy buffer to identify the available operating frequencies. Adding these capabilities may increase energy consumption, resulting in worse performance than a static frequency configuration.

As we describe in Chapter 5.2, the literature provides very few works that apply DVFS to battery-less devices [9, 10, 36]. These works mainly target multi-core processors with DVFS hardware capabilities [10, 36] and adapt MCUs operating frequency to achieve power-neutral operations [9, 10] without necessarily selecting the most efficient one. Unlike available literature, we target highly resource-constrained MCUs, providing (i) a system design for efficient DVFS in battery-less devices, (ii) the first concrete implementation and fabrication of hardware/software co-designs that enable DVFS in battery-less devices, and (iii) a detailed evaluation that highlights the benefits of applying DVFS to battery-less devices.

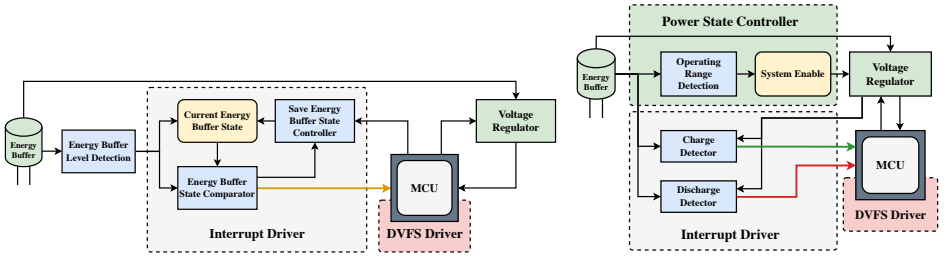
In the remainder of this chapter, we report on the main aspects of our DVFS system design concepts, whereas a more detailed description is available in the paper attached in Chapter 13.

Efficient DVFS design. We devise a generic system design that enables efficient DVFS in energy-constrained battery-less devices.

The key functionality of our system design is the identification of available MCU performance windows, consisting of the most efficient combinations of voltage and frequency settings. Therefore, a performance window consists of an operating frequency and its minimum operating voltage, as it yields the lowest possible energy consumption. For example, for the MSP430-G2553 [48] MCU, we consider the four factory-calibrated frequency settings and their corresponding minimum operating voltage, thus identifying four performance windows: (i) $16MHz$ at $3.3V$, (ii) $12MHz$ at $2.8V$, (iii) $8MHz$ at $2.2V$, and (iv) $1MHz$ at $1.8V$.

At any instant, we apply the most efficient performance window among the ones available. For the MSP430-G2553 [48], this corresponds to the performance window with the highest clock frequency among the available ones. Note that we consider a performance window available if its minimum operating voltage is higher than the energy buffer voltage.

We avoid periodic monitoring of the energy buffer voltage by only tracking changes in the available performance windows. We logically partition the energy buffer into discrete energy levels, one for each identified performance window. For example, for the MSP430-G2553 [48] MCU, we



(a) D^{2}VFS system design. (b) FBTC system design.

Figure 10.3: *Logic representation of our system designs.*

have four discrete energy levels: (i) the range $3.6V$-$3.3V$, (ii) the range $3.3V$-$2.8V$, (iii) the range $2.8V$-$2.2V$, and (iv) the range $2.2V$-$1.8V$. Therefore, we track changes in the discrete energy level, and we change the performance window whenever we identify a change in the discrete energy level. We provide a detailed description of our generic system design in the paper attached in Chapter 13.

Following our generic system design, we implement two hardware/software co-designs for the MSP430-G2553 [48], D^{2}VFS and FBTC. Figure 10.3 depicts the main feature of each system. Both designs use an MCU-driven voltage regulator to set the system operating voltage.

D^{2}VFS. Figure 10.3(a) depicts the system design of D^{2}VFS. D^{2}VFS identifies discrete energy levels using an array of four voltage detectors, one for each operating voltage of the performance windows. The outputs of the four voltage detectors identify the current discrete energy level, which D^{2}VFS stores into a flip-flop. D^{2}VFS detects changes in the discrete energy level using a hardware comparator that compares the content of the flip-flop against the outputs of the voltage detectors. When the comparator detects a change, D^{2}VFS updates the flip-flop and fires an interrupt to the MCU, which probes the voltage detectors to identify the new performance window and applies it. We provide more details of D^{2}VFS system design in the paper attached in Chapter 13.

FBTC. Figure 10.3(b) depicts the system design of FBTC. FBTC offers a more complex system design than D^{2}VFS, which leads to a lower quiescent current and lower delay when scaling to higher performance windows, at the expense of less precise detection of discrete energy levels.

Unlike D^{2}VFS, FBTC powers the system only when the energy buffer voltage is in a pre-defined voltage range, and allows users to select the power-on voltage among four pre-defined voltages. FBTC identifies changes

in the energy buffer level using two operational amplifiers, which detect energy buffer charge and discharge by comparing the energy buffer level against the output of the voltage regulator.

Similarly to D^{2}VFS, FBTC fires an interrupt to the MCU whenever the operational amplifiers detect an energy buffer charge or discharge. The MCU runs a DVFS driver that tracks the current performance window and changes it in response to interrupts. We provide more details of FBTC system design in the paper attached in Chapter 13.

Results. As we describe in Figure 1.8, we rely on `ScEpTIC` [69, 75] to evaluate the performance of D^{2}VFS and FBTC. We extend `ScEpTIC` with the ability to simulate devices' energy consumption, energy buffers, energy sources, circuitry external to the MCU, and custom hardware designs. We then implement a model of D^{2}VFS and FBTC into `ScEpTIC`. These updates are available in `ScEpTIC` repository [69].

We compare D^{2}VFS and FBTC against static frequency configurations using a heterogeneous set of benchmarks representing typical workloads in intermittent computing and various energy source configurations. Our experiments show that D^{2}VFS and FBTC reduce devices' energy consumption by up to 170% and workloads completion time by up to one order of magnitude. A more detailed description of our experiments is available in the paper attached in Chapter 13.

CHAPTER 11

Conclusion and Future Directions

Battery-less devices represent a great opportunity to enable a sustainable Internet of Things. However, as we argued in Chapter 1, their unstable power supply poses several challenges that harden their adoption as mainstream sensors for the Internet of Things [43, 63, 85]. Throughout the PhD research, we tackled a subset of these challenges, demonstrating the potential of this technology, and improving the reliability and efficiency of battery-less devices.

As we described in Chapter 2, the literature was missing examples of long deployments of battery-less devices that account for end-user needs. In Chapter 6, we described our multi-year deployment of battery-less devices that requires zero maintenance without compromising end-user requirements [4]. Our results demonstrated the potential of this technology, showing that battery-less sensors are a viable alternative to battery-powered ones. Moreover, we believe the lesson learned from this work will help future deployments of battery-less devices.

The literature provides a broad range of state-retention techniques [8, 11, 12, 16, 25, 52, 54, 59, 64–68, 76, 86, 88, 100, 103] that enable program forward progress across energy failures, which we described in Chapter 3. A large subset of these techniques [25, 54, 59, 64–67, 76, 100, 103] rely on

mixed-volatile platforms [49, 50] to ease persistent state management at the expenses of an increased systems and programs complexity [25, 59, 64–66, 76, 103], intermittence anomalies [74, 75], and sub-optimal performance [54, 72, 73, 100]. Ensuring safe, reliable, and efficient operations in mixed-volatile platforms provided several analysis and optimization opportunities that we tackled throughout the PhD research.

First, as described in Chapter 4, energy failures may cause unexpected behaviors that may lead to the computation of results different than a continuous execution. The literature lacked both tools and techniques to identify and analyze where these behaviors may happen. In Chapter 7, we described our work [75] on these unexpected behaviors, which we identify as intermittence anomalies [75]. We classified intermittence anomalies and identified new types that previous literature overlooked, which may happen whenever battery-less devices interact with the environment. We provided an in-depth description of their causes, and we designed a set of techniques to verify their occurrence. The knowledge built from this line of research allowed us to identify key insights that we later used in the PhD research for conceiving intermittence awareness [72] and for the design of `ALFRED` [73]. Finally, we implemented `ScEpTIC`, a testing environment for battery-less devices, which we provided to the community as an open-source release [69]. We constantly updated `ScEpTIC` throughout the PhD research, as it helped us evaluate various system designs and techniques, demonstrating a handy tool for battery-less devices research.

Building on our research on intermittence anomalies, we devised intermittence awareness [72], a novel program design pattern considering a new perspective on intermittence anomalies. As we described in Chapter 8, intermittence awareness intentionally allows the occurrence of specific intermittence anomalies to gain new information regarding intermittent executions of programs, enabling developers to consider intermittence as a new program input. We demonstrated one of the many possibilities that intermittence awareness unlocks by designing an intermittence-aware technique that reduces the overhead required to preserve the computation achieved inside loops. On average, our technique reduces programs' energy consumption by $35.2x$ and workloads completion time by $48.4x$.

In Chapter 5, we described the various approaches to reduce the energy consumption of state-save and state-restore operations. Available techniques rely on mixed-volatile platforms to map slices of program state onto non-volatile memory, reducing state-save operations overhead due to a lower volatile state that needs to be saved. However, allocating portions of main memory onto non-volatile memory increases program energy con-

sumption due to non-volatile memory accesses during the computation. Further, it may also introduce intermittence anomalies, whose avoidance requires dedicated instructions that further increase program energy consumption. To address these problems, we devise ALFRED, a virtual memory abstraction and compilation pipeline for mixed-volatile platforms that automatically identifies the most efficient mappings of the program state across volatile and non-volatile memory. In Chapter 9, we described the key elements of ALFRED compile-time techniques and demonstrated ALFRED performance. Our experiment results show that ALFRED enables a faster and more efficient intermittent computation, reducing program energy consumption by up to two orders of magnitude.

Finally, in Chapter 5, we also described the various approaches to ensure battery-less devices operate using the most efficient settings. Available literature fails to adapt battery-less operating voltage and frequency, as they lack hardware and software support for such operations. Therefore, in Chapter 10, we show how dynamic voltage and frequency scaling techniques can be applied to battery-less devices. We devised a system design that captures the key feature required to enable intelligent runtime regulation of supply voltage and operating frequency. We developed two hardware/software co-designs that capture these features, one of which we fabricated. Our experiment results show up to 170% lower energy consumption and up to one order of magnitude faster workload completion time.

In conclusion, the PhD research provided previously-unavailable tools to test intermittence programs, analysis techniques to analyze their behaviors, and techniques that significantly improve the energy efficiency of battery-less devices.

Future directions. Alongside the existing community of intermittent computing researchers, we believe that battery-less devices represent the future for a sustainable Internet of Things [43]. However, this research field still presents untackled challenges.

As we argued in Chapter 4, the literature provides very few tools and testbeds to test battery-less devices. Although ScEpTIC represents a step toward this direction, we aim at extending ScEpTIC to simulate network communications and multiple devices simultaneously. This would enable developers to test multiple device configurations, communication technologies, and network topologies before deployments without requiring additional hardware components for distributed testing of battery-less devices [39].

Further, ALFRED demonstrated better forward progress, as it signifi-

cantly improves devices' energy consumption. ALFRED energy consumption improvement can allow stretching the distance between state-save operations, further reducing their overhead. However, ALFRED works alongside forward progress techniques, requiring already-placed state-save operations. Therefore, to overcome this limitation and further improve device efficiency, we plan to extend ALFRED to automatically insert state-save operations and partition programs into compilation units. This would make ALFRED partition programs by considering where memory accesses happen, reducing both the number of state-save operations and the number of operations that need to target non-volatile memory.

Bibliography

[1] The LLVM compiler infrastructure. https://llvm.org/, 2003 (last access: April 15th, 2023).

[2] Adafruit. I2c non-volatile fram breakout board. https://www.adafruit.com/product/1895, 2013 (last access: April 15th, 2023).

[3] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. The signpost platform for city-scale sensing. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2018. doi:10.1109/IPSN.2018.00047.

[4] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, 2020. doi:10.1145/3384419.3430722.

[5] Saad Ahmed, Qurat ul Ain, Junaid Haroon Siddiqui, Luca Mottola, and Muhammad Hamad Alizai. Intermittent computing with dynamic voltage and frequency scaling. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks*, EWSN '20, 2020. URL: https://dl.acm.org/doi/10.5555/3400306.3400319.

[6] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. The betrayal of constant power $\times$ time: Finding the missing joules of transiently-powered computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES, 2019. doi:10.1145/3316482.3326348.

[7] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, 2019. doi:10.1145/3316482.3326357.

[8] Saad Ahmed, Naveed Anwar Bhatti, Martina Brachmann, and Muhammad Hamad Alizai. A survey on program-state retention for transiently-powered systems. *Journal of Systems Architecture*, 2021. doi:10.1016/j.sysarc.2021.102013.

[9] Domenico Balsamo, Anup Das, Alex S. Weddell, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Graceful performance modulation for power-neutral transient computing systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016. doi:10.1109/TCAD.2016.2527713.

[10] Domenico Balsamo, Benjamin J. Fletcher, Alex S. Weddell, Giorgos Karatziolas, Bashir M. Al-Hashimi, and Geoff V. Merrett. Momentum: Power-neutral performance scaling with intrinsic mppt for energy harvesting computing systems. *ACM Transactions on Embedded Computing Systems*, 2019. doi:10.1145/3281300.

[11] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016. doi:10.1109/TCAD.2016.2547919.

[12] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 2015. doi:10.1109/LES.2014.2371494.

[13] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SENSYS, 2008. doi:10.1145/1460412.1460418.

[14] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Sytare: a lightweight kernel for nvram-based transiently-powered systems. *IEEE Transactions on Computers*, 2018. doi:10.1109/TC.2018.2889080.

[15] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A. Syed, and Luca Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks*, 2016. doi:10.1145/2915918.

[16] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2017. doi:10.1145/3055031.3055082.

[17] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SENSYS, 2019. doi:10.1145/3356250.3360033.

[18] Bernhard Buchli, Daniel Aschwanden, and Jan Beutel. Battery state-of-charge approximation for energy harvesting embedded systems. In *Wireless Sensor Networks*. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-36672-7_12.

[19] Doug Carlson, Jayant Gupchup, Rob Fatland, and Andreas Terzis. K2: A system for campaign deployments of wireless sensor networks. *Real-World Wireless Sensor Networks*, 2010. doi:10.1007/978-3-642-17520-6_1.

[20] Matteo Ceriotti, Michele Corra, Leandro D'Orazio, Roberto Doriguzzi, Daniele Facchin, SÂ¸ Tefan Guna, Gian Paolo Jesi, Renato Lo Cigno, Luca Mottola, Amy L. Murphy, Massimo Pescalli, Gian Pietro Picco, Denis Pregnolato, and Carloalberto Torghele. Is there light at the ends of the tunnel? wireless sensor networks for adaptive lighting in road tunnels. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, IPSN, 2011. URL: https://ieeexplore.ieee.org/document/5779037.

[21] Matteo Ceriotti, Luca Mottola, Gian Pietro Picco, Amy L. Murphy, Stefan Guna, Michele Corra, Matteo Pozzi, Daniele Zonta, and Paolo Zanon. Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, IPSN, 2009. URL: https://ieeexplore.ieee.org/document/5211924.

[22] Qi Chen, Ye Liu, Guangchi Liu, Qing Yang, Xianming Shi, Hongwei Gao, Lu Su, and Quanlong Li. Harvest energy from the water: A self-sustained wireless water quality sensing system. *ACM Transactions on Embedded Computing Systems*, 2017. doi:10.1145/3047646.

[23] Holly Chiang, James Hong, Kevin Kiningham, Laurynas Riliskis, Philip Levis, and Mark Horowitz. Tethys: Collecting sensor data without infrastracture or trust. In *Proceedings of the 3rd IEEE/ACM International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018. doi:10.1109/IoTDI.2018.00032.

[24] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *SIGOPS Operating Systems Review*, 2016. doi:10.1145/2980024.2872409.

[25] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2016. doi:10.1145/2983990.2983995.

[26] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, 2018. doi:10.1145/3178372.3179525.

[27] Alexei Colin, Emily Ruppel, and Brandon 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*, ASPLOS, 2018. doi:10.1145/3173162.3173210.

[28] Peter Corke, Philip Valencia, Pavan Sikka, Tim Wark, and Les Overs. Long-duration solar-powered wireless sensor networks. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EMNETS, 2007. doi:10.1145/1278972.1278980.

[29] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, 2008. URL: https://doi.oirg/10.1109/IPSN.2008.58, doi:10.1109/IPSN.2008.58.

[30] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN, 2005. doi:10.1109/IPSN.2005.1440983.

[31] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN, 2006. doi:10.1145/1127777.1127839.

[32] Istituto Nazionale Geofisica e Vulcanologia. Earthquake data in italy. http://cnt.rm.ingv.it.

[33] Fiona Edwards Murphy, Emanuel Popovici, PÃ¡draig Whelan, and Michele Magno. Development of an heterogeneous wireless sensor network for instrumentation and analysis of beehives. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, I2MTC, 2015. doi:10.1109/I2MTC.2015.7151292.

[34] Varick L. Erickson, Stefan Achleitner, and Alberto E. Cerpa. Poem: Power-efficient occupancy-based energy management system. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN, 2013. doi:10.1145/2461381.2461407.

[35] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, and Thiemo Voigt. Mspsim - an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks, Poster/Demo session*, EWSN, 2007.

[36] Benjamin J. Fletcher, Domenico Balsamo, and Geoff V. Merrett. Power neutral performance scaling for energy harvesting mp-socs. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE, 2017. doi:10.23919/DATE.2017.7927231.

[37] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh Gupta. Pible: Battery-free mote for perpetual indoor ble applications. In *Proceedings of the 5th Conference on Systems for Built Environments*, BUILDSYS, 2018. doi:10.1145/3276774.3282823.

[38] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, 2016. doi:10.1145/2996884.2996889.

[39] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless iot. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, 2019. doi:10.1145/3356250.3360042.

[40] Margherita Guarducci. Ricordo della magia in un graffito del mitreo del circo massimo. In *Mysteria Mithrae*. Brill, 2015. In Italian. doi:10.1163/9789004295605_011.

[41] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, 2014. doi:10.1145/2668332.2668336.

[42] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017. doi:10.1145/3131672.3131674.

[43] Josiah Hester and Jacob Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SENSYS, 2017. doi:10.1145/3131672.3131699.

[44] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017. doi:10.1145/3131672.3131673.

[45] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems*, 2016. doi:10.1145/2903140.

[46] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SENSYS, 2011. doi:10.1145/2070942.2070966.

[47] Natsuki Ikeda, Ryo Shigeta, Junichiro Shiomi, and Yoshihiro Kawahara. Soil-monitoring sensor powered by temperature difference between air and shallow underground soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, (IMWUT), 2020. doi:10.1145/3380995.

[48] Texas Instruments. MSP430-G2553 datasheet. https://www.ti.com/lit/ds/symlink/msp430g2553.pdf, 2013 (last access: April 15th, 2023).

[49] Texas Instruments. MSP430-FR5739 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5739.pdf, 2017 (last access: April 15th, 2023).

[50] Texas Instruments. MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf, 2017 (last access: April 15th, 2023).

[51] Texas Instruments. MSP430 family of mcus. https://www.ti.com/msp430, (last access: April 15th, 2023).

[52] Bashima Islam and Shahriar Nirjon. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2020. doi:10.1109/RTAS48715.2020.00-14.

[53] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN, 2019. doi:10.1145/3302506.3310400.

[54] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quickrecall: A hw/sw approach for computing across power cycles in transiently powered computers. *ACM Journal on Emerging Technologies in Computing Systems*, 2015. doi:10.1145/2700249.

[55] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Transactions on Embedded Computing Systems*, 2017. doi:10.1145/2983628.

[56] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems*, 2007. doi:10.1145/1274858.1274870.

[57] Young-Han Kim, Hyun-Seok Ahn, Changseok Yoon, Yongseok Lim, and Seung-ok Lim. An ambient rf energy harvesting and backscatter modulating tag system enabling zero-power wireless data communication. In *Proceedings of the Seventh International Conference on the Internet of Things*, IoT '17, 2017. doi:10.1145/3131542.3140260.

[58] Charles A. Kircher, Aladdin A. Nassar, Onder Kustu, and William T. Holmes. Development of building damage functions for earthquake loss estimation. *Earthquake Spectra*, 1997. doi:10.1193/1.1585974.

[59] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020. doi:10.1145/3373376.3378476.

[60] Ted Tsung-Te Lai, Wei-Ju Chen, Kuei-Han Li, Polly Huang, and Hao-Hua Chu. Triopusnet: Automating wireless sensor network deployment and replacement in pipeline monitoring. In *Proceedings of the 11th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2012. doi:10.1145/2185677.2185686.

[61] Fujitsu Semiconductor Limited. MB85RC64V $8kb$ i^2c feram datasheet. https://www.fujitsu.com/jp/group/fsm/en/documents/products/fram/lineup/MB85RC64V-DS501-00013-7v0-E.pdf, 2015 (last access: April 15th, 2023).

[62] Gael Loubet, Alexandru Takacs, and Daniela Dragomirescu. Implementation of a battery-free wireless sensor for cyber-physical systems dedicated to structural health monitoring applications. *IEEE Access*, 2019. doi:10.1109/ACCESS.2019.2900161.

[63] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *Proceedings of the Summit on Advances in Programming Languages*, SNAPL, 2017. doi:10.4230/LIPIcs.SNAPL.2017.8.

[64] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2015. doi:10.1145/2737924.2737978.

[65] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM Programming Languages*, (OOPSLA), 2017. doi:10.1145/3133920.

[66] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2018. URL: https://dl.acm.org/doi/10.5555/3291168.3291178.

[67] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2019. doi:10.1145/3314221.3314613.

[68] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020. doi:10.1145/3385412.3385998.

[69] A. Maioli. ScEpTIC documentation and source code. http://sceptic.neslab.it/, 2021 (last access: April 15th, 2023).

[70] A. Maioli. ScEpTIC extension implementing a prototype of ALFRED pipeline. http://alfred.neslab.it/, 2021 (last access: April 15th, 2023).

[71] A. Maioli and L. Mottola. Alfred: Virtual memory for intermittent computing, 2021 (last access: April 15th, 2023). arXiv:2110.07542.

[72] Andrea Maioli and Luca Mottola. Intermittence anomalies not considered harmful. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys '20, 2020. doi:10.1145/3417308.3430266.

[73] Andrea Maioli and Luca Mottola. Alfred: Virtual memory for intermittent computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, 2021. doi:10.1145/3485730.3485949.

[74] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. On intermittence bugs in the battery-less internet of things (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES, 2019. doi:10.1145/3316482.3326346.

[75] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks*, EWSN 2021, 2021. URL: https://dl.acm.org/doi/10.5555/3451271.3451272.

[76] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks*, 2020. doi:10.1145/3360285.

[77] Ramona Marfievici, Pablo Corbalán, David Rojas, Alan McGibney, Susan Rea, and Dirk Pesch. Tales from the c130 horror room: A wireless sensor network story in a data center. In *Proceedings of the First ACM International Workshop on the Engineering of Reliable, Robust, and Secure Embedded Wireless Sensing Systems*, FAILSAFE, 2017. doi:10.1145/3143337.3143343.

[78] Paul Martin, Zainul Charbiwala, and Mani Srivastava. Doubledip: Leveraging thermoelectric harvesting for low power monitoring of sporadic water use. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SENSYS, 2012. doi:10.1145/2426656.2426679.

[79] Gaia Maselli, Mauro Piva, and John A. Stankovic. Adaptive communication for battery-free devices in smart homes. *IEEE Internet of Things Journal*, 2019. doi:10.1109/JIOT.2019.2913231.

[80] Luca Mottola, Gian Pietro Picco, Matteo Ceriotti, Ştefan Gunundefined, and Amy L. Murphy. Not all wireless sensor networks are created equal: A comparative study on tunnels. *ACM Transactions on Sensor Networks*, 2010. doi:10.1145/1824766.1824771.

[81] Saman Naderiparizi, Aaron N. Parks, Farshid Salemi Parizi, and Joshua R. Smith. Î¼monitor: In-situ energy monitoring with microwatt power consumption. In *Proceedings of the IEEE International Conference on RFID*, RFID, 2016. doi:10.1109/RFID.2016.7488017.

[82] Miguel Navarro, Tyler W. Davis, Yao Liang, and Xu Liang. A study of long-term wsn deployment for environmental monitoring. In *Proceedings of the 24th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications*, PIMRC, 2013. doi:10.1109/PIMRC.2013.6666489.

[83] Shuai Peng and Chor Ping Low. Throughput optimal energy neutral management for energy harvesting wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, WCNC, 2012. doi:10.1109/WCNC.2012.6214186.

[84] Adrian I. Petrariu, Alexandru Lavric, and Eugen Coca. Renewable energy powered lora-based iot multi sensor node. In *Proceedings of the 25th IEEE International Symposium for Design and Technology in Electronic Packaging*, SIITME, 2019. doi:10.1109/SIITME47687.2019.8990693.

[85] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, 2014. doi:10.1145/2618128.2618136.

[86] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. *ACM SIGARCH Computer Architecture News*, 2011. doi:10.1145/1961295.1950386.

[87] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. Restop: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 2018. doi:10.3390/s18010172.

[88] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2019. doi:10.1145/3314221.3314583.

[89] Ali Saffari, Mehrdad Hessar, Saman Naderiparizi, and Joshua R. Smith. Battery-free wireless video streaming camera system. In *Proceedings of the IEEE International Conference on RFID*, RFID, 2019. doi:10.1109/RFID.2019.8719264.

[90] Muhammad Moid Sandhu, Kai Geissdoerfer, Sara Khalifa, Raja Jurdak, Marius Portmann, and Brano Kusy. Towards optimal kinetic energy harvesting for the batteryless iot. In *IEEE International Conference on Pervasive Computing and Communications Workshops*, PerCom Workshops, 2020. doi:10.1109/PerComWorkshops48775.2020.9156195.

[91] Nurani Saoda and Bradford Campbell. No batteries needed: Providing physical context with energy-harvesting beacons. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, ENSSYS, 2019. doi:10.1145/3362053.3363489.

[92] Emilio Sardini and Mauro Serpelloni. Self-powered wireless sensor for air temperature and velocity measurements with energy harvesting capability. *IEEE Transactions on Instrumentation and Measurement*, 2011. doi:10.1109/TIM.2010.2089090.

[93] Edward Sazonov, Haodong Li, Darrell Curry, and Pragasen Pillay. Self-powered sensors for monitoring of highway bridges. *IEEE Sensors Journal*, 2009. doi:10.1109/JSEN.2009.2019333.

[94] Vinod Sharma, Utpal Mukherji, Vinay Joseph, and Shrey Gupta. Optimal energy management policies for energy harvesting sensor nodes. *IEEE Transactions on Wireless Communications*, 2010. doi:10.1109/TWC.2010.04.080749.

[95] Philipp Sommer, Jiajun Liu, Kun Zhao, Branislav Kusy, Raja Jurdak, Adam McKeown, and David Westcott. Information bang for the energy buck: Towards energy- and mobility-aware tracking. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, 2016. URL: https://dl.acm.org/doi/10.5555/2893711.2893738.

[96] Lorenzo Spadaro, Michele Magno, and Luca Benini. Poster abstract: Kinetisee - a perpetual wearable camera acquisition system with a kinetic harvester. In *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2016. doi:10.1109/IPSN.2016.7460706.

[97] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/o dependent idempotence bugs in intermittent systems. *Proceedings of the ACM Programming Languages*, (OOPSLA), 2019. doi:10.1145/3360609.

[98] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SENSYS, 2004. doi:10.1145/1031495.1031521.

[99] Claudia Tavolieri and Paola Ciafardoni. Mithra. un viaggio dall'oriente a roma: l'esempio del mitreo del circo massimo. *Archaeology Archives, BA*, 2010. In Italian.

[100] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016. URL: https://dl.acm.org/doi/10.5555/3026877.3026880.

[101] Krishna Vijayaraghavan and Rajesh Rajamani. Novel batteryless wireless sensor for traffic-flow measurement. *IEEE Transactions on Vehicular Technology*, 2010. doi:10.1109/TVT.2010.2050013.

[102] Fan Yang, Ashok Samraj Thangarajan, Sam Michiels, Wouter Joosen, and Danny Hughes. Morphy: Software defined charge storage for the iot. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, 2021. doi:10.1145/3485730.3485947.

[103] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SENSYS, 2018. doi:10.1145/3274783.3274837.

CHAPTER 12

Published Works

Battery-less Zero-maintenance Embedded Sensing at the Mithræum of Circus Maximus

Mikhail Afanasov[*∠], Naveed Anwar Bhatti[*‡], Dennis Campagna[*], Giacomo Caslini[*], Fabio Massimo Centonze[*], Koustabh Dolui[*△], Andrea Maioli[*], Erica Barone[#], Muhammad Hamad Alizai[+], Junaid Haroon Siddiqui[+], and Luca Mottola[*†]

[*]Politecnico di Milano (Italy), [†]RI.SE Sweden, [+]LUMS (Pakistan), [‡]Air University (Pakistan), [#]Microsoft Italy, [△]KU Leuven (Belgium), [∠]Credit Suisse (Poland)

ABSTRACT

We present the design and evaluation of a 3.5-year embedded sensing deployment at the *Mithræum of Circus Maximus*, a *UNESCO-protected underground* archaeological site in Rome (Italy). Unique to our work is the use of energy harvesting through thermal and kinetic energy sources. The extreme scarcity and erratic availability of energy, however, pose great challenges in system software, embedded hardware, and energy management. We tackle them by testing, for the first time in a multi-year deployment, existing solutions in intermittent computing, low-power hardware, and energy harvesting. Through three major design iterations, we find that these solutions operate as isolated silos and lack integration into a complete system, performing suboptimally. In contrast, we demonstrate the efficient performance of a hardware/software co-design featuring accurate energy management and capturing the coupling between energy sources and sensed quantities. Installing a battery-operated system alongside also allows us to perform a comparative study of energy harvesting in a demanding setting. Albeit the latter reduces energy availability and thus lowers the *data yield* to about 22% of that provided by batteries, our system provides a comparable *level of insight* into environmental conditions and structural health of the site. Further, unlike existing energy-harvesting deployments that are limited to a few months of operation in the best cases, our system runs with *zero maintenance* since almost 2 years, including 3 months of site inaccessibility due to a COVID19 lockdown.

CCS CONCEPTS

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

KEYWORDS

Intermittent computing, low-power hardware, energy harvesting.

ACM Reference Format:

Mikhail Afanasov[*∠], Naveed Anwar Bhatti[*‡], Dennis Campagna[*], Giacomo Caslini[*], Fabio Massimo Centonze[*], Koustabh Dolui[*△], Andrea Maioli[*], Erica Barone[#], Muhammad Hamad Alizai[+], Junaid Haroon Siddiqui[+], and Luca Mottola[*†]. 2020. Battery-less Zero-maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *The 18th ACM Conference on Embedded Networked Sensor Systems (SenSys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3384419.3430722

SenSys '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-7590-0/20/11…$15.00
https://doi.org/10.1145/3384419.3430722

1 INTRODUCTION

Ambient energy harvesting is progressively enabling battery-less embedded sensing. A variety of harvesting techniques exist that apply to, for example, light, vibrations, and thermal phenomena [14]. These technologies are naturally attractive wherever replacing batteries is unfeasible or impractical, and represent a foundation to achieve *zero-maintenance* embedded sensing [53].

Real-world deployments. Besides systems that use solar radiation as energy source, few examples exist of long-term deployments demonstrating energy-harvesting zero-maintenance systems [20, 21, 45], as we discuss in Sec. 2. The longest-running such deployment is reported to be operational for 3 months [20]. Further, very few of these deployments serve the needs of actual end users; rather, they are most often instrumental to demonstrate isolated software, hardware, or energy harvesting techniques. We argue that the limited span and scope of such real-world experiences is a sign that current technology is not ready for prime time, as a complete-system perspective is sorely missing.

This paper is about our first-hand experience of such state of affairs, specific to a 3.5-year embedded sensing deployment at the *Mithræum of Circus Maximus*, a UNESCO-protected archaeological site in Rome (Italy). Such an effort is prompted by the municipality of Rome, motivated by the need to understand environmental and structural conditions of the site, as we illustrate in Sec. 3. The site, shown in Fig. 1, is generally closed to the public, completely *underground*, and only accessible through spiral staircases and provisional ladders. Access to the site is strictly regulated to avoid gatherings that may create detrimental environmental conditions and requires authorization from the municipality to assign an accompanying officer. Artificial lighting is temporary, as it is deployed impromptu by archaeologists and restorers only for the duration of their visits.

Our work. Our deployment unfolds through three distinct phases, shown in Fig. 2 and summarized in Fig. 3.

The first design iteration, called Kingdom and illustrated in Sec. 4, is largely based on off-the-shelf components and operates with batteries. We use a commercial platform coupled with acceleration, inclination, temperature, and relative humidity sensors, along with a sub-GHz radio. Despite its satisfactory performance

(a) Mitra altar. (b) Concrete columns.

Figure 1: *Mithræum of Circus Maximus* in Rome, Italy. *The site is underground and only accessible through spiral staircases and provisional ladders, along with proper authorizations.*

during operational times, its reliability is limited, mainly because of batteries. Due to the difficulties to access the site to replace them, this renders the system impractical. After 1.5 years of operation, we eventually turn to energy harvesting. Besides making battery replacement a hurdle, however, the site characteristics rule out most of the energy-rich sources, notably including light.

The second design iteration, called Republic and described in Sec. 5, starts out from the overly optimistic belief—somehow fueled by the lack of experiences akin to ours—that relying on ambient energy is as simple as replacing batteries with a suitable harvester. Due to the site characteristics, we rely on thermal and kinetic sources, harvesting energy from temperature gradients and structural vibrations. We do not expect to achieve energy-neutral operation [8, 65], and design the system as an intermittently-executing one [41]. Intermittent executions interleave periods of active operation with periods of solely recharging energy buffers. We use existing programming techniques [15, 60, 72] to implement sensing, data processing, and communication. The system now operates with essentially zero maintenance, but lower energy availability causes data yield to degrade compared to the Kingdom, which we keep in place (and continue to maintain) as a baseline.

Based on the lessons learned from the earlier designs, the third iteration, called Empire and discussed in Sec. 6, is rooted in two key observations, namely *i)* a hardware/software co-design is required to efficiently manage the little available energy, and *ii)* in our deployment, a form of coupling exists between energy sources and sensed quantities [21, 64]. We make the former concrete through dedicated hardware designs that tightly integrate with program structure and execution model. As for the latter, we capitalize on structural vibrations representing both the energy source and the data we sense. As a result, while not remedying the decreased data yield, Empire greatly improves the *level of insight* into environment conditions and structural health of the site, provably bringing it back to the same level as the battery-operated system.

Outcomes. We report on site-specific insights from sensed data and on system performance in Sec. 7. We consider Kingdom as a baseline for our evaluation, as similar technology demonstrates remarkable measurement accuracy in previous embedded sensing deployments [12, 17–19, 27, 32, 44, 51, 66].

We illustrate the novel understanding of the *Mithræum* conditions we offer to the end users, and how that influences restoration and preservation activities. We show, for example, that relative humidity levels easily cross 90% in a 21C°-25C° temperature range, motivating the need of dedicated preservation procedures. We also analyze the performance trade-offs through the three design iterations and compare energy harvesting to battery-powered operation. We specifically show that in the same conditions, energy harvesting reduces energy availability and thus lowers the system's data yield to about 22% of that provided by batteries, but our design in Empire retains quality of collected data. For example, the conclusions drawn on the site's structural conditions remain unaltered using energy harvesting in Empire as compared to batteries in Kingdom.

Still in Sec. 7, the account of our experience culminates in demonstrating the *zero-maintenance* operation of Empire. Amidst the COVID19 lockdown in Rome, Kingdom goes down as batteries are exhausted while we are prevented from accessing the site, and yet Empire makes the most of the little energy available by promptly recording the occurrence of a moderate earthquake on May 11th, 2020. Analysis of our acceleration data result in a 3.14 estimate of Richter magnitude, close to the (3.2, 3.7) interval officially reported using professional seismographs [29].

In Sec. 8, we discuss key take-aways and design choices that apply more generally to zero-maintenance embedded sensing systems in contrast to the ones that are specific to our deployment. Sec. 9 ends the paper with brief concluding remarks.

2 BACKGROUND AND RELATED WORK

Our work touches upon different areas. We discuss next the relation to those works we deem closer to ours.

2.1 Deployments

A rich body of literature exists on deploying battery-powered embedded sensing systems at different scales and in various environments [12, 17–19, 27, 32, 44, 51, 63, 66, 68, 80]. Common to these efforts are the many sources of unreliable operation and the hectic experience with frequent battery replacements. Lessons from these works help us swiftly set up a fully functional Kingdom, but despite decades of research, limited and unpredictable battery lifetime remains the root cause of malfunction.

Various works demonstrate prolonged lifetime using rechargeable batteries backed by solar [1, 21, 24, 28, 46, 64, 67] or sometimes kinetic and thermal energy harvesting [21, 64]. The longest such deployment is understandably based on solar, and demonstrates a 2-year uninterrupted operation [24]. In contrast, deployments based on thermal [64] and kinetic [21] energy harvesting are limited in lifespan, extending up to four weeks [64]. Our deployment location is void of solar energy, mandating the use of lower-energy sources like thermal and kinetic, yet the lifetime performance of Empire matches the one of the longest deployment using solar energy.

Fewer examples exist replacing rechargeable batteries with environment-friendly super-capacitors [20, 34, 46, 55, 70] or regular capacitors [45, 73, 75–77] to buffer energy and smoothen harvesting fluctuations. Again, only a fraction of these works consider energy sources other than solar [55, 73, 75, 77], let alone real-world deployments [20, 45]. The longest such deployment uses microbial fuel cells to power nodes for water quality monitoring for three months [20]. Although these efforts communicate invaluable

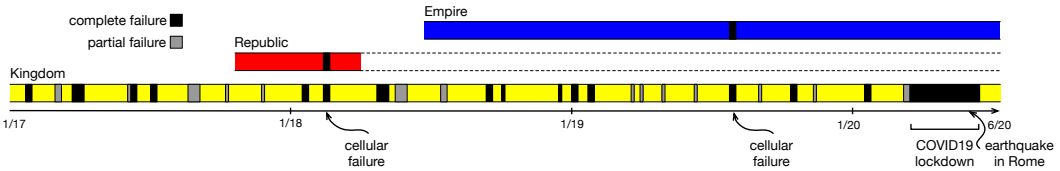


Figure 2: Time evolution of deployments at *Mithræum of Circus Maximus*. *Kingdom is battery-operated and covers the entire deployment duration, representing a baseline for the other systems. Republic and Empire use energy harvesting. A partial (total) failure occurs when at least one (all) device(s) stop operating, and lasts until it is resolved through one or multiple site visits.*

Phase	Time span	Energy source	Hardware
Kingdom	January 2017 - time of writing	Batteries	Off the shelf
Republic	November 2017 - March 2018	Thermal and kinetic	Custom
Empire	June 2018 - time of writing	Thermal and kinetic	Custom

Figure 3: Design iterations at *Mithræum*. *We name them after the three major ages of ancient Rome. These names, however, have no relation to the legacy of the* Mithræum.

lessons on specific techniques, they provide no evidence of a complete system design. Similarly, only a few of them concretely fulfills the requirements of real end users [20, 45], unlike what we do here.

2.2 System Support

Limited form factors impose restrictions on the harvesting unit, limiting power supply to tens of *mW* [20, 34, 46]. This creates a demand-supply gap, which is tackled through two possible approaches: *energy-neutral system design* and *intermittent computing*.

Energy-neutral systems. The idea is to aggressively tune system performance to achieve a demand-supply balance, thus enabling continuous operation [8, 9, 33, 69, 78, 79]. A range of hardware and software optimizations exist to improve energy generation or reduce its consumption, such as maximum power point tracking (MPPT) [9, 75], variable duty-cycling [69, 78, 79] and dynamic voltage and frequency scaling (DVFS) [8].

Techniques for energy neutrality, however, tend to cap the system performance, squeezing the set of feasible applications. Energy neutrality, moreover, may simply not be feasible whenever the average input power is lower than a minimum requirement. This is precisely our setting, where the thermal and kinetic sources offer an insufficient energy content to even conceive continuous operation.

Intermittent computing. Unlike energy-neutral system design, intermittent computing allows energy to buffer for performing operations whose power consumption may exceed the maximum harvesting capabilities. Executions thus become intermittent [41]: periods of active operation are interspersed with periods for recharging energy buffers, while the rest of the system is quiescent.

Intermittent systems typically employ techniques such as checkpointing [4, 10, 11, 15, 47, 60, 61, 72, 85] or task-based programming abstractions [22, 57, 59, 62, 74, 89] to recover from power failures. The former consist in replicating the application state on non-volatile memory, where it is retrieved back once the system resumes with sufficient energy. The latter target mixed-volatile platforms and offer abstractions that programmers use to define and manage persistent state, while taking care of data consistency in case of repeated executions of non-idempotent code [85].

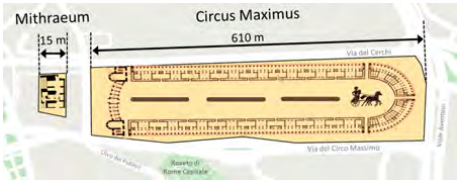


Figure 4: *Mithræum* location relative to the *Circus Maximus*.

Most existing solutions in intermittent computing, again, operate in isolation and lack integration into a complete system. Our work uses a hardware/software co-design for higher efficiency in a complete system and ultimately represents one of the few examples of intermittent computing long-term deployment.

3 MOTIVATION

The *Mithræum of Circus Maximus* is an archaeological site in Rome (Italy). It is largely considered one of the "hidden gems" of its age [81] and is part of the larger UNESCO heritage site in Rome [84].

Site. The *Mithræum* was accidentally discovered in 1931 while performing construction works to build a workshop for the local Opera Theater. Historians conjecture that the location was originally used to host horses and carriages (*carceres*) before entering the nearby *Circus Maximus* for the traditional chariot races. Fig. 4 shows the location of the *Mithræum* relative to the *Circus Maximus*. In the third century *d.C.*, a place of worship to god Mitra was created.

The site unfolds as a series of small communicating rooms, covered by barrel vaults whose remains are shown in Fig. 1. Of unique historical and artistic value are the plaster layers on the walls and the Mitra altar, shown in Fig. 1(a). The workshop of the Opera Theater currently sits right above the *Mithræum* and hosts large machinery and equipment for building theatrical backdrops and sceneries. A set of concrete columns support the ground level of the workshop, reaching into several of the rooms of *Mithræum* or standing on top of the barrel vaults, as shown in Fig. 1(b).

Goal and requirements. The *Mithræum* belongs to a set of archeological sites the municipality of Rome plans to open up to the larger public. Before doing so, an intense process of preservation and restoration is to be carried out. These activities must be planned and executed based on a thorough understanding of the current conditions. Two environmental aspect are key at the *Mithræum*:

[R1] Temperature and relative humidity of plasters: The integrity of the plaster layers may be affected by specific patterns of temperature and relative humidity. Given a certain

Measurement	Sensor	Accuracy	Relevance	Device
Temperature	SHT85	$\pm 0.1C^{\circ}$	Plasters	I/A and T/H
Relative humidity	SHT85	$\pm 1.5\%$	Plasters	I/A and T/H
Acceleration	ADIS16210	$\pm 1mg$	Vaults	T/H
Inclination	ADIS16210	$\pm 0.1^{\circ}$	Vaults	T/H

Figure 5: Sensed physical quantities, corresponding sensing equipment, and device configuration.

temperature, a threshold exists in humidity where hygroscopic salts start forming on the surfaces. The salts absorb water from vapor in the air, causing a corrosion process to happen on the surface. In a site with no external ventilation like the *Mithræum*, this process may only be prevented using specific chemicals whose type, quantity, and method of deposition depend on temperature and humidity [56].

[R2] **Vibrations around the barrel vaults:** Vibrations originating from surrounding vehicular traffic and from the activities at the workshop above may affect the structural stability of the site, and be especially detrimental to the integrity of the barrel vaults [38]. No studies currently exist on the structural conditions of the site and no evidence is available motivating the need for specific interventions, such as installing auxiliary reinforcements of the barrel vaults or deploying dedicated damping mechanisms [38].

Collecting data to support a quantitative investigation on these aspects at the *Mithræum* must co-exist with specific constraints:

[C1] *Placing devices* to record vibrations is difficult, as it requires installing accelerometers *on the columns* supporting the Opera Theater workshop. This literally necessitates climbing up the barrel vaults to access the device, putting at risk the operator safety and the integrity of the vaults. This kind of maintenance operations are to be reduced to a minimum.

[C2] *Form factors* must be reduced, because of the visual impact on historical and artistic pieces. Since the very beginning of our effort, this aspect limits the size of deployed batteries. Such a constraint is not unique to our experience and many embedded sensing deployments, especially in heritage buildings, share similar limitations [12].

[C3] *Commercial chemical batteries* are considered dangerous by the restorers. With average relative humidity values in excess of 90% at the *Mithræum*, as discussed in Sec. 7, the chances that batteries start leaking greatly increase [90]. This is, of course, not welcome in such a sensitive environment.

Lowering the *maintenance effort* is thus key, as it determines how practical is the system and, thus, beneficial for end users.

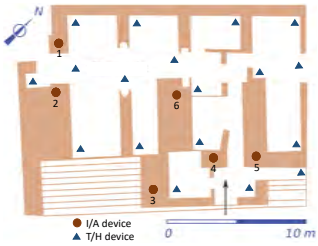
4 BATTERIES → KINGDOM

We set off by using commercial off-the-shelf components. As such, Kingdom represents a baseline based on established solutions.

4.1 Design and Deployment

We describe next the hardware we use for Kingdom, the software we implement, and the initial deployment at *Mithræum*.

Hardware. We use Libelium Waspmotes [54] as the computation and communication core. We couple the computing core with an XBee 868LP sub-GHz radio for communication to a data sink.



(a) Deployment map.

(b) Paper authors installing I/A devices.

(c) I/A device in place.

Figure 6: Deployment at *Mithræum*. *We install 18 T/H devices with temperature and humidity sensors and 6 I/A devices with temperature, humidity, inclination, and acceleration sensors.*

Fig. 5 summarizes the deployed hardware. To read temperature and humidity, we use a Sensirion SHT85 digital sensor through I^2C because of the low-power operation and temperature accuracy, which is sufficient to enable the analysis sought by the restorers [38, 56]. It also features a PTFE membrane for protection against liquids and dust as per IP67 specifications, without affecting the response time. The nodes equipped with this sensor are termed T/H nodes.

Acceleration readings are obtained through an Analog Devices ADIS16210 combined inclinometer and accelerometer, connected through SPI on a subset of the deployed devices. High accuracy of acceleration sensing and availability of the on-board inclinometer motivate this choice; the latter may be used to detect permanent changes in the structure [43]. We calibrate each sensor using a shake table and piezoelectric accelerometers for seismic vibrations as a reference [19]. The nodes equipped with this sensor *in addition* to the temperature/humidity one are termed I/A devices.

Software. We implement a periodic procedure to sense temperature and humidity every 20 minutes and to locally store the readings. At every hour, average and standard deviation of these quantities are computed and reported via radio to the sink. Such a sensing period is deemed to provide sufficient granularity [38, 56].

Onboard I/A nodes, every 20 minutes we additionally record a one minute burst of acceleration readings at 400Hz and sample the inclinometer, according to the guidelines of the structural engineers. At every hour, we process acceleration data by computing the Fast Fourier transform and determining the fundamental frequency as well as spectral density. These information are compressed and also reported to the sink. Such a form of periodic acceleration sensing is common to many deployments for structural analysis [43].

Upon reception, the sink timestamps the data along a global time reference. Between every sensing period, the radio is switched off and the system is placed in low-power mode.

Deployment. Fig. 6 illustrates the deployment. We install a total of 24 devices; 18 devices of type T/H, and 6 devices of type I/A, laid down as shown in Fig. 6(a). For the latter, we use industry-grade epoxy resins to attach the inclinometer/accelerometer sensor to the structure, as shown in Fig. 6(b) and Fig. 6(c) during and after installation. The devices are powered with six type-C batteries.

We deployed a data sink using a Raspberry Pi 3 computer, not shown in the picture, connected to the Internet via 4G. The sink is powered from the grid and, due to the availability of cellular connectivity, could only be installed in a different building at about 250 meters from the *Mithræum*. This motivates the choice of a sub-GHz radio, as the signal needs to penetrate two layers of concrete to reach the sink. Using this radio, no multi-hopping is necessary.

4.2 Lessons Learned

Sec. 7 provides a quantitative account of the performance of Kingdom. We anticipate the fundamental lessons learned, which are input to the following design iteration.

> **Lesson 1:** *Whenever there is sufficient energy, embedded sensing runs like a charm.*

Whenever energy is available, the system provides substantial data yield. Compared to the earlier efforts discussed in Sec. 2, we also note that the effort required to go from zero to a fully-working embedded sensing deployment also drastically reduced. We quantify this effort from one to two person-months.

> **Lesson 2:** *Batteries are the one and only aspect that makes Kingdom unreliable.*

As shown in Fig. 2, Kingdom experiences a number of failures. Batteries are ultimately accountable for all such occurrences, but for two cases of cellular failure. The latter, however, is no significant problem as the sink locally caches sensor data. Our experience contrasts the literature discussed in Sec. 2, where earlier deployment experiences resulted in a number of failures due to a variety of factors, including hardware failures and software bugs [12, 44].

Through a series of ad-hoc experiments, we try and improve the energy figure. The first attempt is based on multi-hop networking [7]. Based on lab experiments in a setting akin to *Mithræum*, we quickly realize the use of such protocols to be detrimental to energy consumption, due to control traffic overhead. In a short-term deployment at *Mithræum* alongside Kingdom, we integrate a transmission power control protocol [86]. Over a 10-day span, we measure the energy performance to improve by a mere 1.7%.

The peculiar conditions at the *Mithræum* makes predicting the system lifetime extremely difficult. High ambient humidity and temperature fluctuations cause the alkaline batteries we use to fail unpredictably. This complicates planning the maintenance visits and the associated logistics, causing the periods of down to prolong. Using different battery technology, such as industry-grade alkaline, pro-alkaline, or lithium make essentially no difference.

Despite our best efforts, maintenance represents a hampering factor regardless of the value of the data. We have two options to proceed. One possibility is to apply iterative improvements to lower the system energy consumption and extend the maintenance cycle. The unpredictability of failures would, however, remain. The other option is to tackle the root of the problem, namely, to seek energy sources other than batteries. We choose the latter.

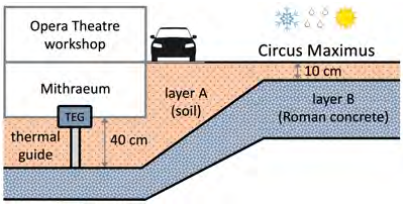


Figure 7: TEG harvester installation. *We exploit thermoelectric generation between air inside the* Mithræum *and layer B, made of Roman concrete and found 10cm below the surface at* Circus Maximus.

5 ENERGY HARVESTING → REPUBLIC

Energy harvesting is often advertised as a direct alternative to battery-powered operation [14]. Because of this, we set off by merely swapping batteries for a suitable harvester, performing the *minimum* effort to make the system work on harvested energy rather than batteries. As a result, the degree of hardware/software co-design here is limited to adapting the software to work with the chosen hardware, that is, the design process starts with the selection of hardware components and ends with the implementation of the necessary software functionality. We eventually recognize that this approach is, in fact, naive.

5.1 Design and Deployment

The opportunities for energy harvesting at the *Mithræum* are minimal. As described in Sec. 3, the site is underground and is not illuminated besides when someone is there. Moreover, the nature of the site requires minimally-invasive solutions. T/H and I/A devices thus use different energy harvesting mechanisms because of their different deployment configuration; T/H being placed next to the ground, whereas I/A being attached to the structure.

Thermoelectric energy harvesting. Fig. 7 shows the setup for T/H devices. At about 40cm below the soil at *Mithræum*, a layer of debris is found largely composed of what is called *Roman concrete* [52]. The same layer is found at the nearby *Circus Maximus* at about 10cm below the surface. Scholars conjecture that the two layers are, in fact, the same [52, 81]. This means that *Circus Maximus* potentially acts as a $\approx$73,000m^2 thermal surface linking the *Mithræum* to the outside. The heat flux generated because of the thermal transfer between air, layer A, and layer B creates an opportunity to employ a thermoelectric energy generator (TEG).

A broad range of commercial TEGs exists. Based on the air temperature values collected during Kingdom and the outdoor seasonal trends in Rome, we expect the thermal deltas between air and layer B to be of some K°. We thus choose a Thermalforce 254-150-36 TEG [82], offering a 30mm by 60mm harvesting surface, connected to layer B through a thermal guide, as shown in Fig. 7.

Available harvesting management circuits usually combine battery charge functionality and output voltage regulation. Solutions for TEG may be passively controlled converters or actively controlled single inductor circuits [83]. The latter offer a dynamic

conversion ratio and maximum power point tracking (MPTT) [9], but require a higher minimal input voltage. Despite this, we use a BQ25570 due to its high efficiency for the range of input voltages that most likely correspond to the TEG output in our conditions.

Because the output voltage of the TEG depends on the direction of heat transfer, depending on time of the day, its output may be positive or negative. However, the BQ25570 does not support negative input voltages and hence the TEG output needs to be rectified before being input to the harvesting circuit. We build an ultra low-power rectifier using SiR404DP switches, based on the observation that the TEG output only switches twice a day [37].

Piezoelectric energy harvesting. Thermoelectric generation is not available for I/A devices, as they are too far from the ground. We absorb energy from structural vibrations to power them, taking advantage of the piezoelectric effect. The limited vibrations of the structure, however, require a careful dimensioning of the harvester and of the energy management circuitry, as we discuss next.

We employ a ReVibe modelD energy harvester [30]. The device can be customized by the manufacturer for highest efficiency at a given resonance frequency. We do this based on vibration data gathered with Kingdom. We choose this specific harvester over alternatives, for example, the modelQ [31] of the same manufacturer, because of the higher power output at the target frequencies. The harvester is attached to the columns of Fig. 1(b) using the same epoxy resins used for attaching the accelerometer/inclinometer.

Based on similar considerations as for T/H devices, we use a BQ25505 here as well. No rectifier circuit is needed.

Computing and communication. As discussed in Sec. 2.2, due to the limited energy availability, it is not conceivable to achieve energy-neutral operation [8, 65]. Therefore, we design the system to work in an intermittent fashion [41].

The Libelium Waspmote we use for Kingdom is not designed to work in such a setting. We opt to build our own computing and communication platform, using an MSP430FR5989 MCU coupled to a CC1101 transceiver. The choice of an MCU from the FR series is motivated by the need of non-volatile memory to manage persistent state. The radio chip retains the advantages of sub-GHz transmissions described in Sec. 4, with comparable energy consumption. The sensors we use are the same as in Kingdom.

We configure the output voltage of the BQ25505 buck converter to 2.2V, which represents the worst-case energy need including sensing, local processing, and data transmission. This means that the device is activated as soon as the capacitor voltage is at or above 2.2V. We also configure the BQ25505 to operate in pass-through mode whenever the capacitor voltage falls below this value, to prolong the execution for as long as possible.

We use a 20μF capacitor as energy buffer. We determine its size through a mixed analytical and experimental approach [83], striking a balance between charging times and available energy to guarantee eventual progress. A too large capacitor may take long to charge to a sufficient level, yielding large periods of no system operation when interesting environmental events might be missed. A too small capacitor may not suffice to supply enough energy to complete energy-intensive operations, such as transmitting data.

An external Abracon AB18X5 real-time clock (RTC) keeps track of the passing of time while the MCU is off, connected via I^2C. We choose this over remanence timekeepers [26, 42] because of the lower power consumption in the setting we consider. As we only require minute granularity, using the internal RC oscillator on the AB18X5 requires a mere 14nA current. Should the capacitor voltage fall below the RTC supply voltage, causing the latter to reset, we post-process the data at the sink to re-align the timestamps to the global time reference [87]. According to our logs, this happens roughly twice a year in our deployment.

Programming. As described in Sec. 2.2, system supports exist for intermittent computing [41]. In Republic, we use a static checkpoint approach [15, 72], which inlines calls to a checkpoint library to copy the complete system state onto FRAM. To place checkpoints, we profile the energy consumption of different parts of the code [3] and accordingly inline checkpoint calls. At each call, a checkpoint takes place if the capacitor voltage drops below a threshold that barely guarantees the energy to dump the state on FRAM.

We opt for static as opposed to dynamic checkpoints [10, 11, 47, 48], as we cannot afford additional hardware. Compared to task-based programming abstractions [22, 42, 57, 59] that require significant restructuring of the program [50], we wish to leverage the earlier codebase used in Kingdom.

Sensing. As the device activates depending on energy intake, the periodicity of sensing can no longer be guaranteed. Depending on harvesting performance, we may simply not have sufficient energy to activate the device every 20 minutes. As a result, we modify the local processing and data transmission functionality to execute only when the same amount of data as in Kingdom locally accumulates.

It may also happen that the required operation complete with some energy left. To avoid unnecessarily performing a checkpoint at this time, we enter a sleep state, which is a technique borrowed from Lukosevicius et al.[58]. This includes switching the radio off, putting the MCU in the lowest power mode, and setting a timer to trigger another round of sensing in 20 minutes. This also ensures that, at least in the cases where some consecutive rounds may be achieved, this happens with the same period as in Kingdom.

5.2 Lessons Learned

Republic represents the *minimum* of design and implementation effort to turn a battery-operated system into an energy-harvesting one. Similar to Sec. 4.2, we discuss here the main learned lessons and postpone the performance discussion to Sec. 7.

> **Lesson 3:** *When executions are intermittent, peripherals become markedly decisive.*

The workload at *Mithræum* is peripheral-bound. Peripherals execute asynchronously with respect to the computing unit. Their functioning is characterized by own states, frequently updated due to the execution of I/O instructions. Information on peripheral states is not automatically reflected in main memory, neither it may be simply queried or restored [16]. System support for intermittent computing often only provides support for the computing unit and expect developers to take care of peripherals [22, 57, 89]. Similarly, the few systems addressing the intermittent peripheral problem are not integrated with those for the computing unit [6, 13, 16].

To address this issue in Republic, we manually replicate the initialization procedures of all peripherals, including sensors and

radio, at every point in the code where execution can possibly resume after a power failure. This is necessary as we cannot anticipate for how long an execution proceeds after resuming and thus what peripherals are used when. The profiling data we use to place checkpoint calls indicates, however, that re-initializing peripherals this way accounts for about 28% of the overall energy consumption, opening up avenues for energy savings with a better solution.

We also crucially realize how the use of radio and sensors vastly determines how far the computation can progress. We observe that the first checkpoint call right a packet transmission is systematically triggering a checkpoint, as radio operations are sufficient to cause the capacitor voltage to fall below the checkpoint threshold. However, handling peripherals is not the only source of inefficiency.

> **Lesson 4:** *When energy is scarce, sleeping may not be a smart choice.*

The technique we use in case some energy is left after completing the required workload ultimately represents a waste of energy. Based on the logs we collect, after setting a timer to expire in 20 minutes, in about 89% of the cases the node dies before the timer fires. This means that the energy invested in keeping the system in sleep mode is wasted, as another round of sensing cannot happen in the majority of the cases. In Sec. 7, we further quantify the performance impact of this design choice.

To some extent, this is again an effect of how peripherals impact the energy figure. As every time the device activates at least one peripheral is used, the chances that some energy is left that could power the sleep state for another 20 minutes are slim. This problem aggravates if the radio is also used. If we only consider the cases where we set the 20-minute timer after a packet transmission, in 98% of the cases the node dies before the timer fires.

> **Lesson 5:** *Energy availability may not necessarily overlap with events of interest.*

We expect the data yield to be affected, due to the lower availability of energy. As reported in Sec. 7, Republic can only provide about 22% of the net amount of data Kingdom provides on a monthly basis. Worse is that the information gain obtained from Republic is comparatively way below the reduction in data yield.

This observation particularly applies to I/A devices. In Kingdom, the relative abundance of acceleration data forgives that acceleration sensing is not necessarily synchronized with events of interest, such as activities at the Opera Theatre workshop or vehicular traffic. In Republic, I/A devices activate only depending on capacitor voltage levels, which might cross 2.2V merely because of vibration noise of no interest [43]. The structural engineers state that, by only using the acceleration data from Republic, *no structural analysis is possible*, due to the signal information being too poor for modal analysis, irrespective of the amount of collected data [43].

6 BETTER ENERGY HARVESTING → EMPIRE

We eventually choose to co-designing the hardware and software. This is primarily based on the experience and insights gained from Republic, but also on a number of (failed) attempts at remedying the deficiencies of Republic by only working at software level.

For example, alongside Republic, we eventually deploy two additional I/A devices with a customized software implementation that postpones acceleration sensing to the next active cycle in case the initial 1-sec burst of data indicates no specific event of interest. In a sense, we bet on the fact that we may have better chances to capture something interesting at the next time around. Over a two-week span, we realize that system performance stays the same in terms of enabling structural analysis, as the hardware may keep activating the device because of vibration noise and events of interests are completely uncorrelated with that. In terms of data yield, the performance even degrades, because of the additional processing required to decide on the possible postponement.

In contrast to Republic, therefore, we attack the problem by looking at hardware and software together. As a result, for example, we discover and exploit further opportunities by capturing the coupling between energy sources and sensed data.

6.1 Design and Deployment

We realize different designs for T/H and I/A devices. Their key elements are described next, whereas attached sensors and the timer subsystem remain the same as in Republic.

Programmable activation threshold. Fig. 8 shows the block diagram of the 2nd generation T/H device. It offers two fundamental features: *i)* it allows the MCU to dynamically configure the amount of energy available at the next device activation, and *ii)* it provides a software-controlled shutdown functionality, which the MCU uses once the required operations are completed.

To achieve these functionality, we place three voltage comparators in parallel; each corresponding to a different activation threshold. We select comparators from the BU49xx series corresponding to voltage levels matching the energy required for *i)* sensing (V_s^{th}), *ii)* sensing and local processing (V_{sp}^{th}), and *iii)* all application functionality including data transmission (V_{spt}^{th}), where $V_s^{th} < V_{sp}^{th} < V_{spt}^{th}$. Every threshold also includes the energy required to dump the state on FRAM once the necessary operations complete. An ADG704 digital multiplexer selects the comparator to use based on the input of a two-bit memory the MCU can program by manipulating two GPIO pins. The choice of components is dictated by both their low energy consumption and their matching with the voltage threshold we require, given the same capacitor size as in Republic.

We implement the two-bit memory using two SN74AUP1G74 flip-flops in a cascading configuration. These feature both an extremely limited quiescent current and a low reset voltage. Below 0.8V, however, they lose their state. We have evidence that this was the case for only eleven times in almost two years. If the flip-flops reset, the circuit causes the multiplexer to select the lowest threshold V_s^{th}. This ensures that some progress is eventually achieved.

We deploy a TPS62736 buck converter, which is optimized for the target range of currents. A further voltage detector turns the "power good" signal up to make the buck converter activate the device whenever the selected input comparator switches its output. As device activation is now separately controlled, we configure the output of the buck converter exactly to 2.1V, which represents an energy-efficient regime for both the MCU [3] and the radio [25] As in Republic, the converter operates in pass-through mode whenever the capacitor voltage falls below this value. The TPS62736 also features an independent "enable" signal that can be used by

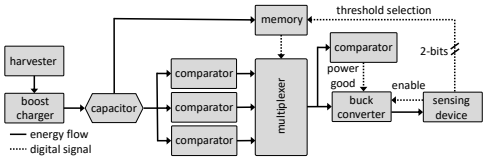


Figure 8: Second generation T/H device in Empire. *The MCU can programmatically configure the amount of energy available at the next device activation and shutdown the system via software.*

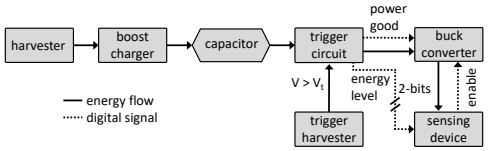


Figure 9: Second generation I/A device used in Empire. *A secondary piezo element triggers device activation. A 2-bit input line informs the MCU of the energy available at activation time.*

the MCU to disconnect from the power sub-system, effectively implementing a software-controlled shutdown.

Our concept of programmable activation shares similarities with Capybara [23] and Dynamic Energy Burst Scaling (DEBS) [36], but we trade generality for a lower energy overhead. The whole power sub-system, in fact, only consumes 5.35μA of quiescent current.

Vibration-triggered activation. Fig. 9 depicts the block diagram of the 2nd generation I/A device. It features two key elements: *i)* a second piezo element that operates as a trigger, activating the device only when vibrations above a certain frequency are detected, and *ii)* a 2-bit input line that informs the MCU of the energy available at activation time, which the application uses to determine what operations may be completed in the active cycle.

We use a Piezo.com Q220-H4BR-2513YB bending transducer [71] as trigger, enclosed in a PPA-500x clamping base with a 13mg tip mass. A custom trigger circuit turns up the "power good" line of the buck converter whenever the trigger piezo generates an output voltage above a threshold V_t and the capacitor voltage is above a threshold V_s^{ia} sufficient for acceleration sensing. The buck converter then activates the device. We select both piezoelectric element and tip mass in a way that V_t can be accurately detected and corresponds to vibrations of interest [43].

A set of TLV369x comparators and SiR404DP switches control the "power good" line of the buck converter and the 2-bit "energy level" input line connected via GPIO to the MCU. Upon activating the device, the latter informs the MCU of the amount of energy available, based on whether three additional voltage thresholds are crossed. These correspond to the energy for *i)* sensing and local processing (V_{sp}^{ia}), *ii)* sensing and data transmission (V_{st}^{ia}), and *iii)* all application functionality (V_{spt}^{ia}), where $V_s^{ia} < V_{sp}^{ia} < V_{st}^{ia} < V_{spt}^{ia}$. Depending on this input, the application schedules the operations it can perform given a certain energy budget.

The roles and connections of the remaining components are similar to the T/H devices. In this case, the power sub-system only consumes 4.98μA of quiescent current.

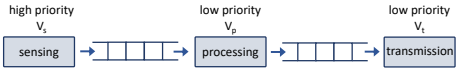


Figure 10: Task-based program structure. *Every task demands a different amount of energy. Tasks have different priorities and are connected through non-volatile data pipelines.*

Programming. Both designs aim to exert a higher control on erratic energy patterns. T/H devices achieve that by giving the MCU the ability to decide the energy available for the next iteration. I/A devices proactively provide the MCU with information on available energy at the time of activation. Both designs also give the MCU a means to shutdown the device whenever required.

Taking advantage of these features requires to co-design the software in ways to *i)* precisely isolate and decouple the functionality corresponding to different voltage thresholds, *ii)* abandon the strictly-sequential execution semantics, so different functionality can execute independent of each other, depending on available energy. In doing so, we must come to terms with the need to refactor the codebase created in Kingdom, which is unavoidable now.

We opt for a task-based structuring of the code, shown in Fig. 10. A task is an atomic piece of functionality that executes in a transactional manner [22, 57, 59, 62, 89]. If energy suffices and a task completes, its output are committed onto a non-volatile data pipeline. If a power failure happens before the task completes, the effects of a partial execution are lost and the task restarts from the beginning. Unlike existing solutions, our design enables a form of energy-aware scheduling that simplifies system operation, while reducing overhead. Upon device activation, the sensing task is enabled and sensors are (re-)initialized. The power sub-system ensures that sufficient energy is available for this when activating the device, as V_s^{ia} is certainly crossed. We additionally enable any other task with input data and whose energy demands match the available energy and (re-)initialize (only) the necessary peripherals.

We set higher priority for the sensing task to make sure we do not miss any environment data. Among enabled tasks, we therefore run the sensing task first and commit its results on FRAM. We proceed to run the other enabled tasks and similarly commit the results on FRAM. For I/A devices, as long as sufficient energy is available not to starve the transmission task, no data buffers overflow. For T/H devices, we can proactively ensure this by configuring the activation threshold to provide the transmission task with sufficient energy.

In Sec. 8, we discuss the limitations of our work and cast our design rationale in the larger context of battery-less systems.

6.2 Lessons Learned

The efficient operation of Empire, reported in Sec. 7, leads us to additional lessons learned.

> **Lesson 6:** *Determine how much energy you need, when, and for which operation.*

In Empire, knowledge of energy demands and is key in our hardware/software co-design. The inefficient operation of Republic, instead, stems from the application whimsically unfolding through three distinct phases with different energy demands and periods. Sensing is moderately energy consuming and happens most often.

Local processing is the least energy consuming, but comparatively happens more rarely, as it needs a batch of sensed data to operate on. Data transmission is the most energy hungry, and must happen as frequently as local processing, as it relays the results to the sink.

Energy management in Republic was totally unaware of these aspects and only operated based on the net energy inputs from the harvesters. Existing literature, besides a few exceptions [3, 35, 39], offers little support for gaining or exploiting this information, and almost never includes peripherals in the picture.

> **Lesson 7:** *Only perform an operation when you have the energy required; consume no more, no less.*

Empire performs efficiently because it is provided with just the right amount of energy for the required operation and for committing the results on FRAM. This means, for example, that sufficient energy to perform local computation and data transmission is at disposal whenever the required amount of data is available. The device then completely switches off to avoid wasting energy doing nothing in sleep state, as in Republic.

> **Lesson 8:** *Selectively activate peripherals, and only when you need them.*

The execution pattern we enforce in Empire also allows us to shave off some of the significant energy overhead for re-initializing peripherals, discussed in Sec. 5.2. Knowing what operations are going to be performed within the given energy budget, only the required peripherals are initialized.

Republic is unable to guarantee this; for example, because we cannot anticipate for how long an execution proceeds after a power failure, as explained in Sec. 5.1. Again available solutions provide a limited foundation to build upon these observations [16, 23, 40].

> **Lesson 9:** *Capturing the coupling between energy sources and sensed quantities is key, if one exists.*

In Empire, we make use of vibrations both as energy source and as a quantity to sense. Significant vibrations are used for both harvesting energy and for triggering the sensing process, if the accumulated energy suffices. On the other hand, crossing the activation threshold merely because of vibration noise does not lead to activating a device, and we rather keep accumulating energy. Only a few solutions currently exist in this direction [21, 64].

7 EVALUATION

We study multiple complementary dimensions. In Sec. 7.1 we evaluate our deployment as a scientific instrument to fulfill the requirements in Sec. 3. We compare the system performance of the three design iterations in Sec. 7.2. The overarching question we seek to answer is whether accurate *zero-maintenance* embedded sensing is possible in our setting. We elaborate on this in Sec. 7.3.

7.1 Application

We separate the discussion of the environmental information we gather, as per requirement **R1** in Sec. 3, from the structural analysis of the site, as per requirement **R2** in Sec. 3. The differences in sensed values between the three design iterations are discussed in Sec. 7.2.

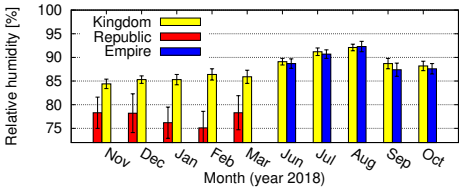


Figure 11: Monthly average of relative humidity recorded by Republic and Empire, compared to Kingdom in the same time interval. *High relative humidity and ambient temperature in the 21C°-25C° range may cause the creation of hygroscopic salts.*

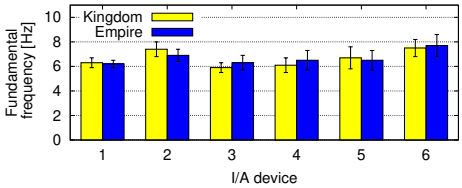


Figure 12: Fundamental frequencies using Kingdom and Empire when both systems are operational. *The structures at* Mithræum *have different fundamental frequencies than possible external excitating phenomena, ruling out resonance behaviors.*

Environment. Fig. 11 shows the monthly average of relative humidity recorded by the three systems in 2018. We obtain comparable trends also in other periods and for temperature data, making the following conclusions applicable throughout the deployment duration. The differences in sensed values between the three systems are due to different energy efficiency, as discussed in Sec. 7.2.

Fig. 11 shows that relative humidity at the *Mithræum* is markedly higher than in a regular environment, with distinct seasonal trends and peaks in the summer months. This may be attributed to the nature of the soil combined with the lack of external ventilation. Together with our recording of ambient temperature in the 21C°-25C° range, the situation corresponds to roughly 15 grams of vapor per kilogram of air, with peaks of 18 grams in the summer months. This is way above the threshold for the creation of hygroscopic salts that possibly cause corrosion processes to occur on the surfaces [56], as explained in Sec. 3, and prompts immediate action by the restorers.

This information is also crucial for a public opening of the site. Existing standards for thermal comfort indicate a maximum of 6 grams of vapor per kilogram of air [5]. This value is less than half of what we record[1]. Ensuring thermal comfort for the general public at the *Mithræum* requires the installation of an auxiliary ventilation system, at least during the opening times. In turn, this would likely change the general environment conditions at the site, making them more variable depending on the operation of the ventilation system. A permanent installation of a minimally-invasive *zero-maintenance* sensing system thus becomes even more fundamental.

Structure. Fig. 12 shows a sample output of the analysis on acceleration data, plotting the average fundamental frequencies recorded throughout the deployment at every I/A device. As explained in

[1]We also experience the thermal discomfort at the *Mithræum*, as we can barely work at the site continuously for more than a couple of hours without reaching the outside.

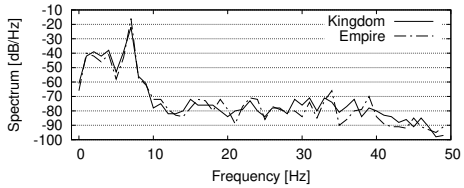


Figure 13: Spectral density at I/A device #6 using Kingdom or Empire when both systems are operational. *Only one dominant frequency exists and most of the energy is concentrated below 10Hz, supporting general validity of the structural analysis.*

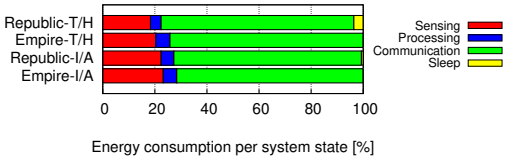


Figure 14: Breakdown of percentage energy consumption depending on functionality. *Peripherals bear an impact on energy, as per Lesson 3 in Sec. 5.2. Accurate energy management makes better use of the energy spent in sleep mode at T/H devices, as per Lesson 4 in Sec. 5.2, as well as Lesson 6, 7 and 8 in Sec. 6.2.*

Sec. 5.2, Republic provides no usable data. This information is valuable in that, if the fundamental frequency of an external excitating phenomenon match those of the structure, then the motion of the structure is amplified, resulting in resonance behavior [43].

The external phenomena may be, in our case, activities at the Opera Theater workshop or vehicular traffic. The values in Fig. 12, however, indicate that the fundamental frequencies of the structures at the *Mithræum* are relatively far from those possibly characterizing the aforementioned phenomena, which are thought to lie above 10Hz [38]. Resonance behaviors may thus be safely ruled out.

This reasoning is confirmed by the information on spectral density, shown in Fig. 13 for node #6 as an example. Only one dominant fundamental frequency exists and most of the signal energy is concentrated below 10Hz. As every fundamental frequency follows a specific deflection shape, usually referred to as vibrational mode, we can argue only one such mode exists for the structure at the sampling points. The analysis on the dominant fundamental frequency thus bears general validity [43].

7.2 System

We assess the performance of the two energy harvesting systems compared to the battery-operated one. We take Kingdom as a baseline hereafter, as the sensing equipment is the same across the three systems and only the power source and associated designs differ.

T/H devices. We return to Fig. 11 to analyze the compare the three systems. The plot shows how Republic constantly underestimates the relative humidity at the site, whereas Empire provides values closer to those of Kingdom. The variability of data around the average is also much higher for Republic than for Empire. This is an effect of Republic's inability to accurately manage the available energy, which is used opportunistically and partly wasted.

Based on detailed logs we collect at a subset of the devices, Fig. 14 quantifies this aspect by showing the breakdown of energy consumption across the four main system states in Republic and Empire. The plot demonstrates the impact of the peripherals on the energy figure, supporting our claims in Lesson 3 in Sec. 5.2. For Republic, it also shows the contribution of entering a sleep state when the required operations complete with energy left, as described in Sec. 5.1. Crucially, the latter accounts for almost the same fraction of energy consumption as local processing, thus providing a quantitative indication for Lesson 4 of Sec. 5.2. Empire shifts this energy budget to other functionality, as our hardware design offers a way for the software to shutdown the device when the current workload completes. This testifies that Lesson 6, 7 and 8 in Sec. 6.2 are key to achieve better energy management.

Fig. 15 examines data yield, plotting the amount of sensed data that reaches the end user using either of the energy-harvesting systems, normalized to the monthly performance of Kingdom throughout the deployment. Even though the data yield is slightly higher in Empire due to better energy management of T/H devices, the improved design in Empire is not meant to increase data yield, but to repurpose available energy to capture more data that describes events of interest. In contrast, inefficient energy management in Republic makes it unable to capture humidity and temperature readings faithfully, as less data reported to the sink means certain trends in humidity or temperature are missed.

Fig. 16 corroborates this reasoning by showing how Republic provides values close to Kingdom at sunrise or sunset, that is, when the heat flux from *Circus Maximus* to/from *Mithræum* is larger and thus the TEG provides more energy. We find that the number of times a T/H device activates in these periods is roughly equal for Republic and Empire. At times when the heat flux is lower, the measure obtained with Republic appear to deviate from Kingdom. Our logs indicate that the T/H devices in Republic activate about 32% fewer times in these periods compared to Empire. The latter makes better use of the lower energy available at these times, reporting as valuable information as the battery-operated system.

This performance is enabled by accurate energy management at T/H devices in Empire by *i)* programmatically configuring the energy available at the next device activation, and *ii)* enforcing a device shutdown whenever the current workload is accomplished.

I/A devices. Different from T/H devices, Fig. 14 shows that the impact of using low-power modes at I/A devices is minimal. This is consistent with Fig. 15, which shows that the monthly data yield for I/A devices is generally the same for Republic and Empire, and anyways about one fourth of what the battery-operated system can collect. In this case, the nature of the data matters.

In Kingdom, the amount of data collected is sufficient to comprehensively describe the signal features. As anticipated, no structural analysis is possible using Republic, in that acceleration data is unsuitable for modal analysis [43]. The acceleration data we gather with Republic, while being the same as Empire in net amount, is of a different nature: it poorly describes the signal features. Unlike the functioning of Empire, I/A devices in Republic may activate at times where no specific vibration of interest takes place. This substantiates our claims in Lesson 5 of Sec. 5.2.

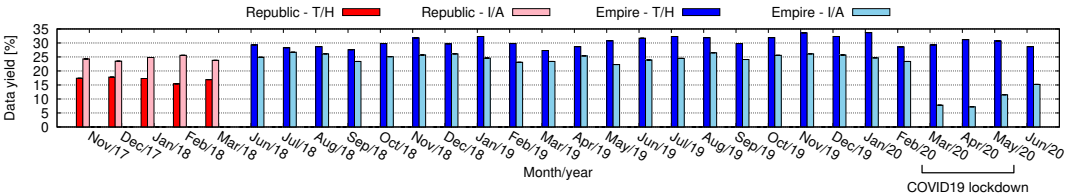


Figure 15: Monthly data yield of Republic or Empire normalized to that of Kingdom throughout the deployment duration. *For T/H devices, Republic provides a lower data yield than Empire because of less accurate management of energy. Data yield for I/A devices in Empire lowers during the COVID19 lockdown because of reduced vibrations useful for energy harvesting.*

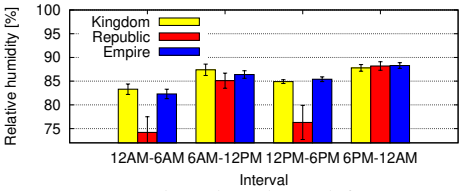


Figure 16: Average relative humidity at different times of a day, recorded by Republic and Empire compared to Kingdom throughout the system lifetime. *Less accurate energy management in Republic becomes apparent in periods of energy scarcity.*

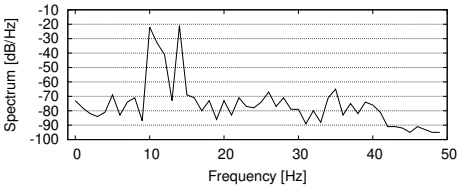


Figure 17: Spectral density at I/A device #6 using Empire during May 11th earthquake in Rome. *The data comes in a period where Empire experiences very little energy intake, due to the ongoing COVID19 lockdown. The Richter magnitude from the acceleration data we obtain is* 3.14*, close to the official* (3.2, 3.7) *estimates [29].*

Conversely, the ability of Empire to activate when a relevant phenomenon occurs counterbalances the smaller amount of collected data. Fig. 12 and Fig. 13, for example, demonstrate that the structural analysis obtained using Kingdom or Empire is largely equivalent, as the outputs are quite similar in absolute value and variability. This performance is enabled by our design of I/A devices, including *i)* the use of a secondary piezo element to activate the device upon detecting vibrations of interest, crucially based on Lesson 9 in Sec. 6.2, and *ii)* the 2-bit "energy level" input line that enables energy-aware scheduling of tasks.

7.3 Maintenance

We record a partial (total) failure of Kingdom as the point in time when at least one (all) device(s) stop operating. The duration of a failure is the time between when the failure is recorded first, until it is resolved through one or multiple site visits.

Batteries. By subtracting the time of partial or total failures from the deployment duration, we find that the uptime of Kingdom is roughly 71%. This notably includes a period of almost three months, shown on the right in Fig. 2, where a partial failure on March 3rd, 2020—eventually turned into a total failure—was impossible to resolve promptly as the city of Rome, as much as the entire country, is under a lockdown due to COVID19 [88]. During this period, citizens mobility was limited to the bare essential for one's sustenance and access to health services. As restrictions are progressively lifted in late May 2020, we access the site for the required maintenance.

Besides two failures in the cellular connection at the sink, shown in Fig. 2, all failures in Kingdom are due to battery problems. Each failure requires accessing the site for maintenance, including obtaining authorizations from the municipality, scheduling an appointment with the accompanying officer based on her availability and not overlapping with other people's visits, accessing the site for the time required, performing the maintenance work, and rebooting the system. We estimate this effort to be around six person-month until the time of writing. On a yearly basis, this equals the effort for the initial development of the system, discussed in Sec. 4.2; throughout the duration of the deployment, the maintenance effort is right now more than twice the development one.

Zero maintenance. We cannot similarly indicate a measure of system uptime for the energy-harvesting ones, due to the lack of ground truth on the availability of ambient energy. However, we can offer ultimate evidence of the *zero-maintenance* operation of Empire. We not only do not touch the system other than the initial installation, but during the lockdown we are *prevented* to do so. The lockdown is, nonetheless, apparent in the data yield of I/A devices in Fig. 15. As activities at the Opera Theater workshop come to a halt and vehicular traffic greatly lowers, vibrations useful for energy harvesting sharply reduce. The system then provides less acceleration data to end users during March, April, and May 2020.

Nonetheless, capturing the coupling between energy sources and sensed data, as per Lesson 9 in Sec. 6.2, allows the I/A devices in Empire to become operational when needed, even during the lockdown. On May 11th, 2020 at 3.03AM UTC, an earthquake of moderate intensity hits the area north of Rome [29]. The I/A devices in Empire are activated by the trigger piezo, while they accumulate enough energy to execute at least the sensing task. Fig. 17 shows the spectral density of the acceleration signal we eventually receive. The sharp difference compared to Fig. 13 testifies the different nature of the vibration, with two peaks at frequencies much higher than those in normal circumstances, as explained in Sec. 7.1. Kingdom is,

on the other hand, hopelessly down since the beginning of March and thus unable to provide any data.

Using existing computational methods [49], we estimate the Richter magnitude of the earthquake from spectral density and Fast Fourier transform of the signal we collect. We obtain a value of 3.14, close to the official estimates [29] of the Italian Institutes of Geophysics that report a (3.2, 3.7) interval, obtained using numerous professional seismographs around Rome. Our estimate, in contrast, is obtained using an energy-harvesting embedded sensing device that operates with *zero maintenance* since almost two years.

8 KEY TAKE-AWAYS

We articulate how the insights we gain through our specific experience may serve to other system builders and seed new directions. Our primary message is that, in situations of energy scarcity like ours, generality in concrete implementations is a luxury one cannot afford. Different than existing literature that seeks generality in *both* concepts and concrete implementations, our experience motivates developing *general concepts* supported by *application-* or even *deployment-specific implementations*.

Evidence of our reasoning is found on the hardware side, where existing works that focus on accurate energy management [23, 36, 40] largely trade generality for overhead. The generic implementation of the federated energy architecture concept in the Flicker platform [40], for example, costs 10.24μA in device quiescent current: almost twice what we have for T/H devices in Empire. Similar observations apply to Capybara [23] and DEBS [36], both proposing useful concepts coupled with general-purpose implementations whose overhead, in settings akin to ours, are hardly tolerable.

Existing programming techniques largely seek independence from energy patterns and hardware platforms. Most task-based solutions, in particular, adopt a pure software approach [22, 57, 59, 62, 89]. In contrast, our design of Empire fundamentally builds upon Lesson 6, 7, and 8 in Sec. 6.2, as

1) the decision on what task to execute is taken not just based on the availability of input data [22, 57, 59], but also on whether sufficient energy is available; this information is known beforehand in Empire, as it is proactively provided by the power sub-system (I/A devices) or the MCU configures the activation threshold at end of the previous activation cycle (T/H devices).
2) available energy at the start of an active cycle matches the energy demands of a defined subset of tasks, and little to no energy is harvested during an active cycle; as a result, techniques such as two-phase commit of task outputs [59], run-time energy events [89], or task splitting [62] are an unnecessary overhead: if we schedule a task to start, we know it completes successfully.
3) tasks are decoupled and only connected by variable-sized data pipelines; therefore, there is no strict ordering of task executions to be guaranteed [57], neither there are relative timing constraints on their execution [42], as long as the transmission task does not starve, no buffer overflows occur.
4) partitioning the application in tasks explicates the relation between functionality and required peripherals; as a consequence, general solutions for intermittent peripheral operations become unnecessary [6, 13, 16, 23], as every task knows what peripherals it needs and only (re-)initializes those.

In general, we argue that programming techniques must not be oblivious to everything outside software, as long as a proper hardware abstraction layer is defined. Core to this is energy management, both for informing the computing core on its availability and to give the latter the knobs to exert some control on it.

Our arguments do not entail that work in this area is necessarily bound to a narrow scope. One may argue, for example, that our design in Empire is enabled by a priori knowledge of energy demands, which is generally not available and may change at run-time. Tools and techniques to accurately gain this information are, however, emerging [3]. Moreover, as we learn from Lesson 3 in Sec. 5.2, peripherals makes the case of varying run-time energy demands a rare, and often remediable issue, as they dominate the energy figure in our deployment as well as in many others [20, 21, 45, 83]. Should peripherals be used based on run-time information, our design is applicable by scaling down the granularity of individual functionality to the level of single peripheral operation [36].

We thus advocate that our experience be a basis to develop general concepts, backed by *(semi-)automatic methods* to synthesize application-specific implementations *across hardware and software*. For example, the concept of energy buffering for T/H devices in Empire, while similar to Capybara [23] and DEBS [36] that only offer generic implementations, currently has no way to be instantiated with little effort for a different application. Enabling a form of (semi-)automatic generation of hardware/software designs may reap the best of both general concepts and efficient implementations.

9 CONCLUSION

We presented the design and evaluation of a 3.5-year embedded sensing deployment at the UNESCO-protected *Mithræum of Circus Maximus* in Rome, Italy. Besides serving the end users, the effort was an opportunity to assess the state of energy harvesting embedded sensing. We did so through three design iterations.

In our Kingdom design, we find that battery-powered embedded sensing still suffers from the hectic performance of batteries. In our Republic design, we realize that using energy harvesting as a replacement for batteries is not as easy in an energy-scarce setting, mainly due to the lack of complete system solutions. In contrast, a dedicated hardware/software co-design in Empire achieves better utility for data, bringing it back to the level of a battery-powered system. Empire also shows that accurate *zero-maintenance* embedded sensing is possible in a demanding setting. While Kingdom is down since 2+ months due to battery depletion during a COVID19 lockdown, Empire accurately captures a earthquake after almost 2 years of unattended operation. Our 3.14 estimates of Richter magnitude, obtained from acceleration data collected by Empire, is remarkably close to the official (3.2, 3.7) estimate [29].

The hardware schematics and application code for the three design iterations are available [2] for the community to build on.

Acknowledgments. We thank the shepherd and reviewers for the feedback received on the initial submission. This work was supported partly by the Google Faculty Award programme and by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] J. Adkins, B. Ghena, N. Jackson, P. Pannuto, S. Rohrer, B. Campbell, and P. Dutta. 2018. The Signpost Platform for City-Scale Sensing. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[2] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. [n.d.]. Battery-less Zero-maintenance Embedded Sensing at the Mithræum of Circus Maximus: Hardware Schematics and Source Code. https://www.neslab.it/mitreo

[3] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[4] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[5] ANSI/ASHRAE. [n.d.]. Standard 55 - Thermal Conditions for Human Comfort. Retrieved July 10th, 2020 from https://www.ashrae.org/technical-resources/55

[6] A. R. Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* (2018).

[7] N. Baccour, A. Koubâa, L. Mottola, M. Zúñiga, H. Youssef, C. Boano, and M. Alves. 2012. Radio Link Quality Estimation in Wireless Sensor Networks: A Survey. *ACM Transactions on Sensor Networks (TOSN)* 8, 4 (2012).

[8] D. Balsamo, A. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Graceful Performance Modulation for Power-Neutral Transient Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[9] D. Balsamo, B. J. Fletcher, A. S. Weddell, G. Karatziolas, B. M. Al-Hashimi, and G. V. Merrett. 2019. Momentum: Power-Neutral Performance Scaling with Intrinsic MPPT for Energy Harvesting Computing Systems. *ACM Transactions on Embedded Computing Systems* (2019).

[10] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[11] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[12] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. 2008. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[13] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (2018).

[14] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[15] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[16] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SENSYS)*.

[17] D. Carlson, J. Gupchup, R. Fatland, and A. Terzis. 2010. K2: A System for Campaign Deployments of Wireless Sensor Networks. (2010).

[18] M. Ceriotti, M. Corrà, L. D'Orazio, R. Doriguzzi, D. Facchin, G. P. Jesi, R. L. Cigno, L. Mottola, A. L. Murphy, M. Pescalli, et al. 2011. Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*.

[19] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corrà, M. Pozzi, D. Zonta, and P. Zanon. 2009. Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*.

[20] Q. Chen, Y. Liu, G. Liu, Q. Yang, X. Shi, H. Gao, L. Su, and Q. Li. 2017. Harvest Energy from the Water: A Self-Sustained Wireless Water Quality Sensing System. *ACM Transactions on Embedded Computing Systems* (2017).

[21] H. Chiang, J. Hong, K. Kiningham, L. Riliskis, P. Levis, and M. Horowitz. 2018. Tethys: Collecting Sensor Data without Infrastructure or Trust. In *Proceedings of the 3rd IEEE/ACM International Conference on Internet-of-Things Design and Implementation (IoTDI)*.

[22] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[23] A. Colin, E. Ruppel, and B. Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-Harvesting Devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] P. Corke, P. Valencia, P. Sikka, T. Wark, and L. Overs. 2007. Long-Duration Solar-Powered Wireless Sensor Networks. In *Proceedings of the 4th Workshop on Embedded Networked Sensors (EMNETS)*.

[25] Datasheet. [n.d.]. ChipCon 1101. Retrieved July 10th, 2020 from https://www.ti.com/lit/ds/symlink/cc1101.pdf

[26] J. de Winkel, C. Delle Donne, K. S. Yildirim, P. Pawelczak, and J. Hester. 2020. Reliable Timekeeping for Intermittent Computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. 2005. Design of a Wireless Sensor Network Platform for Detecting Rare, Random, and Ephemeral Events. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*.

[28] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. 2006. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN)*.

[29] Istituto Nazionale Geofisica e Vulcanologia. [n.d.]. Earthquake Data in Italy. Retrieved July 10th, 2020 from http://cnt.rm.ingv.it

[30] ReVibe Energy. [n.d.]. modelD Piezoelectric Energy Harvester. Retrieved July 8th, 2020 from https://revibeenergy.com/modeld/

[31] ReVibe Energy. [n.d.]. modelQ Piezoelectric Energy Harvester. Retrieved July 8th, 2020 from https://revibeenergy.com/modelq/

[32] V. L. Erickson, S. Achleitner, and A. E. Cerpa. 2013. POEM: Power-Efficient Occupancy-Based Energy Management System. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks (IPSN)*.

[33] B. J. Fletcher, D. Balsamo, and G. V. Merrett. 2017. Power Neutral Performance Scaling for Energy Harvesting MP-SoCs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*.

[34] F. Fraternali, B. Balaji, Y. Agarwal, L. Benini, and R. Gupta. 2018. Pible: Battery-Free Mote for Perpetual Indoor BLE Applications. In *Proceedings of the 5th Conference on Systems for Built Environments (BUILDSYS)*.

[35] M. Furlong, J. Hester, K. Storer, and J. Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSYS)*.

[36] A. Gomez, L. Sigrist, M. Magno, L. Benini, and L. Thiele. 2016. Dynamic Energy Burst Scaling for Transiently Powered Systems. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE)*.

[37] A. Gomez, L. Sigrist, T. Schalch, L. Benini, and L. Thiele. 2017. Efficient, Long-Term Logging of Rich Data Sensors Using Transient Sensor Nodes. *ACM Transactions on Embeddded Computing Systems* (2017).

[38] M. Guarducci. 2015. Ricordo della Magia in un Graffito del Mitreo del Circo Massimo. In *Mysteria Mithrae.* In Italian.

[39] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[40] J. Hester and J. Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[41] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[42] J. Hester, K. Storer, and J. Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[43] R. C. Hibbeler and T. Kiang. 2015. *Structural analysis.* Pearson Prentice Hall Upper Saddle River.

[44] T. W. Hnat, V. Srinivasan, J. Lu, T. I. Sookoor, R. Dawson, J. Stankovic, and K. Whitehouse. 2011. The Hitchhiker's Guide to Successful Residential Sensing Deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.

[45] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[46] N. Jackson, J. Adkins, and P. Dutta. 2019. Capacity over Capacitance for Reliable Energy Harvesting Sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks (IPSN)*.

[47] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[48] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Transactions on Embedded Computing Systems* (2017).

[49] C. Kircher, A. Nassar, O. Kustu, and W. Holmes. 1997. Development of building damage functions for earthquake loss estimation. *Earthquake spectra* 13, 4 (1997).

[50] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawelczak. 2020. Time-Sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[51] T. T. Lai, W. Chen, K. Li, P. Huang, and H. Chu. 2012. TriopusNet: Automating wireless sensor network deployment and replacement in pipeline monitoring. In *Proceedings of the 11th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[52] H. N. Lechtman and L. W. Hobbs. 1987. Roman concrete and the Roman architectural revolution. In *High-Technology Ceramics: Past, Present, and Future-The Nature of Innovation and Change in Ceramic Technology*.

[53] E. A. Lee and S. A Seshia. 2016. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press.

[54] Libelium. [n.d.]. Waspmote. Retrieved July 10th, 2020 from http://www.libelium.com/products/waspmote/

[55] G. Loubet, A. Takacs, and D. Dragomirescu. 2019. Implementation of a Battery-Free Wireless Sensor for Cyber-Physical Systems Dedicated to Structural Health Monitoring Applications. *IEEE Access* (2019).

[56] B. Lubelli, R.P.J. Van Hees, and C.J.W.P. Groot. 2006. Sodium chloride crystallization in a salt-transporting restoration plaster. *Cement and concrete research* (2006).

[57] B. Lucia and B. Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[58] G. Lukosevicius, A. R. Arreola, and A. S. Weddell. 2017. Using Sleep States to Maximize the Active Time of Transient Computing Systems. In *Proceedings of the ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSYS)*.

[59] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[60] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[61] K. Maeng and B. Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (PLDI)*.

[62] A. Y. Majid, C. Delle Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak. 2020. Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices. *ACM Transactions on Sensor Networks* (2020).

[63] R. Marfievici, P. Corbalán, D. Rojas, A. McGibney, S. Rea, and D. Pesch. 2017. Tales from the C130 Horror Room: A Wireless Sensor Network Story in a Data Center. In *Proceedings of the First ACM International Workshop on the Engineering of Reliable, Robust, and Secure Embedded Wireless Sensing Systems (FAILSAFE)*.

[64] P. Martin, Z. Charbiwala, and M. Srivastava. 2012. DoubleDip: Leveraging Thermoelectric Harvesting for Low Power Monitoring of Sporadic Water Use. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[65] G. V. Merrett and B. M. Al-Hashimi. 2017. Energy-Driven Computing: Rethinking the Design of Energy Harvesting Systems. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*.

[66] L. Mottola, G. P. Picco, M. Ceriotti, S. Guna, and A. L. Murphy. 2010. Not All Wireless Sensor Networks Are Created Equal: A Comparative Study on Tunnels. *ACM Transactions on Sensor Networks* (2010).

[67] F. E. Murphy, E. Popovici, P. Whelan, and M. Magno. 2015. Development of an heterogeneous wireless sensor network for instrumentation and analysis of beehives. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*.

[68] M. Navarro, T. W. Davis, Y. Liang, and X. Liang. 2013. A study of long-term WSN deployment for environmental monitoring. In *Proceedings of the 24th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*.

[69] S. Peng and C. P. Low. 2012. Throughput optimal energy neutral management for energy harvesting wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*.

[70] A. I. Petrariu, A. Lavric, and E. Coca. 2019. Renewable Energy Powered LoRa-based IoT Multi Sensor Node. In *Proceedings of the 25th IEEE International Symposium for Design and Technology in Electronic Packaging (SIITME)*.

[71] Piezo.com. [n.d.]. Q220-H4BR-2513YB piezoelectric bending transducer. Retrieved July 8th, 2020 from https://piezo.com/products/piezoelectric-bending-transducer-q220-h4br-2513yb

[72] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[73] A. Rodriguez, D. Balsamo, Z. Luo, S. P. Beeby, G. V. Merrett, and A. S. Weddell. 2017. Intermittently-powered energy harvesting step counter for fitness tracking. In *Proceedings of the IEEE Sensors Applications Symposium (SAS)*.

[74] E. Ruppel and B. Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-harvesting Computing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[75] M. M. Sandhu, K. Geissdoerfer, S. Khalifa, R. Jurdak, M. Portmann, and B. Kusy. 2020. Towards Optimal Kinetic Energy Harvesting for the Batteryless IoT. *arXiv preprint arXiv:2002.08887* (2020).

[76] N. Saoda and B. Campbell. 2019. No Batteries Needed: Providing Physical Context with Energy-Harvesting Beacons. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSSYS)*.

[77] U. Senkans, D. Balsamo, T. D. Verykios, and G. V. Merrett. 2017. Applications of Energy-Driven Computing: A Transiently-Powered Wireless Cycle Computer. In *Proceedings of the 5th ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSYS)*.

[78] V. Sharma, U. Mukherji, V. Joseph, and S. Gupta. 2010. Optimal energy management policies for energy harvesting sensor nodes. *IEEE Transactions on Wireless Communications* (2010).

[79] L. Spadaro, M. Magno, and L. Benini. 2016. Poster Abstract: KinetiSee - A Perpetual Wearable Camera Acquisition System with a Kinetic Harvester. In *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[80] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. 2004. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SENSYS)*.

[81] C. Tavolieri and P. Ciafardoni. 2010. Mithra. Un viaggio dall'Oriente a Roma: l'esempio del Mitreo del Circo Massimo. *Archaeology Archives, BA* (2010). In Italian.

[82] Thermalforce. [n.d.]. 254-150-36 TEG. Retrieved July 10th, 2020 from https://www.dropbox.com/s/4xx1z2gwddntc42/TG254-150-36l.pdf?dl=0

[83] M. Thielen, L. Sigrist, M. Magno, C. Hierold, and L. Benini. 2017. Human body heat for powering wearable devices: From thermal energy to application. *Energy conversion and management* (2017).

[84] UNESCO. [n.d.]. Heritage Site Rome. Retrieved July 10th, 2020 from https://whc.unesco.org/en/list/91/

[85] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[86] Y. Wang. 2008. Topology Control for Wireless Sensor Networks. In *Wireless sensor networks and applications*. Springer.

[87] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. 2006. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[88] Wikipedia. [n.d.]. COVID-19 pandemic lockdown in Italy. Retrieved July 10th, 2020 from https://en.wikipedia.org/wiki/COVID-19_pandemic_lockdown_in_Italy

[89] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.

[90] J. Zhang, C. Chen, X. Zhang, and S. Liu. 2016. Study on the environmental risk assessment of batteries. *Procedia Environmental Sciences* (2016).

Discovering the Hidden Anomalies of Intermittent Computing

Andrea Maioli*, Luca Mottola*†, Muhammad Hamad Alizai+, Junaid Haroon Siddiqui+
*Politecnico di Milano (Italy), †RI.SE (Sweden), +LUMS (Pakistan)
Contact e-mail: andrea1.maioli@polimi.it

Abstract

Energy harvesting battery-less embedded devices compute *intermittently*, as energy is available. Intermittent executions may differ from continuous ones due to repeated executions of non-idempotent code. This anomaly is normally recognized as a "bug" and solutions exist to retain equivalence between intermittent and continuous executions. We argue that our current understanding of these "bugs" is limited. We address this issue by devising techniques to comprehensively identify where and how intermittent and continuous executions possibly differ and by implementing them in ScEpTIC: a code analysis tool for intermittent programs. Thereby, we find execution anomalies and their manifested impact on program behavior in ways previously not considered. This analysis is enabled by ScEpTIC design, implementation, and performance. ScEpTIC runs up to *ten orders of magnitude* faster than the baselines we consider, enabling many types of analyses that would be otherwise impractical.

1 Introduction

Energy harvesting is enabling a battery-less Internet of Things (IoT) of resource-constrained devices with small form factors [17, 34, 35, 39]. However, energy supply from the environment is generally erratic, causing frequent and unanticipated device shutdowns. For example, harvesting ambient RF energy for the execution of a simple CRC calculation leads to 16 power failures over a 6 seconds period [32, 6]. Executions thus become *intermittent*, as they consist of intervals of active computation interleaved by periods of recharging energy buffers.

Existing systems rely on small capacitors as energy buffers and on persistent state to ensure forward progress. Many solutions target mixed-volatile platforms, which facilitate handling persistent state as they map slices of the address space to non-volatile memory (NVM) [38, 20, 23].

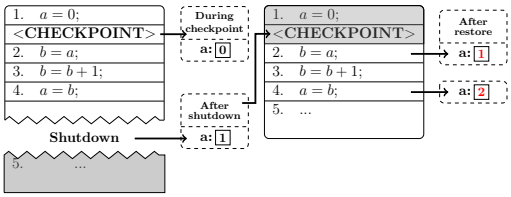


Fig. 1: Ther re-execution of line 2 incorrectly updates variable a allocated on NVM, leading to a memory anomaly.

Explicit *checkpoints* create persistent duplicates of volatile data, including registers and program counter.

Intermittent executions on mixed-volatile platforms introduce the possibility of *execution anomalies* [9, 31, 23, 38, 29], where programs reach states unattainable in a continuous execution. Anomalies may, for example, occur in *memory* due to hazardous read/write patterns caused by the re-execution of non-idempotent code. Fig. 1 shows an example. Variable a is allocated on NVM. A checkpoint occurs after line 1. Lines 2 to 4 eventually modify the value of a. The execution continues until power fails. When energy is back, the execution resumes with the state of volatile data from the checkpoint, that is, it restarts from line 2. However, a being on NVM, it retains its value from line 4 *before* the power failure, that is, the value produced by a later instruction compared to where execution resumes *after* the power failure [31]. Lines 2 to 4 increment a again, producing a different result than a continuous execution.

As we elaborate in Sec. 2, this type of memory anomaly is caused by a specific pattern of `load-store` memory accesses that creates a write-after-read (WAR) hazard. This anomaly is arguably the only one the literature distinctly acknowledges [38, 25, 23, 10, 24, 29]. Existing solutions remedy the problem with custom programming abstractions or compile-time techniques to retain equivalence between intermittent and continuous executions [38, 25, 23, 10, 24]. A few efforts also exist that aim to locate these anomalies and to provide guidelines to programmers for refactoring code [29].

We aim at gaining a deeper understanding of how intermittence affects program behavior. In Sec. 3, we describe techniques to exhaustively check the presence of memory anomalies. Using ScEpTIC, we demonstrate that intermittent programs are vulnerable to a wide variety of memory

International Conference on Embedded Wireless Systems and Networks (EWSN) 2021
17–19 February, Delft, The Netherlands
ISBN: 978-0-9949886-5-2

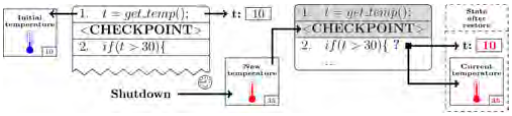


Fig. 2: Accessing a stale temperature reading.

anomalies, beyond those the literature commonly considers. The anomalies we recognize still originate from WAR hazards, and possibly manifest from a disparate set of memory access instructions, such as stack `push-pop` and function `call-ret`. Little discussion exists on these issues [29].

Execution anomalies due to *environment interactions* are also possible and generally harder to ascertain [8, 2, 5]. A power failure may cause programs to process stale environment state, such as an old sensor reading, or to perform unexpected actions on the environment, such as wrongly moving a rotor multiple times. Fig. 2 shows an example. Suppose a system suffers an unpredictably long power failure immediately after the execution of line 2. When the system resumes, the temperature might have changed, but the `if`-condition still evaluates to false with the old value of `t`. However, we can construct a different example where stale data may be valuable; for example, to compute long-term averages.

Determining whether and how execution anomalies affect environment interactions requires analyzing the causal impact of intermittence on the latter. The literature currently lacks the concepts and tools for this. In Sec. 4, we describe custom abstractions to qualify different types of environment interactions and implement proper support in ScEpTIC. Our tool tracks accesses to memory locations of interest and recognizes when a program is vulnerable to processing stale environment data. ScEpTIC also keeps track of the evolution of the environment state, for example, as determined by actuation, and determines if repeated executions of certain output actions can produce undesirable states.

Efficiently enabling the required code analysis is a challenge. A static analysis of the program would not provide run-time information required for analyzing the memory and environment. Checking actual executions in principle requires to analyze any possible combination of checkpoint placement and number of (re-executed) instructions. Running programs on target hardware is therefore plainly impractical, whereas source-level simulation may miss relevant read/write patterns that only manifest in machine code.

It would then appear that machine-level emulation is the only viable choice. That is, however, likely inefficient. A simple *CRC* computation [38] includes $5 \cdot 10^4$ machine-code instructions. If we were to test all possible combinations of checkpoint placement and number of (re-executed) instructions, we would need to analyze $2.34 \cdot 10^{13}$ machine-code instructions. As an example, our prototype emulator runs $5 \cdot 10^4$ instructions per second on a modern PC, which would mean 14 years for testing *CRC* computation.

Our tool ScEpTIC, described in Sec. 5, makes analysis of intermittent programs practical. ScEpTIC helps both system designers and developers analyze various programs, memory configurations, and forward progress mechanisms, to identify the most efficient configurations. System designers may also rely on ScEpTIC to evaluate different strategies for their forward progress mechanisms, which may be tuned accordingly to ScEpTIC results. Our tool is based on custom techniques we devise for analyzing memory anomalies as well as environment interactions. It takes LLVM intermediate-representation (IR) instructions as input to retain platform independence, and captures *all* occurrences of program anomalies due to intermittence, the conditions that cause them, and the effects they bear on program behavior.

In Sec. 6, we quantify the performance of ScEpTIC across different benchmarks and memory configurations. We compare ScEpTIC against a baseline that applies a *brute-force* approach to exhaustively analyze any possible intermittent execution as a function of checkpoint placement, interaction with the environment, and point of power failure. We show that ScEpTIC is up to *ten orders of magnitudes* faster. This means returning the results of code analysis in a matter of minutes rather than hundreds of days, enabling many types of investigations that would be otherwise impractical.

We end the paper in Sec. 7 with brief concluding remarks. ScEpTIC is available as open-source software [28].

2 Background and Related Work

We provide here the necessary background and an account of related work.

Mixed-volatile platforms. Low-power microcontroller units (MCUs) normally employ traditional SRAM as main memory. Thus, power failures cause a complete loss of state.

Frequent power failures motivate the design and manufacturing of mixed-volatile MCUs [30], where slices of the address space map to non-volatile memory facilities, such as FRAM. Data mapped to FRAM do not need to be checkpointed, as they are already persistent, thus sparing the corresponding overhead. This comes at the expense of increased energy consumption and slower memory access during normal operation [16]. FRAM-equipped MSP430 MCUs, for example, increase energy consumption by 2-3$\times$ compared to their volatile memory counterparts, and the MCU may only operate up to half of the maximum frequency without introducing waiting states to synchronize memory accesses [30].

Most importantly, registers and program counter, in addition to any volatile slice of main memory, need to be checkpointed anyways. The dichotomy between non-volatile and volatile memory spaces creates many of the issues we tackle.

Forward progress. Existing checkpoint systems focus on striking a trade-off between postponing the checkpoint; for example, to leverage new ambient energy, and anticipating it to ensure sufficient energy is available to complete it.

For example, Hibernus [3] and Hibernus++ [4] employ specialized hardware support to monitor the energy left. They operate in a *reactive* manner: whenever available energy falls below a threshold, they *react* by firing an interrupt that preempts the application and forces the system to take a checkpoint. Checkpoints may thus take place at *any* arbitrary point along the execution of a program.

Systems such as Mementos [32], HarvOS [7], and Chinchilla [25] employ compile-time strategies to insert specialized system calls to check the energy buffer. These triggers

bind checkpoint operations with a certain condition; for example, a checkpoint is only taken if available energy voltage falls below a threshold. Checkpoints thus happen *proactively* and *only* whenever the execution reaches one of these calls.

A similar duality exists in the solutions available to interact with the environment in intermittent programs, as two approaches exist. The *preventive* method seeks to achieve atomic interactions with the environment, and only initiates them when the remaining energy guarantees completion [21, 12]. Differently, the *recovery* method represents the evolution of peripheral states in main memory to bring the system back to a consistent state when resuming computations [8, 5, 2, 26]. Both of these methods integrate equally well with the techniques we explain next.

Debugging intermittent programs. Tools exist for the general problem of debugging intermittent programs, regardless of execution anomalies. For example, Ekho [15] recreates energy harvesting patterns to enable repeatable in-lab tests. CleanCut [11] identifies *non-termination bugs* in systems using task-based programming with transactional semantics.

Somehow closer to our work are EDB [9] and Siren [14]. In addition to traditional debugging features, EDB can emulate power failures and subsequent reboots. Siren introduces NVM and energy simulation capabilities in MSPSim. Using either tool, one may recognize a subset of the execution anomalies we identify by manually placing breakpoints and resets. This may be extremely laborious without apriori information, for example, a suspect of certain anomalies. Moreover, with Siren breakpoints and resets must be placed at the level of machine instructions and, unlike our work, neither tool provides any automated technique to cover all possible program executions that may manifest anomalies. They also do not consider environment interactions as we do.

Execution anomalies. Ransford and Lucia identify specific instances of memory-related execution anomalies [31]. Their insights provide a foundation for several later works that mask or avoid their occurrence [23, 10, 24, 38].

A specific analysis technique is presented by Van Der Woude et al. [38], who also solely acknowledge the same specific instances. They are, however, unable to assess the actual effects of anomalies and to recognize other instances, such as anomalies occurring on the heap.

Surbatovich et al. [37] identify a subset of the issues we identify for environment interactions. They assume that non-idempotent behaviors due to repeated I/O operations are to avoid, and thus provide a tool that determines the reach of input data through the program so developers fix these behaviors.

Our preliminary work on intermittence anomalies [29] covers only anomalies in main memory and on the stack. We extend our previous contribution with the support for anomalies happening on the heap. Different than our previous contribution, we also provide an analysis of environment interactions, which current literature overlooks, and a quantitative evaluation of our tool's performance.

3 Memory

We present techniques for *locating* memory anomalies and for evaluating their *effects* on program behavior. Both techniques are *sound* and *complete*, namely, they identify all and only the actual cases of memory anomalies.

3.1 Locating Anomalies

In general, memory anomalies due to intermittent executions may occur because of hazardous read/write patterns in NVM and depending on their interleaving with checkpoints.

To locate these anomalies, one should search for the conditions where a checkpoint occurs before a read on NVM, and there exist a write to the same NVM location before a following power failure. If so, a memory anomaly may occur due to WAR hazards, as in Fig. 1. This occurs because of the re-execution of read instructions after resuming, which may cause the program to load a value that was written by a later instruction, but before the power failure.

To be complete in identifying these cases, in principle, one should check all possible combinations of read/write operations on the same NVM address and all possible interleavings with checkpoint locations. For each different setting, one should execute the code for understanding how a given anomaly possibly propagates within the considered execution. As checkpoints might potentially occur at any point in the execution [3, 4], this creates an exponential increase in the number of possible executions that are to be checked, as discussed in the Introduction.

To address this issue, we determine the minimal amount of information necessary for the identification of memory anomalies and devise corresponding analysis techniques. These are based on the crucial observation that if one is only interested in *locating* these anomalies, looking for specific sequences of read/write accesses on NVM in a *single sequential* execution of the code suffices. Depending on the memory segment, these operations may take the form of `load/store`, `push/pop`, or `call/ret` pairs.

Note that we execute the program once to gather information that is usually not avaiable at compile time, such as the address of each accessed memory location, the evaluation of conditional instructions, and the executed branch paths. We then rely on developers to provide a sufficient set of tests that cover all execution paths that are input-dependent.

These techniques and their implementation in ScEpTIC eventually lead us to confirm current findings [31] and to recognize additional memory anomalies.

3.1.1 Data Access Anomaly

Fig. 1 is a case of *data access anomaly*, as reported in literature [31]. We recognize such an anomaly whenever x is a memory address in NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 loads a value from an address x,
- I_n modifies the value stored at address x,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

These conditions entail that if a power failure occurs after I_n, the system resumes before I_1 which is then re-executed; I_1 then reads the value produced by I_n before the power failure, that is, from a later instruction. A fix for this is placing a checkpoint between I_1 and I_n to avoid re-executing the *load* operation when resuming [38].

Here, we reduce the information necessary for locating memory anomalies based on two key observations:

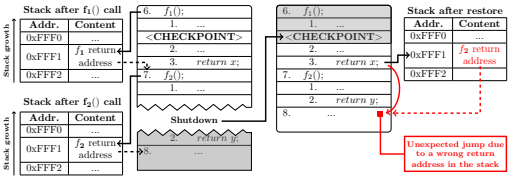


Fig. 3: Return address overwritten by call to f_2 showing an activation record anomaly.

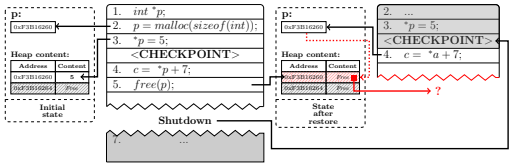


Fig. 4: Example of memory map anomaly.

1. any non-write access after I_1 need not to be checked separately, because the potential memory anomaly it may cause is already captured by the analysis from I_1;
2. only write accesses occurring in the sequence $I_1, ..., I_n$ meet the conditions to produce a WAR hazard; in fact, any other write access that follows a checkpoint after this sequence can not affect prior read accesses, and becomes part of a different sequence $I_{n+1}, ..., I_m$.

These two criteria form the basis to efficiently analyze other, previously unseen kinds of memory anomalies.

3.1.2 *Activation Record Anomaly*

We uncover executions whereby allocating the stack on NVM, upon resuming from a power failure, non-volatile information is read from the activation record of a function to be executed later. This *activation record anomaly* may lead to wrong results, unwanted jumps, or a program crash.

Fig. 3 shows an example. A call to function f_1 executes first and its activation record is placed on the stack. A checkpoint takes place after line 2 inside f_1. When f_1 returns, its activation record pops from the stack and execution continues from line 7. The stack content on NVM is not deleted when returning from f_1; only the stack pointer changes. When placing the activation record of f_2 on the stack, the one of f_1 is overwritten. If a shutdown happens during the execution of f_2, the execution resumes inside f_1 according to the checkpoint data, but the activation record is that of f_2.

Note that Fig. 3 shows the case where the return address from f_2 is read as the one of f_1 when execution resumes. This is only one of the possible outcomes. Worse is if f_2 overwrites f_1 return address with data representing an invalid address, such as a local variable or a saved register, causing a program crash when execution resumes. In general, the sequence of `pop` instruction belonging to the epilogue of f_1 may read the values produced by `push` instructions belonging to the prologue of f_2. Also, note that f_2 may equally be a programmer-defined interrupt handler that fires asynchronously, making the issue even more difficult to track.

We find that an *activation record anomaly* exists whenever the stack is allocated on NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 is a `call` instruction for function f_x,
- the execution of f_x includes at least one checkpoint,
- I_n is a `call` instruction,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

The anomaly exists because a checkpoint is saved inside the context of a function f_1, f_1 returns, and a subsequent call to f_2 overwrites parts of the activation record of f_1. Checkpointing between the return of f_1 and the call to f_2 addresses the issue, preventing the execution from resuming inside f_1.

Here, we reduce the information to check for locating these anomalies by applying the two criteria in Sec. 3.1.1, but also noticing that the analysis need not to consider the code of f_x. Only the fact that f_x somewhere includes a checkpoint matters. We may analyze the code of f_x separately compared to the search of the conditions above, and no information from the analysis at the level of function calls need to percolate into the analysis of callees.

Ratchet [38] identifies a specific instance of the problem arising with interrupts. The general case is, however, overlooked in existing literature and may be recognized only by reasoning at the level of machine code, not source code.

3.1.3 *Memory Map Anomaly*

When read/write instructions on NVM involve operations that possibly change the heap state, a *memory map anomaly* occurs whereby a dynamic memory operation observes a future state of memory upon resuming from a power failure.

Fig. 4 shows an example. Line 2 allocates a heap block and saves its address in pointer p. A checkpoint occurs before line 5, which de-allocates the same memory block. If a shutdown happens after line 5, the execution resumes from line 4, whose memory access may now lead to unpredictable results [40] as the block was previously de-allocated.

It would be possible to construct arbitrary combinations of heap operations before and after a checkpoint, leading to this kind of anomaly. If pointer information are not updated, the re-execution targets the memory address before the shutdown, whereas the memory block may now be freed or re-allocated somewhere else.

We find that a *memory map anomaly* exists whenever the heap is allocated on NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 is a `load` or `store` instruction targeting the heap block pointed by x,
- I_n is a `free` or `realloc` instruction that modifies the heap block pointed by x,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

The anomaly exists because pointer information are not consistent with the state of the heap. Properly placing checkpoints to avoid re-executing instructions based on possibly inconsistent pointer information solves the issue.

Similar to the stack, the two criteria in Sec. 3.1.1 are valid here too to help locate heap anomalies efficiently. In addition, we note how allocating the heap on NVM with a transactional memory controller [36] does not ensure atom-

icity for heap modifications, either. Power failures happening during the execution of any such instructions leave the heap state partially changed. The re-execution of instructions that perform destructive changes to the heap, such as `free` or `realloc`, is also a possible source of anomaly, whereas re-executing memory allocation operations, such as `malloc`, does not affect correctness but may yield memory leaks.

Existing literature overlooks the existence of this kind of anomaly too, which again may only be recognized by reasoning at the level of machine code and raw memory accesses.

3.2 Evaluating Effects

The observations above serve to recognize and locate memory anomalies, but they do not suffice to examine how their effects change the program state compared to a continuous execution. Information on this may be crucial for identifying the cause of a program crash or for performing a post-mortem analysis, as the change of behavior may, for example, corrupt the state in subtle ways and thus percolate throughout possible long-running executions [40].

To this end, a single sequential program execution can only provide partial information. We rather need to emulate the code re-execution, by pretending checkpoints at certain code locations are executed and power failures occur later. We crucially observe that we may use the conditions we identify in Sec. 3.1 also to reduce the number of locations where checkpoints and power failures need to be emulated. In essence, it is sufficient to first locate the anomaly and only then, to re-execute the relevant parts of code.

For example, consider analyzing data access anomalies according to the conditions in Sec. 3.1.1. To understand their effects, we create a new emulated execution starting at I_1 with the state that a continuous execution would have at that point, and proceed up to I_n where we pretend a power failure to happen. Then, we take the state of the NVM there, bring it back to I_1, and combine it with information in the checkpoint that we assume to occur right before I_1. We resume the execution as if the device had new energy and proceed again up to I_n. The program state at this point represents how the data access anomaly alters the program state. Similar techniques are applicable for all memory anomalies in Sec. 3.1.

4 Environment Interactions

Interactions with the environment are a key functionality of embedded sensing devices. As the notion of correctness here is application-specific, understanding how they affect intermittent executions requires to develop both appropriate abstractions and analysis techniques. The problem takes different forms for input (sensing) and output (actuation) interactions. When integrated with approaches to cope with power failures during the interaction itself, as explained in Sec. 2, the techniques we explain next apply to both methods using preventive and recovery techniques.

4.1 Input Interactions

Intermittent executions create a data-time dependency [18]. A piece of urgent data may expire after a long energy outage, requiring the system to sense again before resuming the execution. Old data may still be valuable depending on applications requirements; for example, in applications that are interested in long term trends.

Abstractions. We define two concepts to qualify how, according to the programmers' intentions, input environment data should be accessed in intermittent programs. Under a *most-recent* access model, a program is expected to access the input data only if it is gathered within the same power cycle, that is, no power failure occurs between the time the data is acquired and when it is used. This is the case where applications must take decisions based on the most up-to-date environment data. Differently, under a *long-term* access model, a program may access the input data independent of when it is originally gathered, that is, an arbitrary number of power failures may occur between when the data is acquired and when it is used. This is the case where data is valuable because of its long-term significance.

We ask programmers to tag individual variables storing sensor data as behaving according to either model. The techniques we illustrate next allow programmers to understand whether, depending on checkpoint placement, the semantics of their variables matches the required access model. Note that this analysis is meaningful for system support employing proactive techniques [32, 7, 25], as explained in Sec. 2. Programmers may move the placement of checkpoint calls to ensure that given variables behave according to the desired access model. Differently, in systems employing reactive techniques, checkpoints may happen anywhere in the code [3, 4]. Variables that store sensor data thus behave according to a *long-term* access model, because checkpoints might potentially happen anytime between when the data is gathered and when it is later used.

Analysis. We perform a single sequential execution of the code and use two additional bookkeeping data structures, a *checkpoint clock* and an *access record*. The former establishes an ordering of the events in the code and is incremented each time we pretend a checkpoint to happen. This corresponds to every location in the code where a call to the checkpoint routine is inserted. The access record tracks accesses to memory locations of interest.

We explain the process with the help of Fig. 2. Initially, the *checkpoint clock* is set to 0. After executing line 1, the *access record* for variable t is updated as $\langle t, temperature, 0\rangle$, where variable t contains the value of a given sensor when the checkpoint clock is 0. Next, the *checkpoint clock* is incremented by one as we encounter a further call to the checkpoint routine. Thereafter, the execution of line 2 leads to another update in the *access record* for t with $\langle t, temperature, 1\rangle$. The variable is thus accessed across checkpoint call, and thus behaves according to a *long-term* access model. If the latter differs from the access model the variable is tagged with, a warning is returned.

Note that memory buffers for sensed data on NVM may also suffer from memory anomalies, which can be tested with the techniques described in Sec. 3.

4.2 Output Interactions

Intermittence may cause an application to perform a stale, duplicate, or falsified action on the environment. Fig. 5 shows an example. Line 1 rotates the servo *relatively* by 45°. A power failure occurs right after line 1. When execution resumes from the checkpoint, the servo is rotated again by a

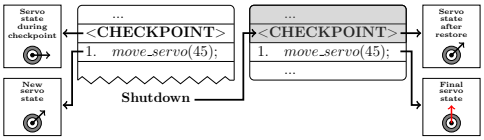


Fig. 5: The re-execution of instruction 1 yields an unexpected environment state.

further 45°, taking its current position to 90°. The outcome does not correspond to a continuous execution.

Abstractions. Similar to Sec. 4.1, we define two semantics for actuation commands: *absolute* or *relative*. The former models the cases of idempotent actuation commands. The latter models the opposite, that is, the resulting state of the environment is a function of the initial state and of actuation.

Application requirements dictate whether to rely on either semantics. Unlike Fig. 5, some applications may want to affect the state of environment whenever an actuation command is executed, regardless of the number of repetitions. Consider, for example, an application that sends an announcement whenever it wakes up from a power failure.

To enable code analysis, programmers are required to express the semantics they expect by tagging the individual calls to actuation commands as behaving according to either model. In addition, they are to provide an abstract specification of how the environment changes in response to the (possibly partial) execution of actuation commands. This specification is primarily meant to check that two environment states are semantically equivalent. This information is, in general, application-dependent. Vast literature exists on the subject [13]. We thus omit the description of such specification for brevity, which is nonetheless available [27].

Analysis. To understand whether intermittent executions match the expected actuation semantics, we execute the program until encountering an actuation command. There, we record the state of environment up to the point of the power failure, either during or after the command execution, according to the abstract environment specification. We re-execute the code from the previous checkpoint up to the same actuation command. We can now compare the new environment state with the previously recorded one.

If the states differ, the command behaves as *relative*, otherwise, it behaves as *absolute*. If this behavior does not match the programmers' expectations, a warning is returned. Programmers may now change the implementation according to application requirements. For example, they may replace an actuation command exposing a *relative* semantics with one implementing an *absolute* one, or build a wrapper around the former to achieve the desired behavior.

5 Implementation

We implement the techniques in Sec. 3 and Sec. 4 in a tool called ScEpTIC, which works in four different modes:

1. ScEpTIC-Locate performs the analysis to *locate* memory anomalies, as in Sec. 3.1.
2. ScEpTIC-Evaluate performs the analysis to *locate* memory anomalies and to evaluate *their effects*, as in Sec. 3.2.

Table 1: Consumers and producers of memory anomalies.

	Data Access - Sec. 3.1.1	Activation Record - Sec. 3.1.2	Memory Map - Sec. 3.1.3
Consumer	`load`	`ret/pop`	`load, store, realloc, free`
Producer	`store`	`call/push`	`malloc, realloc, free`

3. Environment-Inputs verifies the coherence of *input interactions* with programmer-specified semantics, as discussed in Sec. 4.1.
4. Environment-Outputs behaves symmetrically w.r.t. the previous option for *output interactions*, as discussed in Sec. 4.2.

We describe next the architecture of ScEpTIC and the details of the first two modes. The processing required for the other two options is a minimal variation of the former.

5.1 Architecture

ScEpTIC is written in Python and processes LLVM intermediate representation (IR) code to gain independence from specific platforms. It comprises two main modules: the *abstract-syntax tree (AST) builder* and the *emulator*.

The regular LLVM AST builder is augmented with architecture-specific components such as registers, libraries, and proxies for emulating environment interactions, as described next. The resulting AST is then translated to be executed by the emulator module.

ScEpTIC also allows users to annotate functions used for environment interactions. The annotation takes as input: *i)* the type of interaction, namely, input or output; *ii)* the name of the function that interacts with the environment; *iii)* a list of LLVM IR types, representing the function argument types; and *iv)* the type of return value and a logic to generate such values, for example, a generator function or a statically-defined list of values.

The ScEpTIC emulator models user-specified general registers and special purpose registers that exist on all platforms, such as the program counter (PC) and the stack base pointer (EBP). The emulator divides the available memory into three segments: the global symbol table (GST), the stack, and the heap. The GST segment is further subdivided into volatile and non-volatile regions, placing the global variables according to programmer's requirements.

5.2 Locating Memory Anomalies

We call *producer* (*consumer*) any instruction that alters (accesses) the content of NVM. By generalizing the concepts of Sec. 3, we argue that to locate a memory anomaly, we need to identify an ordered sequence of instructions $I_1, \ldots, I_n$, such that I_1 is a consumer, I_n is a producer, I_1 and I_n operate on the same NVM location, and no checkpoint occurs between I_1 and I_n. The pair $\langle I_1, I_n \rangle$ is the one causing the memory anomaly. Tab. 1 indicates the consumers and producers for the memory anomalies discussed in Sec. 3.

The processing we devise for locating memory anomalies only requires sequentially executing the program and collecting a trace of NVM memory states. To create the trace, when a producer or consumer is encountered, we save the state of the target memory locations, the value of an instruction counter that corresponds to its execution, the operation type among those of Tab. 1, and the program counter. We organize this information into a two-level dictionary that has the memory address as first key and the operation type as the

Procedure 1: Locating memory anomalies for a given NVM location.

```
1  function ScEpTIC-Locate (trace, consumer, producer, ED)
2      anomalies ← ∅
3      consumers ← trace[consumer]
4      producers ← trace[producer]
5      foreach pair ⟨counter, consumer_pc⟩ ∈ consumers do
6          window ← SlideWindow(producers, counter, ED)
7          foreach ⟨producer_counter, producer_pc⟩ ∈ window do
8              insert ⟨consumer_pc, producer_pc⟩ into anomalies
9      return anomalies

10 function SlideWindow (producers, consumer_counter, ED)
11     min_counter ← consumer_counter
12     max_counter ← consumer_counter + ED
13     return ∀ producer ∈ producers s.t. min_counter ≤
           Istruction_counter(producer) ≤ max_counter
```

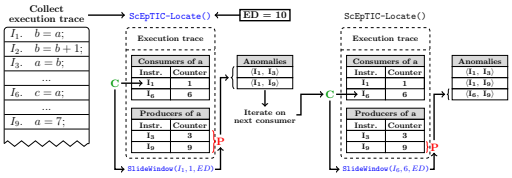


Fig. 6: ScEpTIC-Locate identifies memory anomalies in the execution trace by considering a sliding window of producer instructions altering a given variable.

second one. This allows for an efficient search of memory anomalies when traversing the trace.

Procedure 1 shows the core logic to process the execution trace for a given pair of producer and consumer that possibly cause a memory anomaly. Fig. 6 helps understand the processing with a concrete example. Starting from every consumer, that is, a candidate I_1 that accesses a given memory location (line 5), the procedure operates on a window of producer instructions determined by `SlideWindow` (line 6), with a corresponding instruction counter higher than that of the consumer, and as a function of the checkpoint strategy. In the example of Fig. 6, this extends up to I_9 for consumer I_1. We start from I_1 as we emulate a checkpoint immediately before it; in this case, every producer in the window is a potential I_n that causes a memory anomaly (lines 7-8).

The key for correct and efficient analysis rests in the interplay between `ScEpTIC-Locate` and `SlideWindow`. For the latter, Procedure 1 shows the case of reactive checkpoints, which can potentially happen anywhere in the code. Given the instruction counter corresponding to a consumer operation we consider as the first instruction after a checkpoint, the window extends for a number of instructions whose energy cost equals the energy left after the checkpoint operation and eventually leading to a power failure. These are the instructions that would be possibly re-executed upon resuming. We call this quantity *execution depth* (*ED*)

Actual meaningful values for *ED* depends on the device energy consumption, capacitor size, and checkpoint energy consumption. In Sec. 6.1 we show how to accurately calculate *ED*, as this is essential for obtaining accurate information on intermittence anomalies. Underestimating *ED* may cause the analysis not to identify some intermittence anomaly, whereas overestimating *ED* may cause the analysis to identify bogus intermittence anomalies.

As the analysis of the current window completes, `ScEpTIC-Locate` slides the trace down to the next consumer (line 5), which `SlideWindow` now considers the first instruction executing after a potential checkpoint, that is, a new candidate I_1. In Fig. 6, this happens to be the assignment `c=a`. Note how sliding the instruction window down to any instruction between the former I_1 and the new one does *not* uncover memory anomalies this procedure would not uncover, and thus represents unnecessary overhead. For example, the instructions between `b=a` and `c=a` in Fig. 6 are covered already by the first iteration of the procedure.

The case of proactive checkpoints is a simplified version of Procedure 1. Instead of considering every consumer as the first instruction executed after a potential checkpoint, we simply consider as the candidate I_1 the set of consumer instructions between every statically-inlined checkpoint call and the next producer I_n. Considering consumers I_{n+k}, $k > 0$ past the producer I_n is unnecessary, as they necessarily read the value produced by I_n, as in a continuous execution.

Accordingly, `SlideWindow` now stretches the window of producer instructions from I_1 up to the next statically-inlined checkpoint call, regardless of *ED*. This is the most conservative choice, as it assumes the system has just enough energy to execute every following instruction, but fails to complete the next checkpoint call. The number of possibly re-executed instructions is thus highest. When sliding the window down, `ScEpTIC-Locate` proceeds to the set of consumer instruction after the next checkpoint call, and the procedure repeats.

5.3 Evaluating Effects

The analysis of how memory anomalies possibly impact the program behavior requires the concrete emulation of power failures, with the corresponding code re-execution.

Consider again the case of reactive checkpoints; the case of proactive checkpoints is obtained as a variation of this, similar to Procedure 1. For the analysis to be complete, based on the observations illustrated earlier for locating memory anomalies, it suffices to investigate the case when checkpoints happen before every *consumer* instruction. This is a candidate I_1. The window of instructions that we re-execute extends for *ED* instructions starting from I_1. The instructions that possibly cause the memory anomaly are the producers I_n within this window, whereas the effects of the memory anomaly are manifest, for example, as a consumer instruction I_1 accesses an altered value when it is re-executed upon resuming after a power failure.

Procedure 2 shows the core logic for evaluating the effects of memory anomalies. The example of Fig. 7 helps understand the processing. Before emulating a consumer instruction operating on NVM, we run procedure `ScEpTIC-Evaluate`. It starts off by saving a snapshot of the emulation state, including the current instruction counter (line 2). This information is necessary to roll back the emulated execution to a consistent state in case no memory anomalies are found by re-executing the code from the considered

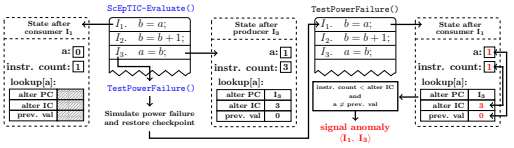


Fig. 7: ScEpTIC-Evaluate procedure to test a checkpoint before instruction I_1.

Procedure 2: Evaluating the effects of memory anomalies for a given NVM location.

```
1  function ScEpTIC-Evaluate (state, checkpoint_data, ED)
2      snapshot ← snapshot of state
3      target_counter ← InstructionCounter(state) + ED
4      lookup ← {}
5      while InstructionCounter(state) < target_counter do
6          pc ← ProgramCounter(state)
7          addr ← TargetAddress(pc)
8          old_content ← NVMContent(state, addr)
9          execute pc
10         if pc is consumer then
11             init lookup[addr] with empty values
12         if pc is producer and addr ∈ keys(lookup) then
13             current_counter ← InstructionCounter(state)
14             lookup[addr] ← ⟨current_counter, old_content, pc⟩
15             if no failure previously simulated after pc then
16                 ⟨state, lookup⟩ ←
                     TestPowerFailure(state, snapshot,
                     checkpoint_data, current_counter, lookup)
17 function TestPowerFailure (state, snapshot, checkpoint_data,
       target_counter, lookup)
18     simulate power failure
19     restore checkpoint_data
20     while InstructionCounter(state) < target_counter do
21         pc ← ProgramCounter(state)
22         execute pc
23         if pc is consumer then
24             addr ← TargetAddress (pc)
25             if addr ∈ keys (lookup) then
26                 val ← NVMContent (state, addr)
27                 ⟨counter2, value2, pc2⟩ ← lookup[address]
28                 if counter2 >
                       InstructionCounter(state) and val2 ≠ val
                       then
29                     signal memory anomaly ⟨pc, pc2⟩
30     if at least one anomaly was found then
31         lookup ← ∅
32         restore snapshot
33     return ⟨state, lookup⟩
```

consumer. Then, it calculates the length of the instruction window to analyze (line 3) and initializes the *lookup* information used for tracking the NVM state (line 4).

For every consumer operation, ScEpTIC-Evaluate initializes the *lookup* information associated to the target addresses (line 6-11). In Fig. 7, this is shown on the leftmost box for variable a. As the execution continues and a producer is found, ScEpTIC-Evaluate verifies if any *lookup* information is present for the target address (line 12). This may entail that an earlier consumer instruction can access an altered information in case of a power failure and subsequent re-execution. If so, we update the *lookup* information of the altered memory location (line 13-14). This is the case for producer I_3 in Fig. 7, as shown in the middle box.

If it is the first time we analyze a specific consumer/producer pair (line 15), we test the effects of a power failure at this point with TestPowerFailure. This resets the volatile state (line 18) and restores the checkpoint (line 19) with the instruction counter at the time of checkpoint. This effectively rolls back the execution to the consumer that triggered the processing, that is, I_1 in Fig. 7. It re-executes the code until it reaches the point of the earlier power failure. Whenever a *consumer* is executed (line 23), TestPowerFailure accesses the *lookup* information to verify if it accesses the value of a producer in the previous power cycle (line 25-29). TestPowerFailure thus identifies a memory anomaly. Fig. 7 shows this happening as soon as I_1 is re-executed, based on the information in the rightmost box.

By continuing the execution, TestPowerFailure assesses the effects that the memory anomaly causes on program behavior, including also other memory anomalies, and up to *ED* instructions from I_1. Upon completion, if any memory anomaly is found, TestPowerFailure restores the snapshot and empties the *lookup* information (line 30-32) before returning control to ScEpTIC-Evaluate. This allows the latter to proceed with the analysis from a clean consistent state, not altered by the effects of memory anomalies.

6 Evaluation

We evaluate our techniques using ScEpTIC on a system with an Intel Xeon E3-1270, 64 Gb of RAM, Ubuntu 19.04, and Python 3.7.2. We use Clang 5.0.1-4 with LLVM 5.0 [22] to produce the LLVM IR [22].

6.1 Memory Anomalies: Setup

We evaluate the performance of the techniques in Sec. 3 by comparing them to a baseline that operates only based on the conditions that possibly lead to a memory anomaly. In contrast, existing forward progress mechanisms [23, 24, 25, 38] avoid the occurrence of intermittence anomalies with analysis techniques that are strictly tied to their system or memory configuration. These may fail to identify the occurrence of anomalies with different memory configurations. As they operate at compile time, they also cannot identify the occurrence of intermittence anomalies that happen across branches, conditional operations, and dynamic memory accesses. Because of this, we use as baseline a "layman" approach that does identify the occurrence of *all* anomalies.

Baseline. Initially, Layman-Memory executes the code sequentially. Every time it needs to analyze a potential checkpoint location, it saves snapshot of the emulation state and then proceeds with the execution. Following the checkpoint location in the code, Layman-Memory records a snapshot of the memory state as the execution unfolds, until it emulates a subsequent power failure. Then, it rolls the execution back to the checkpoint location, emulates a resume operation, and proceeds with the (re-)execution by comparing the snapshots of the memory states produced by the (re-)executed code against those collected earlier.

An anomaly is found whenever a mismatch is detected, as the intermittent execution would be different from the continuous one. Because the comparison occurs on snapshots of the entire memory state, LAYMAN-MEMORY can only provide coarse-grained memory information and cannot pinpoint what instructions are responsible for a given anomaly.

Benchmarks and configurations. We select three benchmarks commonly used in intermittent computing [3, 32, 4, 7, 1]: Cyclic Redundancy Check (CRC) for data integrity, Fast Fourier Transform (FFT) for signal analysis, and Advanced Encryption Standard (AES) for security. They span diverse functionality and expose very different program structures. We use their open-source implementations from MiBench2 [19], that is, a benchmark suite already used for evaluating system support for intermittent computing [38].

For each benchmark, we choose two different memory configurations. One configuration places only *global variables* onto NVM, while allocating all other memory segments, including the stack, on volatile main memory; the other configuration places only the content of the *stack* onto NVM, while allocating all other memory segments, including global data, on volatile main memory.

We consider two different use cases. In the *a-priori* scenario, we use SCEPTIC at a time when the checkpoint strategy is yet to be defined, that is, programmers are to select the most suitable system support. This means checkpoint locations in the code are not known, or reactive checkpoint systems are employed that may preempt the execution at any point in time. For the analysis to be complete in this scenario, every possible checkpoint location should be examined along with any potential location of power failure.

In the *a-posteriori* scenario, we use SCEPTIC when the checkpoint strategy is fixed and checkpoint calls are statically placed in the code. This covers the cases where programmers aim to analyze a specific checkpoint placement [7, 32] or perform a post-mortem analysis of an already-deployed program. We consider the checkpoint placement of Mementos [32]. For AES, we consider both Mementos' *function-return* strategy that positions a checkpoint after the return of each function and its *loop-latch* strategy that places a checkpoint at the end of every loop body. We consider only the latter strategy for the CRC and FFT benchmarks, since they include no significant function calls.

Metrics. The primary performance figure we consider is the net *execution time* required for the analysis, as it determines how practical is a given technique. Moreover, to perform the analysis, a certain technique may need to emulate power failures and possibly re-execute certain instructions, resulting in an increase of the *number of embedded code instructions executed*. We measure this figure as well, as it impacts the execution time. Similarly, the different techniques also collect information about the program state into data structures that are external to the emulated program, so to verify the presence of anomalies. These *accesses to support memory* introduce an overhead that also influences the execution time, and is thus worth measuring as well.

Finally, we compare the number of found memory anomalies by the different systems we test as an indication of the *output noise*. As explained in Sec. 3, SCEPTIC-LOCATE is both sound and complete, and it also returns every memory anomaly as a unique data point, essentially providing the cleanest non-redundant output. Differently, SCEPTIC-EVALUATE may return a higher number of found memory anomalies merely because the effects of the same WAR hazard may differ at run-time based on actual data. We refrain to measure this metric for SCEPTIC-EVALUATE. On the other hand, LAYMAN-MEMORY may point to the same memory anomaly in multiple seemingly different ways, because coarse-grained memory information and the inability to identify the exact instructions responsible for a given anomaly prevents it from filtering out redundant information.

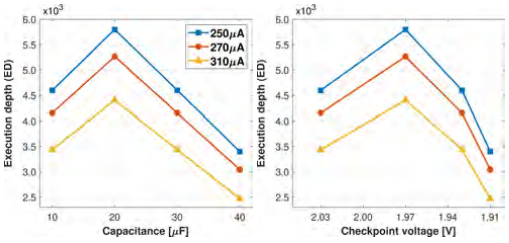


Fig. 8: Execution depth in Hibernus++ [4] with respect to current draw, capacitance, and checkpoint voltage threshold.

Code re-execution. In the *a-priori* use case where checkpoint calls are not statically placed in the code, we need to specify a realistic value for the *ED* parameter, discussed in Sec. 5.2, representing the number of instructions executed after a checkpoint and before the subsequent power failure.

We consider a configuration similar to the one of Hibernus++ [4], using MSP430 [30] MCUs. When the capacitor voltage goes below a certain threshold V_{trig}, Hibernus++ saves a checkpoint. Here, *ED* corresponds to the number of instructions that the MCU executes with the remaining energy, the checkpoint is complete and assuming the ambient provides no additional energy in the meantime.

The capacitor equipping most intermittently-computing systems is an electric bipole characterized by the differential relation

$$i(t) = C\frac{dv(t)}{dt}, \tag{1}$$

where C is the capacitance, $i(t)$ is the current at time t, and $v(t)$ is the capacitor voltage at time t. With a constant current draw, we state

$$\Delta t = C\frac{(V_2 - V_1)}{I} \tag{2}$$

to be the time required to discharge the capacitor from voltage level V_2 to V_1, with a constant current draw I.

Let us consider V_2 to be the voltage level V_{trig} where Hibernus++ [4] triggers a checkpoint and V_1 to be the voltage level V_{min} where the MSP430 powers off. We can express Δt as $t_{chk} + t_{cmp}$, where t_{chk} is the time required for saving a checkpoint and t_{cmp} is the remaining running time of the MCU. From eq (2) we derive

$$t_{cmp} = \Delta t - t_{chk} = C\frac{(V_{trig} - V_{min})}{I} - t_{chk}. \quad (3)$$

Given t_{cmp} as the remaining running time of the MCU, the number of instructions executed within this time when running at a clock frequency f_{MCU} is $ED = t_{cmp} \cdot f_{MCU}$. We can now calculate

$$ED = f_{mcu} \cdot (C\frac{(V_{trig} - V_{min})}{I} - t_{chk}). \quad (4)$$

The MSP430-FR5737 datasheet [30] states that the current draw during the active mode at $1MHz$ goes from $200\mu A$ up to $420\mu A$, depending on cache hit ratio. A group of reasonable values for current consumption is $250\mu A$, $270\mu A$, and $310\mu A$, respectively corresponding to a 75%, 66%, and 50% of cache hit. From Hibernus++ [4] we know that t_{chk} is $1.4ms$ and V_{min} is $1.88V$. Fig. 8 shows the ED that we calculate according to the above derivations, which ranges from 2470 to 5800 instructions. We run our experiments using three representative values for ED: 3000, 4000, and 5000.

Measuring the baseline. LAYMAN-MEMORY must generate an independent test for any possible checkpoint location, which is at any line of code except the last one. For each of these potential checkpoint locations, LAYMAN-MEMORY must simulate a power failure at every instruction that follows the checkpoint within ED following instructions. As a result, the entire analysis for LAYMAN-MEMORY would require years to complete on a standard PC, as we argue earlier.

To obtain a quantitative baseline for comparison, we synthetically calculate the number of instructions executed by LAYMAN-MEMORY for a given benchmark as

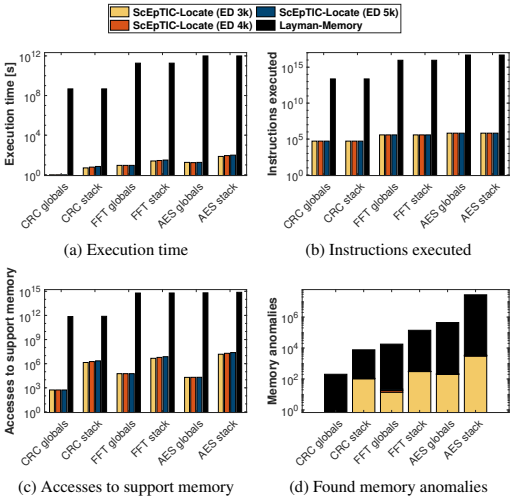
$$(\sum_{i=0}^{n_{ops}-1}(\sum_{j=i+1}^{n_{ops}}(j-i)+1)) - 1, \quad (5)$$

where i represents the checkpoints, j represents the power failures, and n_{ops} is the number of machine instructions in a sequential execution of the same code. The *execution time* for LAYMAN-MEMORY is consequently obtained by considering the emulation speed of SCEPTIC, which runs $5 \cdot 10^4$ instructions per second. With a similar reasoning, we also synthetically calculate the number of accesses to support memory and memory anomalies that LAYMAN-MEMORY finds.

6.2 Memory Anomalies: Results

A-priori scenario. Checkpoint calls are yet to be placed or we are employing reactive system support that potentially triggers checkpoints anywhere in the code. This configuration corresponds to the processing in Procedure 2, where SCEPTIC-EVALUATE stops at the first anomaly found in a given window of instructions, assuming that cascading effects of such anomaly are of no interest. The memory anomaly information that SCEPTIC-EVALUATE provides is thus equivalent to SCEPTIC-LOCATE; we only consider the latter.

Fig. 9 shows the results we obtain. Fig. 9a demonstrates that the execution time of SCEPTIC-LOCATE is 10 orders of magnitude lower than LAYMAN-MEMORY. In absolute terms, SCEPTIC-LOCATE constantly concludes the analysis in practical time across all benchmarks and memory configurations.



(a) Execution time (b) Instructions executed

(c) Accesses to support memory (d) Found memory anomalies

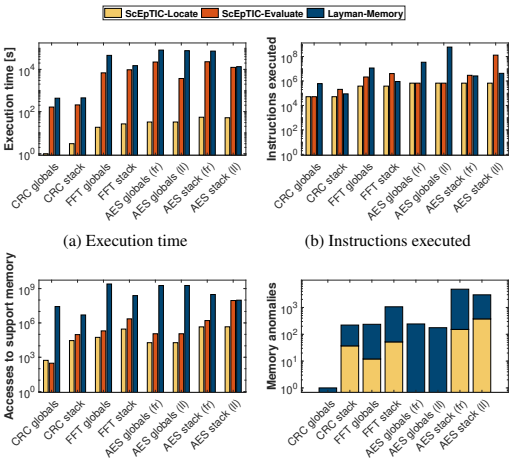
Fig. 9: SCEPTIC-LOCATE is orders of magnitutde faster than LAYMAN-MEMORY in the a-priori scenario. The X axis represents the benchmark and the memory slice on NVM.

Fig. 9a also shows that an increase of ED bears a minimal performance impact on SCEPTIC-LOCATE. The is explained in the results we obtain for the number of embedded code instructions executed and in the number of accesses to support memory, shown in Fig. 9b and Fig. 9c respectively. SCEPTIC-LOCATE executes the program sequentially to collect trace information and later analyzes a sliding window of instructions to locate memory anomalies. Increasing ED therefore does not increase the number of instructions that SCEPTIC-LOCATE executes overall, shown in Fig. 9b. It only slightly increases the accesses to support memory, shown in Fig. 9c.

Fig. 9d shows the number of found memory anomalies to analyze the output noise. LAYMAN-MEMORY returns information on many more memory anomalies due to the coarse-grained information it reasons upon. These anomalies are, however, semantically equivalent to those found by SCEPTIC-LOCATE. Programmers gain no insights from these additional information, which essentially represents a noisy output compared to the programmers' actual needs.

A-posteriori scenario. Fig. 10 shows the results for the *a-posteriori* scenario. For each benchmark, we place checkpoints accordingly to the loop-latch (ll) strategy of Mementos [32]. We also consider the function-return (fr) strategy for the AES benchmark, as it executes a significant number of function calls. Fig. 10a shows SCEPTIC-LOCATE with the lowest execution time. SCEPTIC-LOCATE is on average 3 orders of magnitude faster than LAYMAN-MEMORY and 2 orders of magnitude faster than SCEPTIC-EVALUATE. On the other hand, SCEPTIC-EVALUATE is on average 5 times faster than LAYMAN-MEMORY. LAYMAN-MEMORY has better performance for the AES benchmark as it finds memory anomalies earlier than SCEPTIC-EVALUATE due to the way the code is structured.

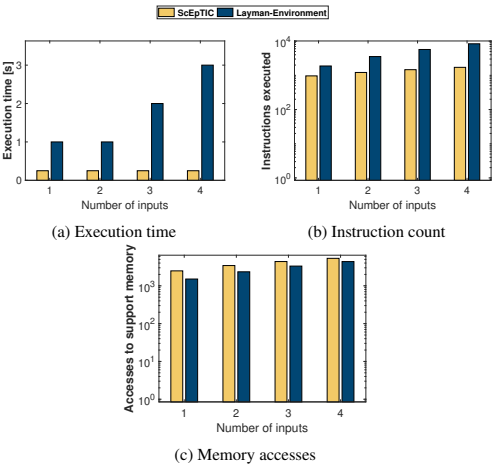
Emulating power failures requires SCEPTIC-EVALUATE and



(a) Execution time (b) Instructions executed

(c) Accesses to support memory (d) Found memory anomalies

Fig. 10: ScEpTIC-Locate is about 3 orders of magnitude faster than Layman-Memory in the a-posteriori scenario.



(a) Execution time (b) Instruction count

(c) Memory accesses

Fig. 11: ScEpTIC is 7x faster in determining the access semantics for environment input variables.

Layman-Memory to save a snapshot of the emulation state at every checkpoint. This is necessary to continue the analysis from a valid state, rather than an inconsistent one, in case any of the re-executed instructions yields a memory anomaly. This causes more accesses to the support memory, shown in Fig. 10c, that make ScEpTIC-Evaluate and Layman-Memory slower than ScEpTIC-Locate even when they execute the same number of instructions. This is the case for CRC and AES with global variables on NVM.

Fig. 10b also shows that when the stack is on NVM, ScEpTIC-Evaluate executes a higher number of instructions compared to Layman-Memory. This is expected, because Layman-Memory cannot pinpoint the instructions that cause a memory anomaly and the ones consuming the altered value. For this reason, it may not simulate power failures for executions where it already recognized a memory anomaly, even though there may be further memory anomalies in the same slice of execution that involve different instructions.

Despite the difference in the number of instructions executed, Fig. 10a shows that ScEpTIC-Evaluate is still faster than Layman-Memory. The reason for this is again in the accesses to support memory, as shown in Fig. 10c: ScEpTIC-Evaluate records only write events and then verifies when a read happens, whereas Layman-Memory compares the entire emulation state with a snapshot, resulting in higher overhead.

6.3 Input Access Analysis: Setup

We evaluate the performance of our analysis of input interactions with the environment, as explained in Sec. 4.1. As the analysis of output interactions uses almost identical techniques, as illustrated in Sec. 4.2, the conclusions we obtain also apply to that. In both cases, the actual procedure we apply is a limited variation of ScEpTIC-Locate.

Similar to Sec. 6.1, we employ a Layman-Environment baseline representative of how one would naturally operate to verify that, in the presence of arbitrary checkpoint operations and power failures, the behavior of a given input variable matches the intended semantics. In essence, this entails determining whether input variables behave according to a *long-term* or *most-recent* semantics against the re-executions of arbitrary code segments.

Benchmarks and configuration. The few works [18, 33, 26, 8, 37] considering intermittent executions together with environment interactions typically employ a typical sense-process-transmit application.

We consider four different input configurations: from one to four environment inputs through corresponding sensors. We use Mementos [32] again as checkpoint mechanism and rely on its manual strategy to keep checkpoint calls balanced with respect to calls to probe sensors. For example, with two inputs we place a checkpoint between their access calls, resulting in the sequence $read1() \rightarrow checkpoint() \rightarrow read2()$. This also corresponds to the approach that existing system support [18, 23, 11] employs to interleave calls to peripherals with checkpoints to ensure atomic execution of individual peripheral interactions.

As the procedure we apply is a variation of ScEpTIC-Locate and the baseline is, in fact, a variation of Layman-Memory, we use the same metrics as in Sec. 6.1. The number of found anomalies is, however, not applicable in this case.

6.4 Input Access Analysis: Results

Fig. 11 shows the results. Both ScEpTIC and Layman-Environment execute the benchmark within seconds, with ScEpTIC performing on average 7 times faster, as Fig. 11a shows. ScEpTIC takes the same time for the execution of the different input configurations, since it executes the program sequentially and requires no re-execution. Instead, the performance of Layman-Environment worsens with the number of inputs, since it needs to re-execute an increasing number of instructions as the number of inputs grows.

This performance reflects in Fig. 11b, where ScEpTIC is

shown to execute almost the same amount of instructions, independent of the number of inputs. The minor increase is merely due to separately processing different inputs. Instead, the number of instructions that LAYMAN-ENVIRONMENT executes increases with the number of inputs present in the benchmark, again because of the increase in the number of re-executed instructions with more inputs.

Fig. 11c accordingly shows that SCEPTIC accesses the support memory 1.4 times more than LAYMAN-ENVIRONMENT on average, as required by the processing applied to the relevant windows of instructions. The overhead that support memory accesses introduced in SCEPTIC is, however, significantly lower than the one of the actual re-execution of code segments in LAYMAN-ENVIRONMENT, ultimately resulting in faster executions for SCEPTIC.

7 Conclusion

Intermittent executions of battery-less embedded devices conceal hidden anomalies whose comprehensive treatment was not addressed by prior work. We fill this gap by investigating the anomalies occurring on memory and through environment interactions. Our contributions are made concrete in SCEPTIC, a code analysis tool for intermittent programs that uncovers previously unknown anomalies. Our evaluation indicates that SCEPTIC is *orders of magnitude* faster than the baselines we consider across a significant set of diverse benchmarks and configurations, enabling many types of analyses that would be otherwise impractical.

8 References

[1] S. Ahmed et al. The betrayal of constant power × time: Finding the missing joules of transiently-powered computers. In *Proceedings of LCTES*, 2019.

[2] A. R. Arreola et al. RESTOP: retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 2018.

[3] D. Balsamo et al. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 2015.

[4] D. Balsamo et al. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[5] 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*, 2018.

[6] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks*, 2016.

[7] N. A. Bhatti and L. Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of IPSN*, 2017.

[8] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of SenSys*, 2019.

[9] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *ACM SIGARCH Computer Architecture News*, 2016.

[10] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of OOPSLA*, 2016.

[11] A. Colin and B. Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of CC*, 2018.

[12] A. Colin, E. Ruppel, and B. Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of ASPLOS*, 2018.

[13] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 2011.

[14] M. Furlong, J. Hester, K. Storer, and J. Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of ENSsys*, 2016.

[15] J. Hester, T. Scott, and J. Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of SenSys*, 2014.

[16] J. Hester and J. Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of SenSys*, 2017.

[17] J. Hester and J. Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of SenSys*, 2017.

[18] J. Hester, K. Storer, and J. Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of SenSys*, 2017.

[19] M. Hicks. Mibench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2, 2016 (last access: Jan 5th, 2021).

[20] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. Quickrecall: A hw/sw approach for computing across power cycles in transiently powered computers. *ACM Journal on Emerging Technologies in Computing Systems*, 2015.

[21] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst.*, 2017.

[22] The LLVM compiler infrastructure. https://llvm.org/, 2003 (last access: Jan 5th, 2021).

[23] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of PLDI*, 2015.

[24] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM Programming Languages*, 2017.

[25] K. Maeng and B. Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of OSDI*, 2018.

[26] K. Maeng and B. Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of PLDI*, 2019.

[27] A. Maioli. Understanding and testing intermittence bugs in transiently-powered computers. Technical Report n 37/2019, Politecnico di Milano (Italy), 2019.

[28] A. Maioli. ScEpTIC documentation and source code. http://sceptic.neslab.it/, 2021 (last access: Jan 5th, 2021).

[29] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. On intermittence bugs in the battery-less internet of things (WiP paper). In *Proceedings of LCTES*, 2019.

[30] Texas Instruments. MSP430-FR5737 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5737.pdf, 2017 (last access: Jan 5th, 2021).

[31] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of MSPC*, 2014.

[32] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on rfid-scale devices. *ACM SIGARCH Computer Architecture News*, 2011.

[33] E. Ruppel and B. Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of PLDI*, 2019.

[34] E. Sardini and M. Serpelloni. Self-powered wireless sensor for air temperature and velocity measurements with energy harvesting capability. *IEEE Transactions on Instrumentation and Measurement*, 2011.

[35] E. Sazonov, H. Li, D. Curry, and P. Pillay. Self-powered sensors for monitoring of highway bridges. *IEEE Sensors Journal*, 2009.

[36] M. Spivak and S. Toledo. Storing a persistent transactional object heap on flash memory. In *Proceedings of LCTES*, 2006.

[37] M. Surbatovich, L. Jia, and B. Lucia. I/o dependent idempotence bugs in intermittent systems. *Proceedings of the ACM Programming Languages*, 2019.

[38] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI*, 2016.

[39] K. Vijayaraghavan and R. Rajamani. Novel batteryless wireless sensor for traffic-flow measurement. *IEEE Transactions on Vehicular Technology*, 2010.

[40] J. Yang et al. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of SenSys*, 2007.

Intermittence Anomalies not Considered Harmful

Andrea Maioli
Politecnico di Milano, Italy
andrea1.maioli@polimi.it

Luca Mottola
Politecnico di Milano, Italy and RISE, Sweden
luca.mottola@polimi.it

ABSTRACT

We consider a new perspective on intermittence anomalies arising in intermittently-computing mixed-volatile systems. Existing forward progress techniques avoid such anomalies by enforcing a computation that corresponds to a continuous one, introducing a significant overhead. We take a different stand: by allowing the presence of specific anomalies, we make the program aware of intermittence, unlocking new design patterns. We argue about the various possibilities emerging from this and we make the concept concrete by applying it to loops. We show how intermittence anomalies allow to preserve the results of loop iterations across power failures, without requiring to save the device's volatile state after each iteration. Compared to existing checkpoint mechanisms, our technique shows on average a 35.2x lower energy consumption and a 48.4x lower execution time across several staple benchmarks.

CCS CONCEPTS

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

KEYWORDS

Intermittence anomalies, intermittent computing.

ACM Reference Format:
Andrea Maioli and Luca Mottola. 2020. Intermittence Anomalies not Considered Harmful. In *The 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3417308.3430266

1 INTRODUCTION

Ambient energy harvesting for embedded sensing devices removes the maintenance costs and environment impact associated with battery replacement and disposal. Being harvested energy erratic and usually not sufficient to power a device continuously, these devices experience frequent power failures. Executions thus become *intermittent* [5], as periods of active computation are interrupted by periods where the device is powered off and recharges its energy buffer. Frequent power failures harm program forward progress, as power outages cause a device to shut down and loose the computational state, making it restart from scratch when power returns.

Managing persistent state. As we point out in Sec. 2, ensuring program forward progress across power failures requires saving a snapshot of the volatile state, namely a *checkpoint*, onto a non-volatile memory (NVM) location, which can be internal or external to the Micro Controller Unit (MCU). When power returns, restoring a checkpoint allows the MCU to resume the computation from where it stopped, as checkpoints contain a copy of main memory, program counter, and register file. Mixed-volatile systems [10, 11, 18] feature an internal NVM that they use as a portion of main memory. NVM is not included into checkpoints, as it already ensures persistency. This reduces checkpoint overhead, as the system saves only the volatile slice of main memory.

ENSsys '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-8129-1/20/11…$15.00
https://doi.org/10.1145/3417308.3430266

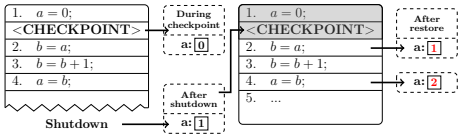


Figure 1: Example of intermittence anomaly. *A checkpoint saves the volatile state and then line* 4 *updates* a *to* 1. *Next, a power failure occurs. When energy returns, computation resumes from line* 2. *Being* a *non-volatile, it is not included in the checkpoint and retains the effects that line* 4 *produced during the previous power cycle. The execution produces a different result than a continuous execution.*

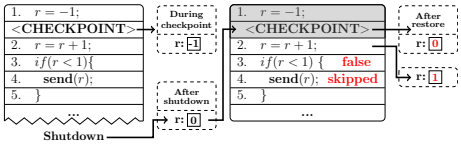


Figure 2: Example of intermittence-aware program. *Line* 2 *experiences the same intermittence anomaly as in Fig. 1. Variable* r *tracks the number of power failures.*

The use of mixed-volatile platforms may cause *intermittence anomalies* [11, 14], due to repeated executions of non-idempotent code. Fig. 1 shows an example. Being variable a non-volatile, it is not included in the checkpoint. The execution reaches line 4, which alters a, then a power failure happens. When the device resumes, it restores the volatile state from the checkpoint, and the execution resumes from line 2. Being non-volatile, a retained the effect that line 4 produced during the previous power cycle. This leads to a result that is unattainable in a continuous execution, as the re-execution of line 4 updates a to 2 instead of 1.

Avoiding intermittence anomalies requires to save additional checkpoints in specific program locations to break harmful sequences [14, 18]. For example, in Fig. 1 a checkpoint between line 2 and line 4 solves the issue. Generally, the more portions of the main memory are non-volatile, the more frequently checkpoints must be placed to avoid intermittence anomalies. This may nullify the performance gains due to reduced volatile state.

Intermittence awareness. Existing checkpoint mechanisms [8, 11, 13, 18] generally aim at enforcing an execution that corresponds to the continuous one. We take a different stand. We show how deliberately allowing the presence of specific intermittence anomalies in mixed-volatile MCUs may unlock new program design patterns. We call this concept *intermittence awareness*.

Intentionally allowing specific intermittence anomalies allows developers to consider intermittence as a program input. The resulting *intermittence-aware* program can change its behavior according to when and where a power failure happens. Fig. 2 shows an example. Variable r is non-volatile. Similar to Fig. 1, when the execution resumes after a power failure, r retains the effects that line 2 produced during the previous power cycle. By deliberately allowing the anomaly to occur, we can use r to track the number of power failures since the last checkpoint, as line 2 increments r every time the computation resumes. This ensures that line 4 is not re-executed when the program resumes, as the if statement of line 3 evaluates to *false*. Such behavior is not possible with existing approaches [3, 11, 16, 18], as they enforce results equivalent to a continuous computation, that is, r must equal 0 after line 2.

Intermittence awareness gives developers new degrees of freedom, as it unlocks new design patterns that would otherwise not be possible, applicable to either program *control flow* or *data flow*. Fig. 2 shows an example where intermittence awareness allows developers to affect the program *control flow* when resuming after a power failure. In constrast, by allowing the intermittence anomaly Fig. 1, we make the computation dependent from the number of power failures by altering its data flow.

Intermittence-aware loops. To demonstrate the use of intermittence awareness, we use it to reduce checkpoint overhead inside loops, as described in Sec. 3.

Power failures cause a device to loose the work done inside loops, unless a checkpoint is saved at the end of each iteration. This introduces a significant overhead, yet it is necessary in the absence of a priori knowledge on energy provisioning patterns. We identify a set of variables, called *loop state set*, that represent the minimum data to preserve. We instrument a loop by allocating its loop state set onto NVM, thus making it intermittence aware. This makes a checkpoint before the loop sufficient for resuming the computation from the latest loop iteration, ensuring forward progress with much lower overhead. Checkpoint frequency and size decrease, as checkpoints inside loops are no longer required and they do not include the loop state set.

Nonetheless, every time a device resumes from a power failure, it restores the latest checkpoint, introducing a startup overhead. Our technique also allows us to reduce this. We exploit intermittence awareness to skip the latest unfinished loop iteration, instead of re-executing it. The latter mitigates the startup overhead at the cost of a decreased precision, resulting in a behavior similar to the loop-perforation technique [17] used in approximate computing [15].

To enable the application of our loop instrumentation technique, we design and implement the LAPSUS[1] programming abstraction. LAPSUS exposes a small set of macros that allow developers to apply our loop instrumentation techniques without manually managing checkpoints, allocating variables, or designing dedicated data structures. Moreover, LAPSUS allows developers to decide and fine-tune where to apply our technique for mitigating the startup overhead when the program resumes after power failures.

In Sec. 4 we evaluate how LAPSUS affects the overhead of existing checkpoint mechanisms, based on staple intermittent computing benchmarks. Experimental results show that LAPSUS significantly lowers the overhead of existing approaches, reducing on average the number of executed instructions by $48.4x$, and obtaining a $35.2x$ lower energy consumption and a $48.4x$ lower execution time. In the worst case, that is, the CRC benchmark where checkpoint size is small, LAPSUS lowers the energy consumption overehead by $2.08x$. Instead, with higher checkpoint sizes, such as the implementation of Dijkstra algorithm, LAPSUS lowers the energy consumption overhead up to $227.79x$.

2 BACKGROUND AND RELATED WORK

We provide the necessary background and a brief discussion of related work here.

Ensuring forward progress. Various techniques [1–3, 10, 16, 18] adopt the concept of checkpoint to ensure program forward progress across power failures. Depending on how and where checkpoints execute, we classify these techniques as *dynamic* or *static*.

Dynamic checkpoint mechanisms, such as Hibernus [1, 2] and QuickRecall [10], rely on external interrupts that signal a low energy buffer for saving checkpoints. The program may thus be preempted at arbitrary places to take a checkpoint. Differently, *static* checkpoint mechanisms [3, 16, 18] place checkpoint calls in the program at compile time, fixing where checkpoints execute in the code. Among these systems, Ratchet [18] always saves a checkpoint when the execution encounters a checkpoint call. Mementos [16] and HarvOS [3] execute "trigger" calls to first verify the energy buffer for deciding whether to save a checkpoint.

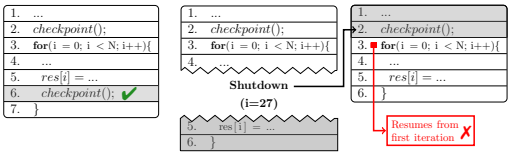
In contrast to checkpoint mechanisms that are applicable to unmodified source code, task-based programming abstractions [4, 6, 12, 19] require programmers to split the application logic in separate tasks executing with transactional semantics.

Intermittence anomalies. Checkpoint operations save the MCU volatile state into a NVM location. When power returns, restoring a checkpoint allows the MCU to resume the computation from where it stopped. However, the resulting runtime state may differ from the one of a continuous execution. In such a scenario, we define the runtime state as *anomalous*.

Resuming the computation with an anomalous runtime state may lead to *intermittence anomalies* [11, 14, 18], consisting in unexpected behaviors unattainable in a continuous execution. The effects of intermittence anomalies depend on how the program interacts with the anomalous part of the runtime state [14]. For example, in Fig. 1, the re-execution of lines 2-4 introduces a write-after-read (WAR) hazard [11, 14, 18]. Being a non-volatile, the re-execution of line 2 sees the effects that line 4 produced on a during the previous power cycle, as if line 2 re-executes just after line 4.

Avoiding intermittence anomalies. Two classes of techniques exist to verify the presence of intermittence anomalies [14] and to avoid their occurence [4, 8, 11, 13, 14, 18, 19]. One class of approaches breaks the sequence of operations involved in WAR hazards using a checkpoint [13, 14, 18] to avoid the operations accessing

[1] **L**ow-overhead intermittence-**A**ware **P**rogram in**S**trumentation techniq**U**e for loop**S**



(a) Checkpoint **(b) No checkpoint**

Figure 3: Preserving forward progress inside loops. *Fig. **(a)** shows a checkpoint placement that preserves progress across power failures. Fig. **(b)** shows the effects of removing the checkpoint. After a power failure occurs, the loop restarts all over again.*

the anomalous runtime state. For example, in Fig. 1, a checkpoint between lines 2 and 4 removes the WAR hazard and solves the anomaly. The second class of approaches creates multiple versions of the involved variables [8, 11], ensuring that read and write operations involved in the WAR hazard access different versions.

To our knowledge, no previous work considers the possibility of taking advantage from selected intermittence anomalies.

3 INTERMITTENCE-AWARE LOOPS

We show one possible application of the concept of intermittence awareness for mixed-volatile MCUs. We rely on specific intermittence anomalies to preserve the computation across loops, reducing checkpoint overhead. In doing so, we primarily target static checkpoint mechanisms. Dynamic checkpoint mechanisms may trigger checkpoints at any place in the code and only when it is strictly required to do so, essentially yielding no re-executions as the device likely immediately dies. This spares the overhead of trigger calls, at the cost of dedicated hardware support [1, 2, 10].

3.1 Example

Existing static checkpoint mechanisms [3, 16, 18] require to possibly save a checkpoint at the end of each loop iteration to preserve the work done inside loops. Such a conservative choice is necessary as, in general, erratic energy patterns may not provide guarantees on the complete executions of multiple loop iterations.

Consider the example of Fig. 3. A power failure inside the loop causes the device to resume from the latest checkpoint. The latter is saved at line 2, thus the computation resumes prior to the loop, re-executing it from scratch and wasting 28 iterations. Placing checkpoint calls at the end of each loop iteration introduces an overhead even if checkpoints do not actually take place, as certain operations occur anyways when executing the call, such as probing the energy buffers for their current energy content [3, 16].

Unlike existing techniques [3, 11, 16, 18], intermittence awareness allows specific intermittence anomalies to preserve the loop computational state across power failures without requiring a checkpoint at each iteration. Fig. 4 shows how to apply this concept to the example of Fig. 3. A checkpoint executes at line 2 and the loop starts the first iteration. Variables i and res are non-volatile. The execution reaches line 5, which stores the result of the first iteration into $res[0]$. Next, i increments to 1, and the second iteration completes. A power failure occurs during the third iteration. The computation eventually resumes from the checkpoint of line 2. Being i and res

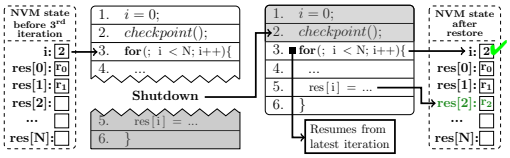
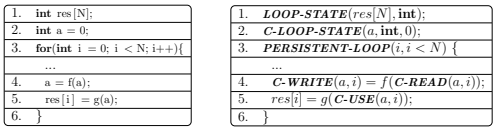


Figure 4: Example of an intermittence-aware loop. *A power failure happens during the third iteration. When the checkpoint is restored, the computation resumes from the beginning of the loop, but being* `i` *and* `res` *non-volatile, they retain the value right before the previous power cycles. Hence, the loop resumes from the third iteration, that is, the one interrupted by the power failure.*



(a) Original **(b) Instrumented**

Figure 5: Example of LAPSUS instrumentation macros. *Fig. **(a)** shows the program to instrument; Fig. **(b)** shows the instrumented program using LAPSUS macros.*

non-volatile, they retain state at the previous power cycle: i has a value of 2, and res stores the results of the previous loop iterations. Thus, the loop starts from the third iteration, as if a checkpoint is saved at the end of the second iteration.

These accesses represent an anomaly, as the value of i is produced during the previous power cycle. By allowing such an anomaly, we obtain the same results of a checkpoint placed at the end of each loop iteration, but without its overhead. Existing techniques for mixed-volatile systems [11, 18] do not allow this behavior. Despite they allow to directly allocate variables into NVM, as we do, they enforce executions equivalent to continuous ones. Thus, they would apply variable versioning [11] or place checkpoints [18] to ensure that line 3 and 5 do not access the anomalous value of variable i.

3.2 Instrumenting Loops

Loop state set. We first need to identify the variables representing the minimum set of data we must preserve across power failures, which we call *loop state set*. This includes, for example, the loop iterator and the variables carrying loop intermediate or final results. In the example of Fig. 3a, the loop state set includes variables i and res, which are the loop iterator and the results vector.

We allocate the loop state set into NVM. Variables not included in the loop state set remain at their original memory location. They are not necessary for resuming the computation without restarting the loop from the beginning, and thus we do not require to preserve them across power failures. A simple example is (volatile) variables local to the loop body, which are recomputed at every loop iteration.

Altering the loop. We remove any checkpoint inside the loop body and place a checkpoint before the loop statement. This ensures that the computation resumes at the beginning of the loop when power fails during the loop execution. Finally, we remove the initialization

of the loop iterator from the loop statement, and we place it before the only checkpoint remaining. This modification ensures that the loop resumes from the latest iteration and not from the first one, as the iterator is no longer re-initialized when resuming.

Fig. 4 shows an example. By allocating the loop state set onto NVM and by removing any checkpoint inside the loop, we are deliberately allowing intermittence anomalies. In a sense, we make the loop iterator i function as a checkpoint, as it saves onto NVM the index i of the last completed iteration. Once i increments, no power failure can lead to the re-execution of a loop iteration previous to i.

LAPSUS. To allow the application of intermittence awareness to loops, we design a programming abstraction called LAPSUS. We can use LAPSUS with a broad range of static checkpoint mechanisms, as our techniques do not rely on specific ones. LAPSUS provides a set of macros that allow developers to instrument a program by specifying the loop to be instrumented and its loop state set.

Fig. 5 shows an example. The original code is in Fig. 5a, whereas Fig. 5b shows the instrumented one. First, we substitute the loop construct with the macro **`PERSISTENT-LOOP`**, which takes two arguments: the loop iterator, and the loop condition. **`PERSISTENT-LOOP`** allocates the loop iterator into NVM and initializes it. Then, **`PERSISTENT-LOOP`** generates the *for* loop statement and places a checkpoint before it. Next, we specify the loop state set, which includes the variables *res*, *a*, and *i*. To that end, we substitute each variable declaration with the macro **`LOOP-STATE`**, except for the loop iterator *i*, which LAPSUS already identifies with **`PERSISTENT-LOOP`**. **`LOOP-STATE`** takes three arguments: the variable name, the variable type, and the initialization value, which is optional. **`LOOP-STATE`** allocates the variables into NVM and initializes them.

3.3 Avoiding Unwanted Anomalies

By allocating the loop state set onto NVM, we may introduce additional unwanted anomalies. Fig. 6a shows an example. Variable a is non-volatile, as it is included in the loop state set. Line 5 represents a WAR hazard [11, 14, 18] that leads to an intermittence anomaly. It first reads the value of a from NVM, executes function f, then writes the result back to NVM. As no checkpoint happens between read and write in line 5, a power failure during or after the function call causes an unwanted intermittence anomaly.

The technique we describe next remedies this issue for scalar variables, with additional overhead. It is not applicable for more complex data structures, such as arrays or linked lists. Addressing this limitation opens up interesting avenues for future work.

Versioning. To avoid placing a checkpoint inside the loop body, we apply a versioning technique. Fig. 6b shows how to avoid the intermittence anomaly of Fig. 6a. Variable a becomes a vector of two elements, each representing a version of a. At each iteration, a write operations target a copy and a read operations target the other. To carry the results across loop iterations, a read and write versions switch after every loop iteration.

This access pattern breaks the sequence of operations involved in the WAR hazard, as now line 5 read and write operations target different copies of a. As such, a power failure can no longer cause line 5 read operation to access an anomalous value, and we can avoid the intermittence anomaly without inserting a checkpoint.

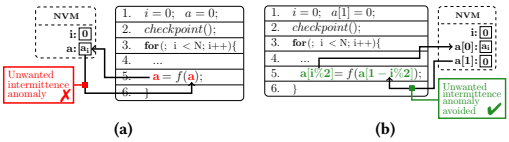


Figure 6: Avoiding unwanted intermittence anomalies in an intermittence-aware loop. *In Fig. **(a)**, line* 5 *read and write operations represents a WAR hazard [11, 14, 18] on variable* a. *Fig. **(b)** shows how to avoid the unwanted anomaly of variable* a.

LAPSUS support. LAPSUS includes macros to protect variables against unwanted intermittence anomalies. We use the example of Fig. 5a, where variable a exposes the same anomaly as Fig. 6a.

Being a part of the loop state set, we substitute its declaration at line 2 with the macro **`P-LOOP-STATE`**, where P stands for protected. **`P-LOOP-STATE`** takes the same arguments of **`LOOP-STATE`**, but it also creates the two copies of variable a that our technique requires. Next, we make read and write operations target the correct copy of a. To this end, LAPSUS provides three macros: **`P-WRITE`**, **`P-READ`**, and **`P-USE`**. They take two arguments: a variable and the loop iterator. **`P-READ`** and **`P-WRITE`** target the copy reserved for read and write operations, respectively. For example, at line 4 of Fig. 5a, we substitute the definition of variable a with **`P-WRITE(a, i)`**. Similarly, we substitute **`P-READ(a, i)`** in line 4. Instead, we use **`P-USE`** to access a value written by a previous operation in the same iteration, as in line 5 in Fig. 5a. Fig. 5b shows the final result. Note that we must address all protected variable accesses using the corresponding macro, even for accesses outside the loop.

3.4 Restore Approximation

Intermittence awareness not only reduces checkpoint overhead, but also makes resume operations more efficient, as the device restores a lower amount of data from the checkpoint.

Nonetheless, we may further tune our technique to mitigate the overhead when resuming. Fig. 7 shows an example. Here we intentionally place the increment of the loop iterator i at the beginning of the loop body. Let us suppose a power failure happens during the third loop iteration. When the computation resumes, i increments as first operation and the loop resumes from the fourth iteration, jumping the iteration interrupted by the power failure.

As a result, we obtain a behavior similar to loop perforation techniques [17] used in approximate computing [15], where iterations are skipped to trade accuracy for reduced energy consumption or execution time. Instead of considering a certain perforation rate to decide which iteration to skip, we skip iterations every time the device resumes after a power failure.

LAPSUS supports both the regular and the approximate approach when resuming. Programmers use macro **`APPROX-PERSISTENT-LOOP`** for selecting the approximation strategy, in place of **`PERSISTENT-LOOP`**. The two macros act similarly and take the same arguments. They declare the loop iterator as non-volatile, initialize it, and select the appropriate loop construct.

4 EVALUATION

We discuss our experimental setup and early results we gather to assess feasibility and potential impact of intermittence awareness.

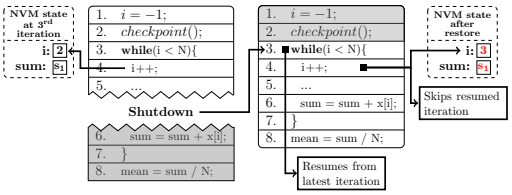


Figure 7: An intermittence-aware loop with restore approximation. *At the second iteration, line* 2 *increments the non-volatile loop iterator* i *to* 2*. Execution then resumes from the beginning of the loop after a power failure, but* i *retains the effects produced during the previous power cycle. Hence, the loop resumes from the third iteration, decrease result precision for mitigating the startup overhead.*

4.1 Setup

We consider the MSP430-FR5969 [9] MCU, an ultra-low power MCU often adopted in intermittent computing [2, 11, 12, 16, 18].

Baseline and benchmarks. We evaluate the performance of our technique by comparing it against a generic trigger-based static checkpoint mechanism that, akin to existing systems [3, 16], uses NVM only for storing checkpoints. At runtime, when a checkpoint call executes, it queries the ADC to decide whether to execute a checkpoint. We use the default compiler configuration when producing machine code [3, 16]. We call this approach TRIGGER.

We consider benchmarks commonly used in intermittent computing [1, 2, 8, 10, 16, 18], including Cyclic Redundancy Check (CRC) for data integrity, Fast Fourier Transform (FFT) for signal analysis, and the Dijkstra algorithm for finding the shortest path among nodes in a graph. We take these benchmarks from the open-source implementation of the MiBench2 [7] benchmark suite.

We do not quantitatively evaluate the approximate restore technique of Sec. 3.4. As with any approximation technique [15], the degree of acceptable approximation is inherently application-specific and thus an unbiased comparative evaluation is difficult.

Metrics. We focus on the main loops of each benchmark. We compare the increase in *i)* number of executed machine-code instructions, *ii)* energy consumption, and *iii)* execution time that LAPSUS and TRIGGER show compared to the non-instrumented program.

We calculate the increase in the number of machine-code instructions by identifying the loop body operations that differ from the non-instrumented program, that are, FRAM accesses, operations to protect against unwanted intermittence anomalies, trigger calls, and actual checkpoints. We then calculate the increase in energy consumption and execution time by considering the executed clock cycles, the energy consumption of each clock cycle, and the energy consumption and access latency for the ADC and FRAM usages [9].

We calculate the energy consumption per clock cycle of various operating modes as $e_x = \frac{V_{cc} * I_x}{f_{mcu}}$, where V_{cc} is the operating voltage ($3V$) and I_x is the current draw of the MCU under the operating mode x. We also consider an operating clock frequency f_{mcu} of either $8Mhz$ and $16Mhz$, as FRAM accesses require one wait state at $16Mhz$. Being not specified in the datasheet, we calculate the current draw I_{fram} of the MCU when it stores only the data segment into FRAM as $\frac{I_{fram\_uni} - I_{sram}}{2} + I_{sram}$, where I_{sram} and I_{fram_uni} are the current draws when the MCU operates respectively from SRAM and FRAM. Note that I_{fram_uni} refers to two FRAM accesses per clock cycle: one for instruction fetch and one for data access.

We consider trigger calls to happen at every loop iteration and the possible checkpoint to save the minimum amount of data. This places the baseline in the best possible conditions.

4.2 Results

Fig. 8 reports in logarithmic scale the overhead of single operations for LAPSUS and TRIGGER compared to the non-instrumented program across the benchmarks we consider. For TRIGGER, we report both the costs of trigger calls and of actual checkpoints, as trigger calls do not necessarily yield a checkpoint.

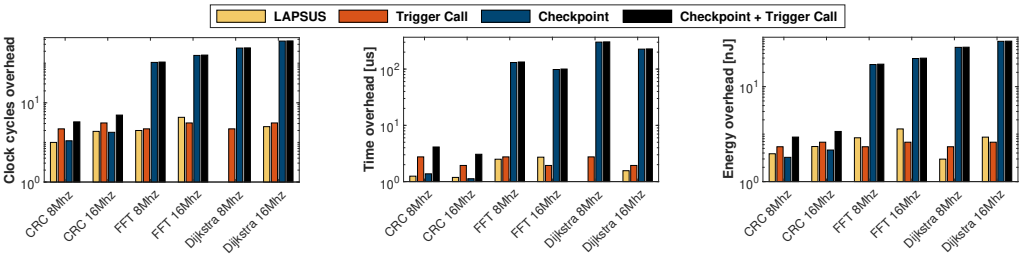
Overall, TRIGGER's complete checkpoint operations require on average a $48.4x$ more clock cycles and execution time than LAPSUS, as illustrated in Fig. 8a and Fig. 8b. LAPSUS consumes on average $35.2x$ less energy than TRIGGER complete checkpoint operations, as Fig. 8c shows. Compared to trigger calls alone, on average LAPSUS executes 37% fewer clock cycles and has a 37% lower execution time, but it has a 24% higher energy consumption. The latter are due to the cost of accessing FRAM.

The results specifically vary depending on the program structure. In the CRC benchmark, LAPSUS has a lower energy consumption than trigger calls alone, as Fig. 8c reports. LAPSUS instrumentation in CRC bears very low overhead, and querying the ADC results in higher energy consumption. Here we also notice that trigger calls represent most of the overhead of a complete checkpoint.

This is not the case for the FFT and Dijkstra. At $8Mhz$, LAPSUS introduces a lower number of clock cycles than trigger calls alone, as Fig. 8a shows. Compared to the same baseline, however, Fig. 8c reports a higher energy overhead for LAPSUS in the FFT benchmark, as now FRAM accesses are more costly than querying the ADC. This is is due to the additional operations to protect the execution against unwanted intermittence anomalies, as explained in Sec. 3.3. Increasing the clock to $16Mhz$ is further detrimental to LAPSUS performance, as now FRAM accesses require one wait state. Notwithstanding the higher energy consumption than trigger calls alone for FTT and Dijkstra, LAPSUS requires $99x$ less energy than complete checkpoint operations, as Fig. 8c shows.

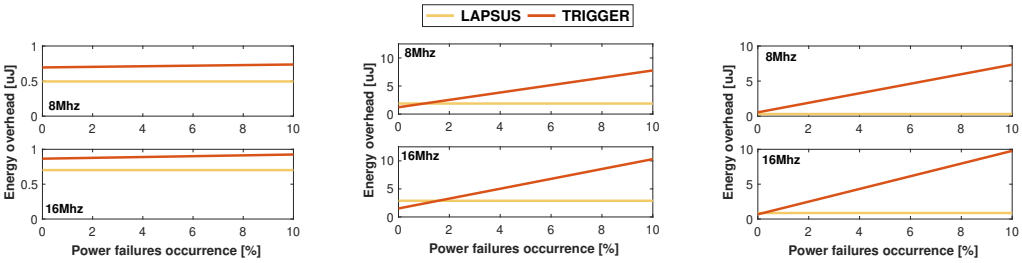
We also investigate how LAPSUS and TRIGGER energy overhead increase with the frequency of power failures. As TRIGGER executes a trigger call at the end of each loop iteration, the frequency of power failures also represents the number of trigger calls converting to a checkpoint. Note that Ransford et al. [16] reports 16 power failures during the execution of the CRC benchmark with $2Kb$ of data to checkpoint when using RF energy sources, corresponding to a 1.56% of trigger calls converting to a checkpoint. Being the FFT and Dijkstra benchmarks way more complex than CRC, we expect a higher frequency of checkpoint occurrences there.

Fig. 9 shows the results. LAPSUS performance across the board is constant, as LAPSUS does not save any checkpoint inside loops. In constrast, Fig. 9a shows that the energy overhead of TRIGGER starts above the one of LAPSUS already with infrequent power failures and slowly grows when power failures occur more often. Consistently with the earlier discussion, Fig. 9b demonstrates that LAPSUS energy overhead is larger than TRIGGER only with very



(a) Clock cycles overhead **(b) Execution time overhead** **(c) Energy overhead**

Figure 8: Overhead per loop iteration. *LAPSUS on average requires* 48.4x *less clock cycles than TRIGGER complete checkpoint operations, lowering the execution time by* 48.4x *and the energy consumption by* 35.2x*. Trigger calls alone in TRIGGER require on average a* 37% *higher number of clock cycles and execution time. However, LAPSUS FRAM accesses cause a* 24% *higher energy consumption than trigger calls.*



(a) CRC benchmark **(b) FFT benchmark** **(c) Dijkstra benchmark**

Figure 9: Energy overhead during complete executions, against a certain rate of power failures. *LAPSUS overhead is constant, whereas TRIGGER overhead increases with more frequent power failures, especially where checkpoints save a significant amount of data. LAPSUS overhead is lower than TRIGGER, except for cases with a scarce frequency of power failures.*

Benchmark	LAPSUS per loop iteration	TRIGGER per checkpoint
CRC	14	5
FFT	33,5	264
Dijkstra	25	605

Figure 10: Average NVM accesses. *Despite TRIGGER requires fewer NVM accesses than LAPSUS for the CRC benchmark at each checkpoint, the latter ultimately yields lower energy overhead, as shown in Fig. 10, because of the energy cost of trigger calls.*

rare power failures. As the latter happen more frequently, LAPSUS becomes most efficient. A similar observation applies to Fig. 9c.

While the energy overhead of LAPSUS is only due to NVM accesses to handle the loop state set, that of TRIGGER comes from a combination of NVM accesses for checkpointing and trigger calls. Fig. 10 reports statistics on NVM accesses for either solution. For LAPSUS, the NVM accesses are an average per loop iteration, whereas for TRIGGER they are required for each checkpoint. Interestingly, NVM accesses for TRIGGER with the CRC benchmark are lower than those for LAPSUS at every loop iteration. We conclude that the better performance of LAPSUS compared to TRIGGER in Fig. 9a is due to the overhead of trigger calls that do not yield a checkpoint. The opposite situation holds for the other benchmarks, even though the two figures are not directly comparable as LAPSUS incurs in the given number of NVM accesses at every iteration, whereas TRIGGER pays the overhead only when checkpointing.

5 CONCLUSION

Intermittence awareness allows the occurrence of specific anomalies to gain new information regarding intermittence, unlocking new design patterns. Developers exploit intermittence awareness to make their program react to intermittence, altering the program control flow and/or data flow accordingly. We make this concept concrete with an instrumentation technique that uses intermittence awareness to reduce checkpoints overhead inside loops. Our technique preserves the loop computational state across power failures, without requiring to save a checkpoint after each iteration. The LAPSUS programming abstraction facilitates developers in applying our technique to loops. We compare LAPSUS against existing trigger-based checkpoint mechanisms. Across the benchmarks we test, on average LAPSUS lowers the energy overhead of existing checkpoint mechanism by 35.2x and reduces the execution time by 48.4x, demonstrating the impact of intermittence awareness.

Our technique has limitations, such as handling non-idempotent accesses to complex data structures. It also introduces some non-determinism that may complicate testing, as program execution becomes dependent on energy patterns. As we plant the seed for intermittence awareness, we also seek to address these issues.

REFERENCES

[1] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[2] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[3] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[4] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[5] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[6] J. Hester, K. Storer, and J. Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[7] M. Hicks. 2016 (last access: September 18th, 2020). MiBench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[8] M. Hicks. 2017. Clank: Architectural Support for Intermittent Computation. (2017).

[9] Texas Instruments. 2017 (last access: September 18th, 2020). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[10] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[11] B. Lucia and B. Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[12] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[13] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[14] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[15] S. Mittal. 2016. A Survey of Techniques for Approximate Computing. *Comput. Surveys* (2016).

[16] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[17] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. 2011. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*.

[18] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[19] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.

ALFRED: Virtual Memory for Intermittent Computing

Andrea Maioli
Politecnico di Milano, Italy

Luca Mottola
Politecnico di Milano, Italy and RI.SE Computer Science and Uppsala University, Sweden

ABSTRACT

We present ALFRED: a virtual memory abstraction that resolves the dichotomy between volatile and non-volatile memory in intermittent computing. Mixed-volatile microcontrollers allow programmers to allocate part of the application state onto non-volatile memory. Programmers are therefore to manually explore the trade-off between simpler management of persistent state against energy overhead and possibility of intermittence anomalies due to non-volatile memory operations. This approach is laborious and yields sub-optimal performance. We take a different stand with ALFRED: we provide programmers with a virtual memory abstraction detached from the specific volatile nature of memory and automatically determine an efficient mapping from virtual to volatile or non-volatile memory. Unlike existing works, ALFRED does not require programmers to learn a new language syntax and the mapping is entirely resolved at compile-time, reducing the run-time energy overhead. We implement ALFRED through a series of machine-level code transformations. Compared to existing systems, we demonstrate that ALFRED reduces energy consumption by up to *two orders of magnitude* given a fixed workload. This enables workloads to finish sooner, as the use of available energy shifts from ensuring forward progress to useful application processing.

CCS CONCEPTS

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

KEYWORDS

Intermittent computing, virtual memory abstraction.

ACM Reference Format:
Andrea Maioli and Luca Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *The 19th ACM Conference on Embedded Networked Sensor Systems (SenSys'21), November 15–17, 2021, Coimbra, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485730.3485949

1 INTRODUCTION

Ambient energy harvesting [10] enables deployments of battery-less sensing devices [1, 20, 23, 27, 48, 50], reducing environment impact and maintenance costs. However, harvested energy is erratic and may not suffice to power devices continuously. Frequent power failures occur that cause executions to become *intermittent*, with periods of active operation interleaved by periods where a device is off recharging energy buffers. Power failures cause devices to lose the program state, restarting all over again when energy is newly available. Forward progress of programs is therefore compromised.

Problem. Several systems exist to ensure forward progress, as we discuss in Sec. 2. Common to these solutions is the insertion of state-saving operations within the execution flow. These operations offer the opportunity to create a replica of the program state, including main memory, register files, and program counter, onto non-volatile memory. The program state is eventually restored from non-volatile memory when energy returns, ensuring forward progress across power failures. The placement of state-saving operations in the program may be either decided in a (semi-)automatic fashion [7, 8, 11, 29, 36, 46, 49] or driven by programmers with custom programming abstractions [16, 34, 35, 44, 47, 52].

Mixed-volatile microcontrollers also exist, which offer the ability to store slices of the program state directly onto non-volatile memory. This is achieved using specific pragma statements [28], as

```
#pragma PERSISTENT(x)
unsigned int x = 5;
```

Program state allocated on non-volatile memory is automatically retained across power failures and may be excluded from state-saving operations, simplifying the management of persistent state.

Using mixed-volatile microcontrollers comes at the cost of increased energy consumption: non-volatile memory operations may require up to 247% the energy of their volatile counterpart [28, 40]. Storing only parts of the program state on non-volatile memory may also yield intermittence anomalies [43, 45], due to re-executions of non-idempotent code, which require further energy to be corrected. Using mixed-volatile platforms, quantifying the advantages in simpler management of persistent state against the corresponding energy overhead is complex, as these depend on multiple factors including energy patterns and execution flow.

ALFRED. We take a different stand. Rather than requiring programmers to manually determine when to use non-volatile memory for what slice of the program state, we promote a higher-level of abstraction through a concept of *virtual memory*. Programmers write intermittent code without explicitly mapping variables to volatile or non-volatile memory. Given a placement of state-saving operations in the code, we *automatically* decide *what* slice of the program state must be allocated onto non-volatile memory, and *at what point* in the execution. Programmers are therefore relieved from deciding the mapping between program state and memory. Moreover, the mapping is not fixed at variable level, but is *automatically adjusted* at different places in the code for the same data item, based on read/write patterns and program structure.

SenSys'21, November 15–17, 2021, Coimbra, Portugal

ACM ISBN 978-1-4503-9097-2/21/11…$15.00
https://doi.org/10.1145/3485730.3485949

ALFRED[1] is our implementation of virtual memory for intermittent computing, based on two key features:

(1) it is *transparent to programmers*: no dedicated syntax is to be learned, and programmers write code in the familiar sequential manner without the need to tag variables.
(2) the mapping from virtual to volatile or non-volatile memory is entirely *resolved at compile-time*, reducing the energy overhead that represents the cost of using the abstraction.

The virtual memory abstraction we conceive does not provide virtualization in the same sense as mainstream OSes. Instead, it offers an abstraction that shields programmers from the need to statically determine a specific memory allocation schema. ALFRED is therefore sharply different compared to mainstream virtual memory systems [5, 19]. These usually provide an idealized abstraction of storage resources, so that software processes operate as if they had access to a contiguous memory area, possibly even larger than the one physically available. Address translation hardware maps virtual addresses to physical addresses at run-time. In ALFRED, we target resource-constrained energy-harvesting devices that compute intermittently [23]. The abstraction we offer provides programmers with a higher-level view on the persistency properties of different memory areas, and automatically determines the mapping from the virtual memory to the volatile or non-volatile one. Because of resource constraints, we determine this mapping at compile-time.

ALFRED determines this mapping using three key program transformation techniques, illustrated in Sec. 3. Their ultimate goal is simple, yet challenging to achieve, especially at compile-time:

Use the energy-efficient volatile memory as much as possible, while enabling forward progress using non-volatile memory with reduced energy consumption compared to existing solutions.

This entails that we need to promote the use of volatile memory whenever convenient, for example, to compute intermediate results or to store temporary data that need not survive a power failure, while allocating the data that does require to be persistent onto non-volatile memory in anticipation of a possible power failure. By doing so, we decrease energy consumption by taking the best of both worlds: we benefit from the lower energy consumption of volatile memory *whenever possible*, and rely on the persistency features of non-volatile memory *whenever required*.

Applying program transformations at compile-time is, however, challenging because of the lack of run-time information. Sec. 4 illustrates how we address the uncertainty that arises, using a set of dedicated program normalization passes. The result of the transformations require a specific memory layout to operate correctly and a solution to the possible intermittence anomalies. We describe in Sec. 5 how we deal with these issues, using an approach that is co-designed with our program transformation techniques.

We build an implementation of ALFRED based on ScEpTIC [38, 43], an extensible open-source emulation environment for intermittent programs. Given fixed workloads and staple benchmarks in the field [7, 8, 26, 29, 43, 46, 49], we measure ALFRED performance in energy consumption, number of clock cycles, memory accesses, and restore operations. We compare ALFRED with multiple baselines obtained by abstracting the key design dimensions of existing systems in a framework that allows us to instantiate baselines matching existing systems, while also exploring alternative configurations. Depending on the benchmark, ALFRED can provide *several-fold improvements* in energy consumption, which allow the system to shift the energy budget to useful application processing. This correspondingly allows the system to achieve comparable improvements in the time to complete the workloads.

[1] **A**utomatic a**L**location o**F** non-volatile memo**R**y for transi**E**ntly-powered **D**evices.

2 RELATED WORK

Ensuring forward progress is arguably the focus of most existing works in intermittent computing [23]. Common to these is the use of some form of persistent state on non-volatile memory.

A significant fraction of existing solutions employ a form checkpointing to cross power failures [3, 7, 11, 36, 46]. This consists in replicating the content of main memory, special registers, and program counter onto non-volatile memory at specific points in the code. Whenever the device resumes with new energy, state is retrieved back from non-volatile memory and computations restart. Systems such as Hibernus [7, 8] operate in a reactive manner: an interrupt is fired from a hardware device that prompts the application to take a checkpoint, for example, whenever the energy buffer falls below a threshold. Differently, systems exist that place explicit function calls to perform checkpoints [11, 36, 46, 49]. The specific placement is a function of program structure and energy patterns.

Other approaches offer abstractions that programmers use to define and manage persistent state [16, 34, 35, 52] and time profiles [24]. For example, DINO [34] allows programmers to split the sequential execution in individual tasks and ensures transactional semantics between consecutive task boundaries. Alpaca [35] goes a step further and provides dedicated abstractions to defines tasks as individual execution units that run with transactional semantics against power failures and subsequent reboots.

Using mixed-volatile platforms, intermittence anomalies potentially occur due to repeated executions of non-idempotent code [43, 45]. These are unexpected program behaviors that make executions differ from their continuous counterparts. Systems are available that address these issues with dedicated checkpoint placement strategies [49] or custom programming abstractions [16, 34, 35, 52], and to test their occurrence [38, 43]. Approaches are available that conversely take advantage of them to realize intermittence-aware control flows, promoting the occurrence of power failures to an additional program input [40]. Additional issues in intermittent computing include performing general testing of intermittent programs [15, 17, 21, 22], profiling their energy consumption [2, 15, 21], and handling peripheral states across power failures [6, 9, 12, 37].

Our work offers a different standpoint. Unlike the works above, we take the decision about what part of the application state to allocate on non-volatile memory away from programmers, and offer a uniform abstraction that does not entail any specific memory configuration. A set of program transformation techniques automatically determines an energy-efficient allocation at compile time, as a function of program structure and read/write patterns. Most importantly, such an allocation is not fixed once and for all at variable-level as in current practice, but is possibly adjusted at different places in the code for the same data item.

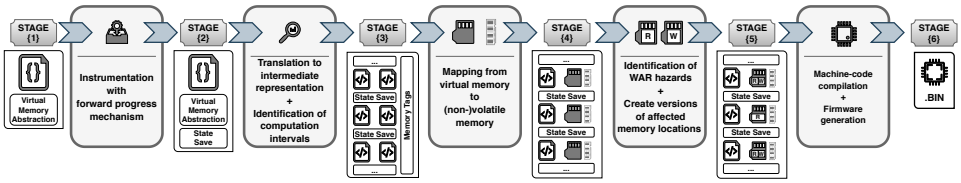


Figure 1: ALFRED compile-time pipeline.

Closest to our work are TICS [31] and the system of Jayakumar et al. [30]. TICS [31] limits the size of persistent state by solely saving the active stack frame and modified memory locations outside of it, which is conceptually similar to our approach. However, TICS primarily helps programmers deal with time across power failures, whereas we specifically target energy efficiency. TICS also exclusively uses non-volatile memory for global data and undo-logging [36] to avoid intermittence anomalies [43, 45]. In contrast, we opportunistically allocate slices of program state onto the energy-efficient volatile memory and employ program transformation techniques that ensure memory idempotency [49].

The system of Jayakumar et al. [30] adjusts the mapping of global variables, program code, and stack frames between volatile and non-volatile memory, doing so at the granularity of individual functions. They rely on hardware interrupts to trigger state-saving operations at runtime and tentatively allocate everything to non-volatile memory first, then incrementally move data or code to volatile memory until forward progress is compromised. At that point, they backtrack to the latest functioning configuration. Besides working at the granularity of single data items and at compile-time, rather than at run-time, our design is fundamentally different, as memory allocations are thought to *systematically* improve energy consumption. Therefore, if forward progress is possible before applying ALFRED, it remains so afterwards. ALFRED is thus never detrimental to the application's ability to do useful work.

3 VIRTUAL MEMORY MAPPING

The program transformation techniques of ALFRED determine the mapping from virtual to volatile or non-volatile memory. They are independent of the target architecture, as they are applied on an architecture-independent intermediate representation of the input program commonly used in compilers [33]. We illustrate the compile-time pipeline in Sec. 3.1, followed by an explanation of the single techniques in Sec. 3.2 to Sec. 3.4.

3.1 Overview

Fig. 1 shows the compile-time pipeline of ALFRED. The input at stage ⟨1⟩ is a program written using the virtual memory abstraction; therefore, variables in the program are not explicitly mapped to either volatile or non-volatile memory.

The program is first processed through the compile-time support an existing checkpoint system [7, 8, 11, 29, 36, 46, 49] or task-based programming abstraction [16, 34, 35, 44, 47, 52]. Either way, at stage ⟨2⟩ the program includes state-save operations inlined in the execution flow as calls to a checkpointing subsystem or placed at task boundaries. These operations are meant to dump program state onto non-volatile memory prior to a power failure and to restore the program state from non-volatile memory when energy is newly available. The techniques we explain next are orthogonal to how state-save operations are placed in the code.

Unlike existing programming systems for intermittent computing, our techniques work at the level of machine-code. At this level, memory operations are visible as they are actually executed on the target platform. At stage ⟨3⟩ in Fig. 1 we translate the program into an intermediate representation of the source code and initially map every memory operation *to volatile memory*. If we were to execute the code this way, state-save operations would need to dump the entire main memory to the non-volatile one when executing.

At the same stage we also partition the code into logical units we call *computation intervals*. A computation interval consists in a sequence of machine-code instructions executed between two state-save operations. For programs using checkpoint mechanisms [7, 8, 11, 29, 36, 46, 49], computation intervals correspond to sequences of instructions between two checkpoint calls. For programs using task-based programming abstractions [16, 34, 35, 44, 47, 52], computation intervals essentially correspond to tasks.

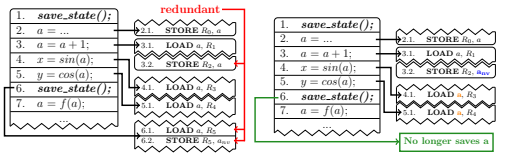
From now on, the three program transformations we illustrate next are applied in the order we present them. We focus on the intuition and general application of each transformation and postpone the discussion about dealing with compile-time uncertainty to Sec. 4. Our techniques operate on every memory target in the program, including not just memory targets that the compiler uses to map variables in source code, but also memory locations used by operations that are normally transparent to programmers, such as `PUSH` or `POP`. We detail how we identify the memory addresses of data items possibly involved in a transformation in Sec. 4 and how to compute their addresses after the transformations in Sec. 5.

As we hinted earlier, the mapping we want to achieve is one where volatile memory is used as much as possible for data that requires no persistency, for example, intermediate results or temporary data, as it is more energy efficient than its non-volatile counterpart. However, we want to make sure to use the latter, paying an energy overhead, whenever persisting data to survive power failures is necessary. Intuitively, the transformations generate a mapping from virtual to volatile or non-volatile memory where the former acts as a volatile cache of sorts.

The snippets we show next include both source and machine code for clarity. Line numbers refer to source code.

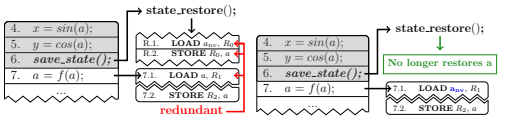
3.2 Mapping Write Operations

The first transformation we apply is based on a key intuition: *a memory write operation should target non-volatile memory as soon*



(a) Before the transformation. (b) After the transformation.

Figure 2: Example of mapping write operations.



(a) Before the transformation. (b) After the transformation.

Figure 3: Example of mapping read operations.

as the written data is final compared to the next state-save operation, so it relieves the latter from the corresponding processing.

The notion of *final* describes situations where the program no longer alters the data before the next state-save operation. Our intuition essentially corresponds to *anticipating* the actions that the state-save operation would perform anyways. This allows these operations to spare the overhead for saving data that can be considered final earlier: after the transformation the data is already on non-volatile memory when the state-save operation executes.

Example. Consider the program of Fig. 2(a) and let us focus on the computation interval extending up to line 6. We find two **STORE** instructions that target the volatile memory location that variable a is initially mapped to. Note that the second **STORE** instruction writes the same value that the state-save operation of line 6 stores for variable a, because the latter is initially allocated onto volatile memory and must be preserved across power failures. This is the case because the data for variable a is *final* already at line 3.

To save the overhead of redundant memory operations, we make the **STORE** instruction of line 3 immediately target non-volatile memory. This transformation allows us to remove the instructions that are necessary to save variable a at the state-save operation of line 6, along with the corresponding energy overhead, as line 3 already saves the content variable a onto non-volatile memory.

Fig. 2(b) shows the resulting program, which has reduced energy overhead because the state-save operation is no longer concerned with variable a that is made persistent already at line 3. Conceptually, this corresponds to moving the **STORE** instruction that would normally be part of the state-save operation to the last point in the program where variable a is actually written.

This transformation does not alter the target of the **STORE** instruction of line 2, where the data is not final yet. Doing so would incur an unnecessary energy overhead due to a write operation on non-volatile memory for non-final data, which is going to be overwritten soon after. In fact, the **STORE** instruction of line 2 produces an intermediate result for variable a, which we need not persist.

Generalization. We apply this technique to an arbitrary computation interval as follows. For each memory location x, we consider the possibly empty set of memory write instructions $I_w = (I_{w1}, ..., I_{wn})$ that manipulate x and are included in the computation interval; I_{wn} is the last such instruction and there is no other memory write instruction before the next state-save operation.

We relocate the target of I_{wn} to non-volatile memory, as whatever data I_{wn} stores is final. The targets of all other write instructions $I_{w1}, ..., I_{w(n-1)}$ remains on volatile memory, as they produce intermediate results that I_{wn} eventually overwrites. Note that this transformation is sufficient to preserve the value of the memory location x across power failures, while reducing the number of instructions targeting non-volatile memory.

By applying this transformation to all computation intervals and all memory locations, state-saving operations at stage ⟨4⟩ in Fig. 1 are left with *only* register file and special registers to handle, and accordingly modified. If a memory location is altered in a computation interval, our technique identifies when such a change is final and persists the data there. Otherwise, if $I = \emptyset$ there is no need to persist the data, as some previous state-save operation already did that the last time the data changed.

This processing not only reduces the operations on non-volatile memory, but also reduces the overhead of state-saving operations. A regular checkpoint mechanism would save the entire content of volatile memory onto the non-volatile one [7, 8, 11, 46], including unmodified memory locations. In our case, memory locations not modified in a computation interval are excluded from processing. We thus achieve differential checkpointing [3] with *zero* run-time overhead in both energy and memory consumption.

Next, consider the read instructions possibly included in the computation interval between I_{wn} and the state-save operation. As the data is now on non-volatile memory, in principle, they should also be redirected to non-volatile memory. Whether this is the most efficient choice, however, is not as simple. The third transformation, described in Sec. 3.4, addresses the related trade-offs.

3.3 Mapping Read Operations

The second transformation is based on the dual intuition: *when resuming, restore routines may be limited to register file and special registers, while memory read operations from non-volatile memory should be postponed to whenever the data is needed, if at all.*

This transformation effectively corresponds to *postponing* the restore operation to when the data is actually used and a read operation would execute anyways. By doing so, we spare the instructions in the restore routines that would load the data back to volatile memory from the non-volatile one. This is the case after applying the first transformation, which makes state-save operations be limited to restoring the register file and special registers. The content of main memory is persisted earlier, when it becomes final.

Example. Consider the program of Fig. 3(a). Following a power failure, the execution resumes from line 6 as the restore routines loads the value of the program counter from non-volatile memory, along with register file, other special registers, and the slice of main memory that was persisted prior to the power failure. However, note that the **LOAD** instruction of line 7 reads the same value for variable a that is loaded earlier as part of the restore routine.

A more efficient strategy is to limit the restore routine to register file and special registers, and make the **LOAD** instruction of line 7 target the non-volatile memory where the data resides. Compared

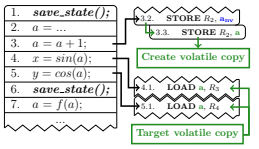


Figure 4: Consolidating read operations.

to a regular checkpoint mechanism, this transformation allows us to remove the instructions that restore variable a from checkpoint data, as the first read instruction that is actually part of the program is relocated to the right address on non-volatile memory.

Fig. 3(b) shows the program after this transformation, which bears reduced energy overhead because the restore routine is no longer concerned with variable a, as it is loaded straight from non-volatile memory if and when necessary. This corresponds to moving the **LOAD** instruction normally be part of the restore to routine for variable a to where in the program variable a is actually read.

Generalization. Similar to the previous transformation, we apply this technique to an arbitrary computation interval as follows. First, we limit restore routines to register file and special registers. Next, for each memory location x, we consider the possibly empty set of memory read instructions $I_r = (I_{r1}, ..., I_{rn})$ that manipulate x and are included in the computation interval. Dually to the first transformation, I_{r1} is the first such instruction and there is no other memory read instruction after the state-save operation at the start of the computation interval. We relocate the target of I_{r1} to non-volatile memory, as that is where the data is to be loaded from.

Whether the remaining $n - 1$ read operations $I_{r2}, ..., I_{rn}$ in a computation interval are to target volatile or non-volatile memory is determined by applying the program transformation that follows.

3.4 Consolidating Read Operations

Starting with a program that exclusively uses volatile memory at stage ⟨3⟩ in Fig. 1, the first two transformations relocate the target of selected read or write operations to non-volatile memory. As data now resides on non-volatile memory near state-save operations, further relocations to non-volatile memory may be required for other read operations. This is the case, for example, for read operations following the last non-volatile write operation that makes data final, as mentioned in Sec. 3.2. Whether this is the most efficient choice, however, is not straightforward to determine.

The third transformation is based on the intuition that *whenever memory operations are relocated to non-volatile memory, it may be convenient to create a volatile copy of data to benefit from lower energy consumption for read operations.*

Example. The program in Fig. 2(b) includes further read operations after line 3 and memory location a is on non-volatile memory as a result of the first transformation. In principle, we should relocate the read instructions on line 4 and 5 to non-volatile memory, as that is where the sought data resides. Because of the higher energy consumption of non-volatile memory, doing so may possibly backfire, outweighing the gains of the first transformation.

We must thus determine whether it is worth paying the penalty for creating a volatile copy of variable a to benefit from the more energy efficient operations there. Such a penalty is represented by an additional **STORE** instruction to create a copy of the data on volatile memory, as shown in Fig. 4. The new **STORE** uses the same source register, hence it represents the only added overhead. The benefit is the improved energy consumption obtained by making the instructions of line 4 and 5 target volatile memory, instead of the non-volatile one. Note that the exact same situation occurs for read instructions following the first **LOAD** instruction in Fig. 3(b).

Consider the frequently used MSP430-FR5969 [16, 28, 34, 35, 40] with internal FRAM as non-volatile memory, and say it runs at $16MHz$, where FRAM accesses require an extra clock cycle. Based on the datasheet [28], we calculate that if read operations in line 4 and 5 target non-volatile memory, the program consumes $1.522nJ$ for these operations. I we pay the penalty of the additional **STORE** instruction, but use volatile memory for all other read operations, the program consumes $1.376nJ$ for the same processing. This is a 10.6% improvement. We accordingly insert an additional **STORE** instruction after line 3 to copy a to volatile memory and we keep the read operations of line 4 and 5 target volatile memory.

Generalization. For each memory location x, we consider the n read instructions $I_{r1}, ..., I_{rn}$ in a computation interval that we need to consolidate, thus excluding those altered by the second transformation. We compute the energy consumption of a single non-volatile memory read instruction as

$$E_{read\_nv} = E_{nv\_read\_cc} * (1 + CC_{read}), \tag{1}$$

where $E_{nv\_read\_cc}$ is the energy consumption per clock cycle of the non-volatile memory read instruction and CC_{read} are the extra clock cycles possibly required, as mixed-volatile microcontrollers may incur in extra clock cycles when operating on the slower non-volatile memory. These clock cycles consume the same energy as a regular non-volatile read operation.

The break-even point between paying the penalty of an additional **STORE** instruction to benefit from more energy-efficient volatile read operations, versus the cost of allocating all read operations to non-volatile memory is determined according to inequality

$$E_{read\_nv} * n < E_{write} + E_{read} * n, \tag{2}$$

where E_{read_nv} is the one of Eq. 1, n is the number of considered memory read instructions, and E_{read} and E_{write} represent the energy consumption of a volatile memory read and write instruction, respectively. This can be rewritten as

$$0 < E_{write} - n * (E_{nv\_read\_cc} * (1 + CC_{read}) - E_{read}). \tag{3}$$

As the energy figures are fixed for a given microcontroller, Eq. 3 is exclusively a function of n, that is, the number of memory read instructions to consolidate in the computation interval. We can accordingly state that creating a volatile copy of the considered memory location is beneficial as long as

$$n > n_{min}, \text{ with } n_{min} = \left\lfloor \frac{E_{write}}{E_{nv\_read\_cc} * (1 + CC_{read}) - E_{read}} \right\rfloor, \tag{4}$$

where n_{min} is the minimum number of memory read instructions to ensure that creating a volatile copy of a memory location incurs in lower overall energy consumption. If the condition of Eq. 4 is not met, we make the n read operations target non-volatile memory.

Interestingly, n_{min} is independent of the specific read/write memory patterns and of program structure. It only depends on hardware

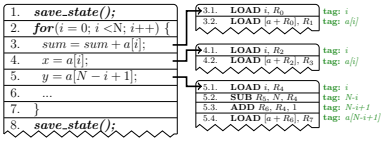


Figure 5: Example of the same group of instructions accessing multiple memory locations.

features. As an example, n_{min} is 0 (2) for the MSP430-FR5969 at a clock frequency of 16MHz (8Mhz). This means that if the microcontroller runs at 16MHz, it is *always* beneficial to create a volatile copy of the relevant memory locations.

4 COMPILE-TIME UNCERTAINTY

The transformation techniques of Sec. 3 rely on program information, such as the order of instruction execution and accessed memory addresses, that may not not be completely available at compile time. Constructs altering the control flow, such as conditional statements or loops, and memory accesses through pointers make these information a function of the run-time state. We describe next how we resolve this uncertainty, making it possible to apply the techniques of Sec. 3 to arbitrary programs.

We distinguish between two types of compile-time uncertainty. *Memory uncertainty* occurs when the exact memory address that a read/write operation targets cannot be determined. We resolve this uncertainty using virtual memory tags, as described in Sec. 4.1. *Instruction uncertainty* occurs when the order of instruction execution is not certain. Addressing this issue requires different techniques depending on program structure. In the interest of brevity, we give an intuition of how we can achieve this in the case of loops in Sec. 4.2. The corresponding generalization is available nonetheless [41].

Here again, the code snippets include both source and machine code for easier illustration, with line numbers pointing to the former, yet ALFRED operates entirely on machine code.

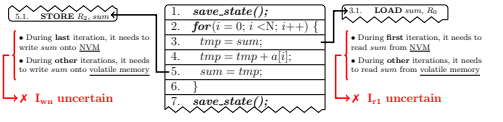
4.1 Memory Uncertainty

Our key observation here is that the techniques of Sec. 3 do not necessarily require exact memory addresses to operate; rather, they must identify the groups of instructions accessing the same memory location, whatever that may be.

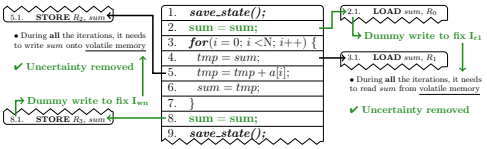
Example. Fig. 5 shows an example. Lines 3, 4, and 5 target multiple memory locations across different iterations of the loop. The corresponding physical addresses in memory change at every iteration.

To apply the techniques of Sec. 3, however, exact knowledge of the physical addresses in memory is not required. We rather need to determine that, at any given iteration of the loop, lines 3 and 4 target the same memory location, whereas line 5 targets a different one. Note that the information available in machine code is insufficient to this end: from that, we can only conclude that lines 3, 4, and 5 access all the addresses in the range $(a[0], a[N-1])$.

We automatically associate a *virtual memory tag* to every memory locations an instruction targets, as shown in Fig. 5. A virtual memory tag is an abstraction of physical memory that aids the application of the techniques of Sec. 3 by succinctly capturing what memory locations are *the same* in a computation interval.



(a) Example of a compile-time uncertainty in a loop.



(b) Normalized form of the loop that removes the compile-time uncertainty.

Figure 6: Example of compile-time uncertainty with loops.

In the program of Fig. 5, we attach the tag $a[i]$ to the memory locations read in lines 3 and 4. Instead, we attach the tag $a[N-i+1]$ to the memory location read in line 5. This information is sufficient for the technique mapping read operations, described in Sec. 3.3, to understand that line 3 and 4 are to be considered as one sequence I'_r, whereas line 5 is to be considered as a different sequence I''_r.

Virtual memory tags are, in a way, similar to debug symbols attached to machine code. They are obtained by inspecting the source code ahead of the corresponding translation, through multiple passes of a dedicated pre-processor. The transformations of Sec. 3 look at these information, instead of the memory locations in machine code. To handle pointers, we combine virtual memory tags with memory alias analysis [14, 32] to identify cases of indirect access to the same memory location. Unlike debug symbols, however, these information is removed from the program at stage ⟨5⟩.

4.2 Instruction Uncertainty → Loops

Key to the application of the program transformations in Sec. 3.2 and Sec. 3.3 is the identification of the last (first) memory write (read) instruction in a computation interval. This may be affected by loops, conditional statements, and function calls that alter the order of instruction execution. Further, whenever the execution of state-save operations depends on run-time information, for example, when a checkpoint call lies in a loop, the span of computation intervals is also undefined at compile time.

We describe next how we address these issues in the case of loops; how we deal with all other cases is available elsewhere [41].

Example. Fig. 6(a) exemplifies the situation. Say we are to apply the mapping of write operations, described in Sec. 3.2. Doing so requires to identify the last memory write instruction I_{wn} before the state-save operation. Depending on the value of i compared to N, the write operation in line 5 may or may not be the one that makes the data final for variable *sum*. The same reasoning is valid when we are to apply the mapping of read operations, described in Sec. 3.3. Depending on the value of i compared to N, the read operation in line 3 may or may not be the first for variable *sum* after the state restore. As a matter of fact, i and N are in control of what write (read) instruction is the I_{wn} (I_{r1}).

One may operate pessimistically and make both the **LOAD** on line 3 and the **STORE** on line 5 target non-volatile memory. This choice may be inefficient, because for all values of i that are neither 0 nor $N-1$, the loop computes intermediate results that are going to be overwritten anyways, so the cost of non-volatile memory operations is unnecessary. To complicate matters, the value of N itself may vary across different executions of the same fragment of code, as it may depend on runtime state.

Normalization. We apply techniques of *program normalization* [4, 51] to resolve this uncertainty, as well as all others that possibly arise when the order of instruction execution depends on run-time information. Program normalization refers to a set of established program transformations designed to facilitate program analysis and automatic parallelization. Many compilers [18] for multi-core processors, for example, include multiple normalization passes.

To resolve the uncertainty in Fig. 6(a), we need to be in the position to persist the value of *sum* once we are sure the loop is over and *before* the state-save operation. Fig. 6(b) shows one way to achieve this. We add a *dummy write* consisting in a pair of **LOAD** and **STORE** instructions for variable *sum after* the loop. These instructions are inserted after code elimination steps and bear no impact on program semantics, but fix where in the code the data for *sum* is final, regardless of the value of i and N. We add a similar instruction *prior* to the loop to fix where the first read for *sum* occurs. We can now make both **STORE** on line 8 and the **LOAD** on line 2 target non-volatile memory without unnecessary overhead. All other operations now concern intermediate results that may be stored on volatile memory. As a result, i and N are no longer in control of what is the I_{wn} (I_{r1}) write (read) instruction that the transformation in Sec. 3.2 and Sec. 3.3 would consider.

The normalization step introduces an overhead. To reduce that, whenever possible we leverage information cached in registers. For example, in Fig. 6(b), the value for *sum* stored in a register in line 6 may be picked up later in line 8, instead of re-loading the value from main memory. Applying this kind of optimization is, however, not always possible, as the content of registers may be overwritten by other instructions that execute in between. In Sec. 6 we prove that, despite the overhead of normalization, ALFRED programs are more energy-efficient than their regular counterparts.

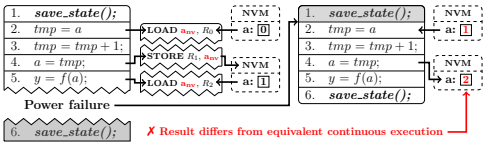
We apply similar normalization passes to resolve the uncertainty possibly arising with conditional statements, function calls, and when the span of computation intervals is undefined at compile time. Further optimizations to abate the overhead we generate are also possible depending on the programming construct [41].

5 MEMORY HANDLING

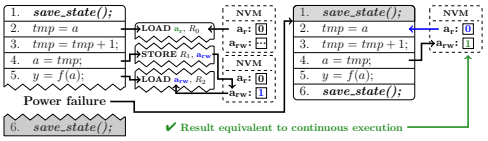
To make the techniques of Sec. 3 and Sec. 4 work correctly, we devise a custom memory layout that can be determined at compile-time and a schema to address the possible intermittence anomalies.

5.1 Memory Layout

Despite virtual memory tags ensure we can correctly group instructions, we still need to identify the addresses of the volatile or non-volatile versions of the same memory location. We address this problem by placing the volatile and non-volatile versions of a



(a) Example of an intermittence anomaly.



(b) Example of how to avoid the intermittence anomaly with memory versioning.

Figure 7: Example of an intermittence anomaly.

memory location at the same offset with respect to the corresponding base address. Note that the compiler treats the two segments as separate memory sections and makes them start at a fixed offset. This ensures that the volatile and non-volatile versions of the same memory location are at a fixed offset, too.

We can then express the address of the non-volatile version of a memory location as a function of the address of its volatile version, and vice versa. This allows us to allocate memory operations to either memory segment with ease, even in the presence of indirect accesses through pointers. To make an instruction that originally operates on volatile memory now target the non-volatile one, we add the offset between volatile and non-volatile segments to its target address. We operate the other way around when we make an instruction target volatile memory from the non-volatile one. When the instruction executes, it retrieves the address information that are unknown at compile-time and calculates the actual target.

5.2 Dealing with Intermittence Anomalies

Using mixed-volatile platforms, the re-executions of non-idempotent portions of code may cause intermittence anomalies [34, 42, 43, 45, 49], consisting in behaviors unattainable in a continuous execution. The problem possibly arises regardless of whether the code is written directly by programmers [34, 42, 43, 49] or is the result of the program transformations of Sec. 3.

Example. Consider the program of Fig. 7(a). Variable a is non-volatile. Following the state-save operation on line 1, the current value of variable a, that is 0, is initially retrieved from non-volatile memory. The execution continues and line 4 updates the value of variable a on non-volatile memory to 1. This is how a continuous execution would normally unfold.

Imagine a power failure happens right after the execution of line 4. When the device resumes as energy is back, the program restores the program state from non-volatile memory, which includes the program counter. The program then resumes from line 2, which is re-executed. As variable a on non-volatile memory retains the effects of the operations executed before the power failure, the value read by line 2 is now 1, that is, the value written in line 4 before the power failure in the previous power cycle. This causes

line 4 to produce a result that is unattainable in any continuous execution, as it updates the value of variable *a* to 2, instead of 1.

Many such situations exist that possibly cause erratic behaviors, including memory operations on the stack and heap [42, 43, 49].

Memory versioning. Intermittence anomalies happen whenever a power failure introduces a Write-After-Read (WAR) hazard [34, 42, 49] on a non-volatile memory location. In Fig. 7(a), the memory read of line 2 and the memory write of line 4 represent a WAR hazard for variable *a*. Several techniques exist to avoid the occurrence of intermittence anomalies [26, 34–36, 42, 43, 49]. In general, it is sufficient to break the sequences of instructions involved in WAR hazards [34, 42, 43, 49] so the involved instructions necessarily execute in different power cycles. Existing solutions place additional checkpoints [49] or enforce transactional semantics to specific portions of code [26, 34–36].

We use a different approach that tightly integrates with the compile-time operation of ALFRED. First, to reduce the number of instructions possibly re-executed, every call to a state-save operation in ALFRED systematically dumps the state on non-volatile memory, regardless of the current energy level. This is different than in many checkpoint systems, where the decision to take a checkpoint is subject to current energy levels [7, 8, 11, 46]. The overhead we impose by doing this is very limited, as state-save operations are limited to register file and program counter after applying the transformations of Sec. 3.

For each computation interval, we then create two versions of each non-volatile memory location possibly involved in a WAR hazard. One version is a *read-only copy* and contains the result produced by previous computation intervals; the other version is a *read-and-write copy* and contains the result of the considered computation interval. We direct the memory read (write) instructions to the read-only (read-and-write) copy. This ensures that in case of a re-execution, the read operations access the values produced by the previous computation interval, as the (partial) results of the current computation interval remain invisible in the read-and-write copies. When transitioning to the next computation interval, the read-only and read-and-write copies are swapped to allow the next computation interval to access the (now, read-only) data of the computation interval just concluded.

Fig. 7(b) shows how this solves the intermittence anomaly of Fig. 7(a). Line 2 reads variable *a*'s read-only copy, whereas line 4 writes variable *a*'s read-and-write copy. Line 4 accordingly reads variable *a*'s read-and-write copy, as it needs the data that line 4 produces. If a power failure happens after line 4 and line 2 is eventually re-executed, that read operation still targets *a* read-only copy, which correctly reports 0. Instead, after swapping the two copies, the next computation interval correctly accesses the copy of variable *a* that reports value 1, equivalently to a continuous execution.

We apply this technique as a further code processing step, as shown in stage ⟨5⟩ of Fig. 1. First, we identify the WAR hazards. For each memory write instruction I_w on a non-volatile memory location with tag x, we check if there exists a memory read instruction I_r such that *i)* I_r targets a non-volatile memory location with the same memory tag x, and *ii)* I_r may execute before I_w, that is, I_r happens before I_w in the control-flow graph. If such I_r exists, the pair (I_w, I_r) represents a WAR hazard.

Next, we create the read-only and read-and-write copies by doubling the space that the compiler normally reserves to the data structure x refers to. As we allocate the two copies in contiguous memory cells, their relative offset is fixed and may be used at compile time to direct the memory operation to either copy. We then make I_r target the read-only copy, together with every memory read instruction that operate on x and executes before I_w. In contrast, we make I_w target the read-and-write copy of x, together with all corresponding memory read instructions that happen after I_w. As this processing occurs after program normalization, the compile-time uncertainty in the order of instruction execution or in the span of computation intervals is already resolved at this stage.

6 EVALUATION

Our evaluation of ALFRED considers multiple dimensions. We describe next the experimental setup and the corresponding results.

6.1 Setting

We opt for system emulation over hardware-based experimentation, as it enables better control on experiment parameters and allows us to carefully reproduce program execution and energy patterns across ALFRED and the baselines we consider. Because of the highly non-deterministic behavior of energy sources, achieving perfect reproducibility is extremely challenging using real devices [22].

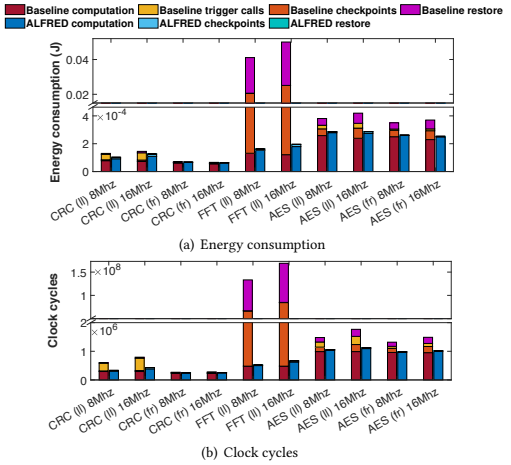
Tool and implementation. We use ScEpTIC [38, 43], an open-source extensible emulation tool for intermittent programs. ScEpTIC emulates the execution of the LLVM Intermediate Representation (IR) [33] of a source code and provides bindings for implementing custom extensions to *i)* apply program transformations and *ii)* map specific performance metrics of the IR to those of machine-specific code, for example, to measure energy consumption.

ScEpTIC organizes the LLVM IR into a set of Abstract Syntax Trees (ASTs), one for each function in source code. Each of these ASTs is generated by augmenting the original LLVM AST with dedicated ScEpTIC elements, which represent information on the emulated instructions and architectural elements, such as I/O operations and registers. We implement the pipeline of Fig. 1 from stage ⟨3⟩ onwards as a set of further transformations of these ASTs. A detailed description of this implementation is available [41], along with an open-source prototype release of our ScEpTIC extension implementing ALFRED transformations [39].

We also implement a machine-specific ScEpTIC extension to map the execution of the IR to the energy consumption of the MSP430-FR5969 [28], a low-power MCU that features an internal and directly-addressable FRAM as non-volatile memory. The MSP430-FR5969 is often employed for intermittent computing [8, 34, 35, 46, 49]. Our extension takes as configuration parameters the energy consumption per clock cycle of various operating modes of the MSP430-FR5969 [28], such as regular computation, (non-)volatile memory read/write operations, and peripheral accesses.

Dimension	Possible instances
Memory configuration	Volatile, NonVolatile
Checkpoint call placement	Loop-Latch, Function-Return, IdempotentBoundaries
Checkpoint execution	Probe, Execute

Figure 8: Design dimensions for baselines.



(a) Energy consumption

(b) Clock cycles

Figure 9: Energy consumption and number of clock cycles comparing ALFRED with a baseline using Volatile, Probe, and either Loop-Latch or Function-Return. *For a baseline, 'll' or 'fr' indicate Loop-Latch or Function-Return.*

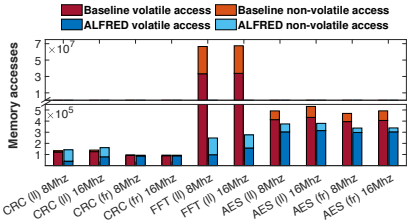


Figure 10: Memory accesses comparing ALFRED with a baseline using Volatile, Probe, and either Loop-Latch or Function-Return.

Benchmark	Baseline VM (bytes)	Baseline NVM (bytes)	ALFRED VM (bytes)	ALFRED NVM (bytes)
CRC (ll) 8Mhz	812	1688	6	850
CRC (ll) 16Mhz	812	1688	26	850
CRC (fr) 8Mhz	812	1636	26	810
CRC (fr) 16Mhz	812	1636	30	810
FFT (ll) 8Mhz	16708	33514	64	29082
FFT (ll) 16Mhz	16708	33514	2188	29082
AES (ll) 8Mhz	1276	2614	40	1334
AES (ll) 16Mhz	1276	2614	42	1334
AES (fr) 8Mhz	1276	2614	58	1338
AES (fr) 16Mhz	1276	2614	62	1338

Figure 11: Volatile memory (VM) and non-volatile memory (NVM) in ALFRED against a baseline using Volatile, Probe, and either Loop-Latch or Function-Return.

Baselines and benchmarks. We compare ALFRED with checkpoint mechanisms that instrument programs automatically [11, 36, 46, 49] by placing calls to a checkpoint library at specific places in the code. We do not consider, instead, checkpoints mechanisms that use interrupts to trigger the execution of checkpoints [7, 8, 29–31], including TICS [31] and the work of Jayakumar et. al [30], as checkpoints do not execute at pre-defined places in the code and thus boundaries of computation intervals cannot be identified. The latter is required for ALFRED to apply the transformations of Sec. 3.

Due to the variety of existing compile-time checkpoint systems, we abstract the key design dimensions in a framework that allows us to instantiate baselines that correspond to existing works, while retaining the ability to explore configurations not strictly corresponding to available systems. Fig. 8 summarizes these dimensions.

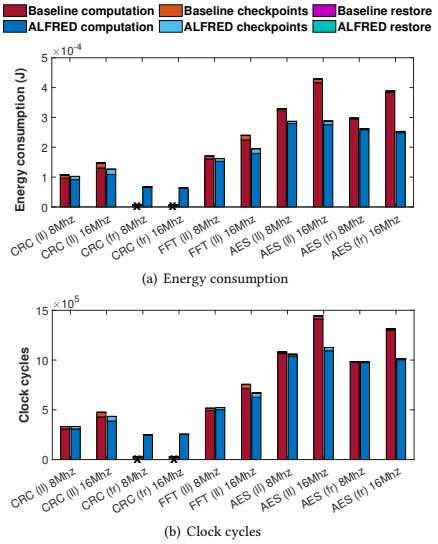
On such design dimension is the *memory configuration*. We consider two possible instances, Volatile and NonVolatile. Volatile allocates the entire main memory onto volatile memory. To ensure forward progress, each checkpoint must therefore save the content of main memory, register file, and special registers onto non-volatile memory. This is the case, for example, in Mementos [46] and HarvOS [11]. Instead, the NonVolatile instance allocates the entire main memory onto non-volatile memory. Here checkpoints may be limited to saving the content of the register file and program counter onto non-volatile memory, as main memory is already non-volatile. This is the case of Ratchet [49].

A given memory configuration is typically coupled to a dedicated *strategy for placing checkpoint calls* in the code. Systems that only use volatile main memory, as in Volatile, may place checkpoints using the Loop-Latch or Function-Return placement strategies of Mementos [46]. Systems that only use non-volatile main memory, as in NonVolatile, place checkpoints using the strategy of Ratchet [49]. This entails identifying idempotent sections of the code and placing checkpoint calls at their boundaries. We accordingly call this strategy IdempotentBoundaries. This ensures that intermittence anomalies are solved by construction, as re-execution of code only occurs across idempotent sections of code.

Once checkpoint calls are placed in the code, the *checkpoint execution policy* dictates the conditions that possibly determine the actual execution of a checkpoint. Indeed, a checkpoint call may systematically cause a checkpoint to be written on non-volatile memory, or rather probe the current energy levels first, for example, through an ADC query, and postpone the execution of a checkpoint if energy is deemed sufficient to continue without it. The former kind of behavior, which we call Execute, is the case of Ratchet [49], Chinchilla [36], and TICS [31] when it relies on checkpoints manually placed by developers; the latter kind of behavior we call Probe and reflects HarvOS [11] and Mementos [46].

A combination of memory configuration, strategy for placing checkpoint calls, and checkpoint execution policy represents the single baseline. Note that not all combinations of these dimensions are necessarily meaningful. For instance a NonVolatile memory configuration necessarily requires checkpoints to behave in an Execute manner, or the risk of intermittence anomalies would be too high and the overhead to address them correspondingly prohibitive [45]. As ALFRED requires as input a placement of state-saving operations, when comparing with a certain baseline we use the same such placement. Moreover, being the FRAM performance and energy consumption affected by the MCU operating frequency [28], we consider both $8Mhz$ and $16Mhz$ clock configurations.

Applications deployed onto battery-less devices typically consist in a sense-process-transmit loop [1, 13, 27]. Checkpoint techniques



(a) Energy consumption

(b) Clock cycles

Figure 12: Energy consumption and number of clock cycles comparing ALFRED with a baseline using NonVolatile, Execute, and either Loop-Latch or Function-Return.

and memory configurations mainly affect processing, whereas sensing and transmissions impose the same overhead regardless of the former. For this reason, similar to related literature, we focus on processing functionality and consider a diverse set of benchmarks commonly used in intermittent computing [7, 8, 26, 29, 43, 46, 49]: Cyclic Redundancy Check (CRC) for data integrity, Advanced Encryption Standard (AES) for data encryption, and Fast Fourier Transform (FFT) for signal analysis. We use Clang version 7.1.0 to compile their open-source implementations, as available in the MiBench2 [25] suite, using the default compiler settings. The binaries output by the compiler never exceed 30*kB*.

Metrics and energy patterns. We focus on *energy consumption* and *number of clock cycles* necessary to complete a fixed workload. Being harvested energy scarce, the former captures how batteryless devices perform when deployed and represents an indication of the perceived end-user performance [1, 13, 27]. The latter allows us to identify how the overhead of ALFRED affects performance, as it mainly consists in the additional instructions required to address the compile-time uncertainties, as described in Sec. 4. Note that the two metrics are not necessarily proportional, because non-volatile memory accesses may require extra clock cycles and consume more energy than accesses to volatile memory [28]. ALFRED may also introduce an overhead in the form of additional memory occupation, as the same data may need space in both volatile and non-volatile memory. To measure this, we keep track of the use of *volatile/non-volatile memory spaces* during the execution. To gain a deeper insight into the performance trends we also record *volatile/non-volatile memory accesses*, and the execution of *checkpoint and restore operations*.

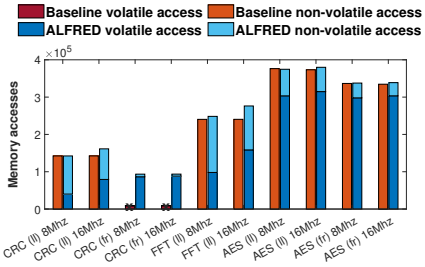


Figure 13: Memory accesses comparing ALFRED with a baseline using NonVolatile, Execute, and either Loop-Latch or Function-Return.

Patterns of ambient energy harvesting may be simulated using IV surfaces [21, 22] or by repetitively simulating power failures after a pre-determined number of executed clock cycles [40, 49]. The former makes simulated power failures happen at arbitrary points in times and provides little control on experiment executions, making it difficult to sweep the parameter space. The latter may be tuned according to statistical models, and offers better control on experiment executions by properly tuning model parameters. The behavior of ALFRED is largely independent of the specific number of executed clock cycles between consecutive power failures; we therefore opt for the second option.

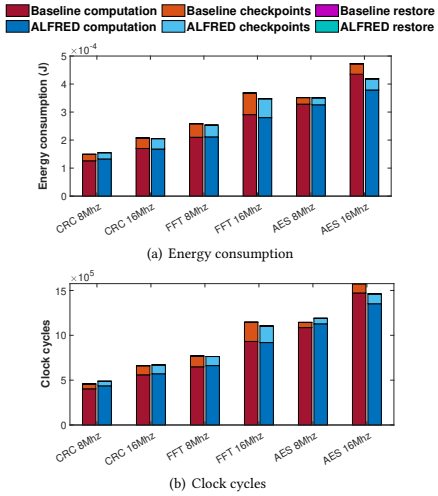
We model an RF energy source. To determine the number of executed clock cycles between two power failures, we rely on the existing measurements from ten real RF energy sources used for the evaluation of Mementos [46], which features a MCU configuration compatible with our setup. To evaluate multiple scenarios, including the worst and best possible ones, we execute each benchmark considering the minimum, average, and maximum number of executed clock cycles between power failures, modeled after the aforementioned real measurements. We report on the results obtained in the average scenario, as there is no sensible difference among the three scenarios. Note that, when using the Probe strategy, we make sure that the last checkpoint call before a power failure is the one that does save a checkpoint, as this represents the same behavior of real scenarios.

6.2 Results

We consider three combinations of the design dimensions of Fig. 8.

Checkpointing from volatile memory. We begin comparing with a baseline configuration using Volatile, Probe, and either Loop-Latch or Function-Return. This configuration represents Mementos [46] and solutions inspired by its design [11, 36].

Fig. 9 shows the results we obtain. Fig. 9(a) shows how, depending on the benchmark, ALFRED provides up to several-fold improvements in energy consumption to complete the fixed workload. CRC computation is the simplest benchmark and has little state to make persistent. The improvements are marginal here, especially when using Function-Return as the checkpoint placement strategy, which is unsuited to the structure of the code in the first place. The improvements grow as the complexity of the code increases. Computing FFTs is the most complex benchmark we consider, and



(a) Energy consumption

(b) Clock cycles

Figure 14: Energy consumption and number of clock cycles comparing ALFRED with a baseline using NonVolatile, Execute, and IdempotentBoundaries.

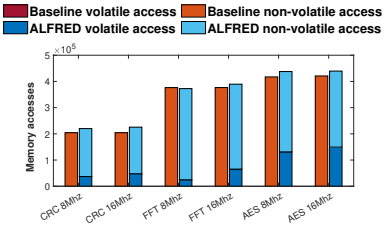


Figure 15: Memory accesses comparing ALFRED with a baseline using NonVolatile, Execute, and IdempotentBoundaries.

Benchmark	Baseline VM Size (Bytes)	Baseline NVM Size (Bytes)	ALFRED VM Size (Bytes)	ALFRED NVM Size (Bytes)
CRC 8Mhz	0	826	6	854
CRC 16Mhz	0	826	16	854
FFT 8Mhz	0	16730	40	29074
FFT 16Mhz	0	16730	1116	29074
AES 8Mhz	0	1294	24	1342
AES 16Mhz	0	1294	40	1342

Figure 16: Volatile memory (VM) and non-volatile memory (NVM) in ALFRED against a baseline using NonVolatile, Execute, and IdempotentBoundaries

the improvements are largest in this case. These observations are confirmed by the measurements of clock cycles, shown in Fig. 9(b).

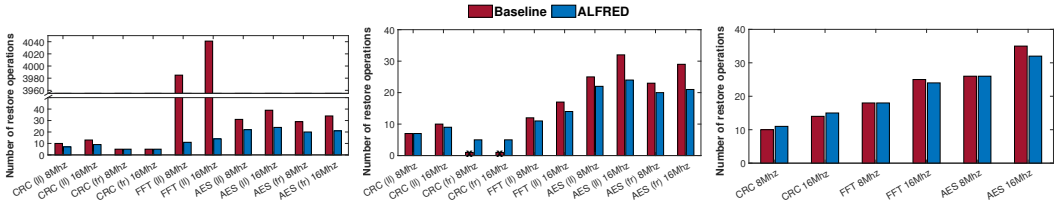
Fig. 10 provides a finer-grained view on the results in this specific setting. The small state in CRC corresponds to the fewest number of memory accesses, especially in volatile memory, as little data is to be made persistent to cross power failures. In both AES and FFT, ALFRED greatly reduces the number of memory accesses. Checkpoint operations in these benchmarks must load a significant amount of data from volatile memory and copy it to non-volatile memory for creating the necessary persistent state. These accesses are not necessary in ALFRED, as data is made persistent as soon as it becomes final; therefore, checkpoint operations do not process main memory, but only register file and program counter. As for the nature of memory accesses, ALFRED can promote, on average, 65% of the accesses the baseline executes on non-volatile memory to volatile memory instead, with a minimum of 20% in *CRC* at 8*Mhz* with a Loop-Latch configuration and a maximum of 95% in *CRC* with a Function-Return configuration. This is a key factor that grants ALFRED better energy performance.

Fig. 11 reports on the use of volatile/non-volatile memory. In the baseline, the state to be preserved across power failures includes the entire volatile memory, the register file, and special registers. Requiring to double-buffer the state saved to non-volatile memory, its use in the baseline amounts to more than double the use of volatile memory. In both CRC and AES, ALFRED requires to double buffer less than *4%* of the program state to avoid intermittence anomalies, resulting in a drastically lower use of non-volatile memory. Interestingly, despite a significant improvement in energy consumption, ALFRED promotes very few memory locations to volatile memory. These correspond to the memory locations that are most frequently accessed, as shown in Fig. 10.

Moving to non-volatile memory. Fig. 12 shows the results we obtain comparing with configuration using NonVolatile, Execute, and either Loop-Latch or Function-Return. This combination represents a hybrid solution combining features of several existing systems [11, 36, 46]. As Loop-Latch and Function-Return do not necessarily guarantee that intermittence anomalies cannot occur, we lend our versioning technique, described in Sec. 5, to the baseline. The major difference between ALFRED and the baseline, therefore, is in the use of volatile or non-volatile memory.

Fig. 12(a) shows that the program transformations we devise are effective at improving the energy performance of intermittent programs. Significant improvements are visible across all benchmarks. Configurations exist where the baseline cannot complete the workload using the energy patterns we consider, as in the case of the CRC benchmark when using Function-Return to place checkpoints. In contrast, ALFRED reduces energy consumption to an extent that allows the workload to successfully complete.

The corresponding results in the number of executed clock cycles, shown in Fig. 12(b), enables a further observation. When running at 16*Mhz*, the baseline shows a significant increase of clock cycles, at least 20% with respect to the same benchmark running at 8*Mhz*. The cause of this increase is in the extra clock cycles required to access the FRAM when the MCU is clocked at 16*Mhz*. In the same scenarios, ALFRED shows a lower increase of clock cycles when comparing the 8*Mhz* and 16*Mhz* configurations, especially in the AES benchmark. Rather than massively employing non-volatile memory, ALFRED switches to volatile memory whenever possible within a computation interval. This not only reduces the clock cycles spent waiting for non-volatile memory access, but also enables energy savings in the operations that involve temporary data or intermediate results that do not need persistency.



(a) Baseline using Volatile, Probe, Loop-Latch or Function-Return (b) Baseline using NonVolatile, Execute, Loop-Latch or Function-Return (c) Baseline using NonVolatile, Execute, Idempotent-Boundaries

Figure 17: Restore operations to complete the fixed workload in ALFRED compared to the three baselines.

Fig. 13 confirms this reasoning, showing that ALFRED promotes an average of 65% of the non-volatile memory accesses in the baseline to the more energy-efficient volatile memory. This functionality grants ALFRED the completion of the CRC benchmark when using the Function-Return configuration. As the baseline directs all memory accesses to non-volatile memory, the resulting energy consumption causes CRC to be stuck in a livelock, as energy is insufficient to reach a checkpoint that would enable forward progress. This situation is called "non-termination" bug [17]. Instead, in the case of CRC, ALFRED promotes more than 95% of the non-volatile memory accesses in the baseline to volatile memory. This significantly reduces the energy consumption of memory accesses to an extent that allows ALFRED to complete the workload.

Note that the use non-volatile memory in the baseline is the same as ALFRED, shown in Fig. 11, as they employ the same technique to avoid intermittence anomalies. The difference in memory occupation consists in the data that ALFRED allocates onto volatile memory, which ultimately yields lower energy consumption.

Ruling out intermittence anomalies. We compare the performance of ALFRED with a configuration using NonVolatile, Execute, and IdempotentBoundaries, as in Ratchet [49]. Because of the specific placement of checkpoint calls and the Execute policy, intermittence anomalies cannot occur by construction. ALFRED and the baseline here only differ in memory management.

Fig. 14 shows the results. Fig. 14(a) illustrates the performance in energy consumption; this time, the improvements of ALFRED are generally less marked than those seen when using Loop-Latch or Function-Return to place checkpoints. The results in the number of executed clock cycles are coherent with these trends, as illustrated in Fig. 14(b). This is because IdempotentBoundaries tends to create much shorter computation intervals, sometimes solely worth a few instructions; therefore, ALFRED has fewer opportunities to operate on the energy-efficient volatile memory. ALFRED still improves the energy efficiency overall, especially for the AES benchmark and the configurations running at 16*Mhz*. At this clock frequency, non-volatile memory operations induce higher overhead due to the necessary wait cycles. Sparing operations on non-volatile memory allows the system not to pay this overhead.

Fig. 15 and Fig. 16 provide an assessment on ALFRED's ability to employ volatile memory whenever convenient. ALFRED promotes the use of volatile memory from the non-volatile use in the baseline in up to 30% of the cases. The impact of this, however, is more limited here because of the shorter computation intervals, as discussed above. In this plot, it also becomes apparent that sometimes, the total number of memory accesses in ALFRED is higher than in the baseline. This is a combined effect of the program transformation techniques of Sec. 3 and of the normalization passes in Sec. 4. The increase in the total number of memory accesses, however, does not yield a penalty in energy consumption, as a significant fraction of these added accesses operate on volatile memory.

These results are confirmed in Fig. 16. Despite being the program partitioned in non-idempotent code sections, our techniques to address compile-time uncertainties introduce intermittence anomalies that require ALFRED to double-buffer a portion of the program state. This situation is particularly evident with FFT. Fig. 16 provides additional evidence of how ALFRED employs volatile memory for frequently-accessed data, which ultimately yields lower energy consumption across all benchmarks executed at 16*Mhz*.

Restore operations. We complete the discussion by showing in Fig. 17 the number of restore operations executed in ALFRED compared to those in the three baseline configurations we consider.

The plots demonstrate that the better energy efficiency provided by ALFRED allows the system to restore the state less times. This trend is especially visible in Fig. 17(a) and Fig. 17(b). As a result, ALFRED shifts the available energy budget to useful application processing, leading to workloads that finish sooner compared to the performance offered by the baselines.

7 CONCLUSION

ALFRED is a virtual memory abstraction for intermittent computing that spares programmers the need to manage application state across memory facilities, and efficiently employ volatile and non-volatile memory to improve energy consumption, while ensuring forward progress. The mapping from virtual to volatile or non-volatile memory is decided at compile time to use volatile memory whenever possible because of the lower energy consumption, resorting to non-volatile memory to ensure forward progress. In contrast to existing works, the memory mapping is not fixed at variable level, but is adjusted at different places in the code, based on read-/write patterns and program structure. Our evaluation indicates that, depending on the workload, ALFRED provides several-fold improvements in energy consumption compared to the multiple baselines we consider, leading to a similar improvement in the number of restore operations required to complete a fixed workload.

Acknowledgments. We thank the shepherd and reviewers for the feedback received on the initial submission. This work was supported by the Google Faculty Award programme and by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys '20)*.

[2] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[3] S. Ahmed, M. H. Bhatti, N. A. Alizai, J. H. Siddiqui, and L. Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*.

[4] Z. Ammarguellat. 1992. A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering* (1992).

[5] J. A. Anderson and G. J. Lipovski. 1974. A Virtual Memory for Microprocessors. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA '75)*.

[6] A. R. Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* (2018).

[7] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[8] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[9] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (2018).

[10] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[11] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[12] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SENSYS)*.

[13] Q. Chen, Y. Liu, G. Liu, Q. Yang, X. Shi, H. Gao, L. Su, and Q. Li. 2017. Harvest Energy from the Water: A Self-Sustained Wireless Water Quality Sensing System. *ACM Transactions on Embedded Computing Systems* (2017).

[14] J. Choi, M. Burke, and P. Carini. 1993. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*.

[15] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. *SIGOPS Operating Systems Review* (2016).

[16] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[17] A. Colin and B. Lucia. 2018. Termination Checking and Task Decomposition for Task-based Intermittent Programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*.

[18] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer* (2009).

[19] P. J. Denning. 1970. Virtual Memory. *Comput. Surveys* (1970).

[20] F. Fraternali, B. Balaji, Y. Agarwal, L. Benini, and R. Gupta. 2018. Pible: Battery-Free Mote for Perpetual Indoor BLE Applications. In *Proceedings of the 5th Conference on Systems for Built Environments (BUILDSYS)*.

[21] M. Furlong, J. Hester, K. Storer, and J. Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*.

[22] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*.

[23] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[24] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–13.

[25] M. Hicks. 2016 (last access: Oct 15th, 2021). MiBench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[26] M. Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA)*.

[27] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[28] Texas Instruments. 2017 (last access: Oct 15th, 2021). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[29] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[30] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Transactions on Embedded Computing Systems* (2017).

[31] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak. 2020. Time-Sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[32] W. Landi and B. G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*.

[33] llvm 2003 (last access: Oct 15th, 2021). The LLVM Compiler Infrastructure. https://llvm.org/.

[34] B. Lucia and B. Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[35] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[36] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[37] K. Maeng and B. Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[38] A. Maioli. 2021 (last access: Oct 15th, 2021). ScEpTIC documentation and source code. http://sceptic.neslab.it/.

[39] A. Maioli. 2021 (last access: Oct 15th, 2021). ScEpTIC extension implementing a prototype of ALFRED pipeline. http://alfred.neslab.it/.

[40] A. Maioli and L. Mottola. 2020. Intermittence Anomalies Not Considered Harmful. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20)*.

[41] A. Maioli and L. Mottola. 2021 (last access: Oct 15th, 2021). ALFRED: Virtual Memory for Intermittent Computing. https://arxiv.org/abs/2110.07542 . arXiv:2110.07542

[42] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[43] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021)*.

[44] A. Y. Majid, C. Delle Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak. 2020. Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices. *ACM Transactions on Sensor Networks* (2020).

[45] B. Ransford and B. Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*.

[46] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[47] E. Ruppel and B. Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-harvesting Computing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[48] E. Sazonov, H. Li, D. Curry, and P. Pillay. 2009. Self-Powered Sensors for Monitoring of Highway Bridges. *IEEE Sensors Journal* (2009).

[49] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[50] K. Vijayaraghavan and R. Rajamani. 2010. Novel Batteryless Wireless Sensor for Traffic-Flow Measurement. *IEEE Transactions on Vehicular Technology* (2010).

[51] T. Wang, X. Su, and P. Ma. 2008. Program Normalization for Removing Code Variations. In *Proceedings International Conference on Software Engineering*.

[52] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.

CHAPTER 13

Annexes

Framework-based Dynamic Voltage and Frequency Scaling for Intermittent Computing

Andrea Maioli*, Kevin Quinones*, Saad Ahmed+, Luca Mottola*†
*Politecnico di Milano (Italy), +Georgia Institute of Technology (US), †RI.SE Sweden

ABSTRACT

We propose a framework to efficiently design battery-less MCUs with Dynamic Voltage and Frequency Scaling (DVFS) capabilities. Battery-less devices are highly resource constrained and lack any DVFS controller due to its additional power draw, as they are powered only with the energy harvested from the environment and consume up to milliwatts of power. Hence, battery-less devices currently use statically-configured clock frequencies, which result in poor performance, as this makes devices unable to adapt their power consumption depending on fluctuations of harvested energy. Our design technique enables battery-less devices to efficiently use DVFS without relying on a dedicated controller. We propose two system designs and fabricate a DVFS-enable device. Compared to static frequencies configurations, our designs demonstrate up to a 170% reduction of energy consumption and a reduction of the workload completion time up to 12x.

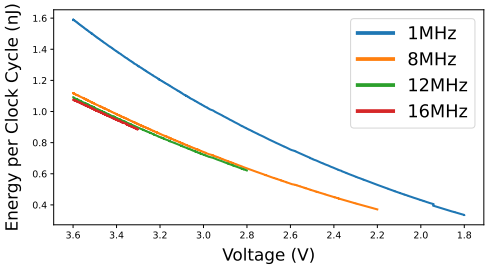


Figure 1: Measurements [4] of the energy consumption per clock cycle of various voltage and frequency ranges of the MSP430-G2553 [26].

1 INTRODUCTION

Ambient energy harvesting [9] enables battery-less devices to work as sustainable sensors for the Internet-of-Things [2, 9, 16, 21, 23, 52, 58], resulting in reduced maintenance costs and a lower environmental impact.

However, harvested energy is scarce and usually insufficient to power battery-less devices continuously. Despite the use of capacitors to buffer harvested energy and to smooth its flactuations, battery-less devices experience frequent and unpredictable power failures. Consequently, the resulting computation is intermittent: periods of active computation are interleaved by periods where they are powered off to recharge their energy buffer.

Power failures cause battery-less devices to immediately shut down, causing the loss of the computational and peripheral state. To ensure forward progress across power failures, multiple techniques [7, 8, 10, 34, 51, 57] allow battery-less devices to periodically save their state onto a non-volatile memory location, which is persistent across power failures. Restoring the saved state from non-volatile memory allows battery-less devices to resume the computation from where the state was saved. Similarly, re-initializing and restoring peripheral state require additional operations [11]. To complicate matters, when resuming the computation, battery-less devices may re-execute portions of programs that can potentially cause unexpected behaviours [45, 46, 55], which are unattainable in an equivalent continuous execution. Preventing this situation requires battery-less devices to save the state more frequently or to execute additional operations [46, 55].

System efficiency. These additional operations required to ensure intermittent computations across power failures introduce a significant computation and energy overhead. In fact, the device pauses the program execution and executes the additional operations required to save or restore the state from non-volatile memory, to re-initialize peripherals, and to avoid unexpected behaviours. Further, accesses to non-volatile memory are slower and less energy efficient than volatile memory accesses [30, 44, 45], further increasing the computation and energy overhead.

The number of instructions a device can execute within the active portion of a power cycle primarily depends on two factors: the amount of energy harvested from the environment and the device energy consumption. Despite using techniques such as MPPT [6] to maximize harvested energy, harvested energy is unpredictable and we can only control the device energy consumption. However, the increase in the energy consumption due to the execution of operations that enable intermittent computations cause a decrease in the number of instructions the device is able to execute within a single power cycle. To reduce the overhead of such operations and extend the computation done in a power cycle, the research community is focusing on improving the performace of state-saving operations [10, 39, 41, 59], memory accesses [35, 43, 44], specific workloads [18, 43], and peripheral state retention and accesses [11, 40]. However, these solutions are workload-specific [18, 43] or target specific elements of the computation [18, 44].

In general, the efficiency of a system depends on its energy consumption and on its computational speed, which are directly affected by the system operating voltage and frequency. However, identifying the setting that maximizes battery-less systems efficiency is non-trivial.

Let us consider the MSP430-G2553 [26] from Texas Instruments, a Micro Controller Unit (MCU) from the MSP430 family, that is, a family of ultra-low-power MCUs popularly used in intermittent computing. Fig. 1 depicts the energy consumption per clock cycle of four factory-calibrated operating frequencies. The higher the clock frequency, the faster is the computation and the lower is the energy consumption per clock cycle. For example, the clock frequency of

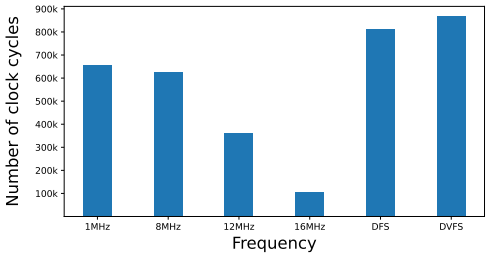
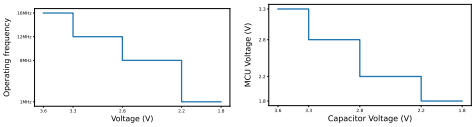


Figure 2: Clock cycles executed in a single discharge from $3.6V$ of a $100\mu F$ capacitor for various frequency configurations for the MSP430-G2553 [26].



(a) Dynamic Frequency Scaling (DFS) (b) Dynamic Voltage Scaling (DVS)

Figure 3: Dynamic Frequency Scaling (DFS) and Dynamic Voltage Scaling (DVS) techniques applied to the MSP430-G2553 [26] to improve system efficiency and prolongue system computation.

$16MHz$ is, on average, 47% more energy efficient and $16x$ faster than $1MHz$. However, compared to $1MHz$, $16MHz$ has a narrow operating voltage range.

Further, conversely from mainstream computation, where a stable power supply provides a constant voltage, battery-less devices use a capacitor as energy buffer, whose voltage eventually decreases due to the lack of new incoming harvested energy. Consequently, in battery-less systems, the reduced voltage range of higher frequencies pones severe limitations that potentially worsen the overall system performance. Fig. 2 shows the number of clock cycles executed by the MCU when no energy is harvested. Despite granting a faster and more efficient computation, the reduced operating voltage range of $16MHz$ results in $3.75x$ less number of clock cycles than the slower and less energy efficient $1MHz$. For this reason, identifying the most efficient system setting for battery-less devices is a non-trivial operation.

Increasing system performance. A common technique used to reduce the device energy consumption while ensuring the best possible performace is Dynamic Voltage and Frequency Scaling (DVFS). The key idea behind DVFS consists in dynamically tuning the device operating frequency and voltage to ensure it always operates in the most efficient setting.

In the example of Fig. 2, despite being the operating frequency with the highest energy consumption, $1MHz$ is the configuration that lead to the highest number of clock cycles executed within a single power cycle. Note that this happens due to the extended operating voltage range of $1MHz$, as we show in Fig. 1. Instead of statically configuring the system with the static frequency of $1Mhz$, we can dynamically alter the operating frequency and select at any instant the highest frequency supported by the current operating voltage, as Fig. 3(a) shows. This is recognized as Dynamic Frequency Scaling (DFS) and has the effect of always selecting the faster and most energy efficient operating frequency, resulting in a 24% performace increase with respect to $8MHz$, as Fig. 2 shows.

Further, in combination of DFS, we can lower the MCU operating voltage to the minimum possible voltage that allows using the selected operating frequency, as Fig. 3(b) shows. The resulting behaviour is recognized as DVFS and further increases the system performace by 7%, as Fig. 2 shows. As a result, applying DVFS to the MSP430-G2553 leads to a 32% performace increase, compared to the most performing static configuration.

Problem. The concept behind the DVFS technique we just described is simple and it is usually available in mainstream processors. However, applying DVFS to battery-less devices is non-trivial for multiple reasons.

First, DVFS is architecture-specific, as different MCU/CPUs have different hardware capabilities, operating voltages, and frequency ranges. Dynamically changing the operating voltage and frequency requires capabilities that need to be available both at hardware and software level. However, battery-less devices do not have such capabilities, as they consist in ultra-low-power MCUs with basic hardware capabilities. Despite common MCUs can be configured with multiple frequencies [26], they lack both hardware components and an operative system able to dynamically tune the operating frequency, let alone the operating voltage.

Second, the hardware and software components enabling DVFS must be very efficient and not increase significantly the MCU energy consumption, as harvested energy is scarce. Note that the power consumption of battery-less devices usually does not exceed the order of tens of mW [].

Third, differently from mainstream devices, battery-less devices are powered with an unstable power source that is subject to significant and frequent voltage flactuations. As such, the DVFS logic must account for such voltage changes and quickly adapt to them. This requires to constantly track the energy buffer voltage, whose probing through the MCU built-in ADC is extremely expensive [26, 43, 44].

Finally, DVFS aims at achieving the best possible performace for a given workload. The usual performace metric considered in mainstream computation is either the energy consumption or the computation time. Instead, intermittently-computing battery-less devices we aim at prolonging the lenght active portions of power cycles, even if this means using a less energy-efficient or slower operating frequency, as we describe in the example of Fig. 2.

Solution. As we point out in Sec. 2, very little research is available to apply DVFS to battery-less devices [5, 15]. The available works mainly target multi-core processors with DVFS hardware capabilities and focus on reaching power neutrality by tuning system power consumption to match harvested energy from solar panels.

In this paper we propose a framework to efficiency design battery-less devices with DVFS capabilities. Fig. 4 shows the principles behind our design framework, which we describe in Sec. 3. Our framework consists in two phases. First, at design time, system designers identify the performance windows setting, consisting in

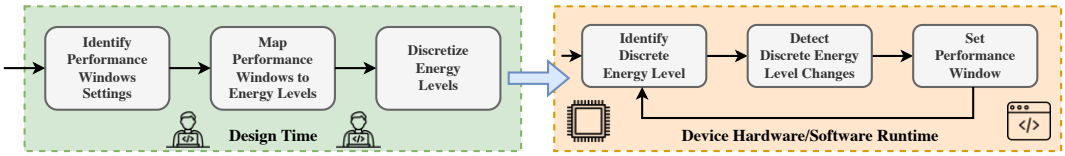


Figure 4: DVFS framework for battery-less devices.

the most efficient combinations of voltage and frequency. Each performace window is then mapped to the energy buffer level where it should be applied. These information is then used to design the MCU hardware and software components that provide the DVFS capability. The MCU needs to constantly check the energy buffer level and, when it detects a change, it changes accordingly its operating voltage and frequency to match the correspoding performace window setting.

In Sec. 4, we show how our framework allows designing DVFS-enabled battery-less devices and we provide two different system designs, D^2VFS and FBTC. We provide the schematics for both the two system designs and we fabricate FBTC.

Finally, in Sec. 5, we evaluate the two system designs against a MSP430-G2553 statically configured with four factory-calibrated frequencies. We show that D^2VFS and FBTC require a smaller energy buffer than the considered baselines and allow for up to 3.75x lower energy consumption and 12x faster execution time.

2 BACKGROUND

We first discuss existing works in the area of intermittent computing to high their limitations and challenges that has hampered the integration of DVFS techqniue for transiently powered systems.

2.1 Intermittent Computing

System powered by harvested energy are unable to gather enough energy to complete application execution in one charge of the cycle. As a result the system under goes numerous power failures in a single execution of the application. System support exists that enables batteryless devices to ensure forward progresss by either inserting checkpoints at compile using program analysis tecqhniuesor by employing additional hardware that allows them to check the current voltage of the system [7, 8]. Orthogonally, works also exist that expose APIs to the programmer to divide a batteryless application into set of atomic tasks with each task performs a lightweight checkpoint at the end of its execution [12, 38, 47]. These solutions, however, are focused more on reducing the computational/energy overhead incurred on maintaining forward progress of the application. Furthermore, these systems are designed with static configuration and are oblivious to the changing input energy and its effect on MCU frequency. Since the number of clock cycles available in an active period are constant, existing works belonging to both categories are focused on reducing the additional computations performed because of a checkpoint thus allowing the device to perform more useful computations. None of the existing system focus on reducing the energy consumption during program execution and increasing the number of computations performed by the batteryless device in the given energy budget. These systems are oblivious to the underlying MCU's frequency and its behavior with the changing input voltage and focus only on reducing the number of unnecessary computations to ensure energy-efficiency.

2.2 Dissecting DVFS

DVFS is a standard technique used in main-stream computers and comprise of two optimizations: voltage and frequency scaling. Each processor has different operational zones and each zone is defined by a frequency and voltage tuple (f,V); the frequency of operation and voltage required to operate on that frequency. This optimization is different from hibernation as it allows the device to conserve energy by consuming lower energy-per-cycle thus allowing the system to go farther on the same charge thus allowing the system to improve batterylife and increasing the length of active period for the device on a given energy budget. Modern computers have sophisticated software and hardware components that enable the fine-grained control over dynamic switching of different operational zones.

Existing literature has explored application of DVFS real-time embedded systems and energy havesting devices with a focus on adapting voltage and frequency scaling based on the power consumption of real-time tasks and ensuring that these tasks meet deadlines. Some of these works have also used DVFS in mixed criticality systems to help execute critical tasks in an efficient manner. Other works have also explored application of DVFS for power neutral systems where frequency scaling [5] and volatge scaling is employed to adapt system performance in response to the changing incoming energy [6, 15]. This allows the system to execute more number of instructions as compared to the static approaches. Although these solutions lay the foundation for applying DVFS in embedded systems, the design goals revolve around simulation and there is no concrete implementation for energy harvesting systems.

2.3 Challenges

Transiently-powered system harvest energy from the environment ot power themselves that has high spatio-temporal variability. Unfortunately, such techqniues can be deployed as-is for transiently-powered systems as they have limited energy at their disposal. Following are some of the chanllenges that have hampered the adoption of DVFS for intermittent computing devices.

Energy/Size Overhead Due to limited energy at their disposal, transiently-powered systems cannot support any new sophicated hardware/software support available in main-stream computers as it adds to the energy consumption of the system. Furthermore, it increases the form-factor of the device. Transiently-powered systems

are deployed in far-to-reach areas and are design to easily weave into the daily fabric of human life. To enable this, the size of these devices has to be small enough. Adding sophisticated hardware increases the form-factor of these devices in addition to consuming more energy.

Adaptation Main-stream computers assume energy as a "limited but continous" resource. This assumption fails for transiently-powered systems as it is "unlimited but variable". This requires these system to adopt reactive mechanisms to adapt according to the chaging energy condition in the environment and re-configure the system based on the new energy parameters. This require more system support thus burdening the energy buffer even further.

2.4 Way Forward

Figure 2 shows the number of clock cycles gained only by using the DVFS technqiue. We can clearly see that the number of clock cycles available for the system using DVFS are significantly more than the any of the static configurations. However, to take this benefit, we have to design the system as efficient as possible so that we can reduce the energy/size overhead of the DVFS tecqhniue while being adaptive to the chaging energy conditions.

3 DVFS FOR INTERMITTENT COMPUTING

We anticipate in Sec. 1 that applying DVFS to transiently-powered devices require to tackle multiple challenges due to the energy and computational constraints of such devices. We thus design a framework that allows system designers to implement DVFS into transiently-powered devices, which can be applied to any architecture and unlocks DVFS under a transiently-powered supply scenario.

Fig. 4 depicts the main steps of our framework.

Design time. At design time, system designers firstly identify the performace windows setting for the target MCU, which consists in the combinations of voltage and frequency settings that grant the most efficient computation. Note that in this phase, system designers have to identify a reasonable subset of the possible operating frequencies of a given MCU and map them to their minimum operating voltage, that is, the voltage granting the lowest possible energy consumption. For example, for the MSP430-G2553 [26], we can consider the four factory-calibrated operating frequencies, thus identifying four performace windows: (i) $16MHz$ at $3.3V$, (ii) $12MHz$ at $2.8V$, (iii) $8MHz$ at $2.2V$, and (iv) $1MHz$ at $1.8V$.

Next, system designers have to map the identified performance windows to an energy level of the energy buffer, that is, a voltage range of the energy buffer. For example, for the four performace windows of the MSP430-G2553 [26], we identify the following energy levels: (i) $16MHz$ in the range $3.6V - 3.3V$ (ii) $12MHz$ in the range $3.3V - 2.8V$, (iii) $8MHz$ in the range $2.8V - 2.2V$, and (iv) $1MHz$ in the range $2.2V - 1.8V$. Note that the energy buffer level associated to the performace windows should not overlap.

Now system designers need to design and identify the hardware/-software components required to apply the identified performance windows.

For transiently-powered MCUs, this step primarely requires system designers to discretize the energy levels associated to each performace window, as otherwise the MCU would need to constantly probe the energy buffer level, thus wasting precious energy. The resulting hardware/software system relies on discrete energy levels to apply a given performace window. A discrete energy level (DEL) consists in a discretization of a continuous energy level level of a performace window, and can be implemented as a digital signal that specifies to the MCU the energy level without having it constantly probe the energy buffer level. Such digital signal can be generated with circutry external to the MCU or by reling on existing components internal to the MCU, such as a voltage detector.

Further, here system designers need to include in their design all the components that allows to tune the device operating voltage and frequency. For example, to set the operating voltage, an external voltage regulator may be required, as this is usually not included inside the MCU package. Instead, system designers can rely on existing MCU functionalities to change the operating frequency through software components. For example, in the MSP430-G2553 [26], the operating frequency can be set by changing the value of a specific register through software.

We describe in Sec. 4 two different designs that discretize the energy levels of the performace windows for the MSP430-G2553 [26].

System Runtime. The overall system runtime applies the DVFS technique. it periodically identifies the current DEL and detects when there is a change, meaning that the performace window has changed. When such event occurs, the system runtime needs to change the MCU setting and make it operate using the voltage and frequency associated to the new performace window. Note that the DVFS system functionalities are split between software and hardware. DEL change detection, operating voltage changes, and operating frequency changes may rely only on hardware or software components, depending on the choices of system designers.

Fig. 6 shows an example of the system runtime, where we consider the performace windows of the MSP430-G2553 [26] that we previously identify. On startup, the system is set to the first performace window, that is, $16MHz$ and $3.3V$. The current DEL is the one associated to $3.6V - 3.3V$. When the energy buffer voltage drops to $3.3V$, the system detects a new DEL, that is, the one associated to $3.3V - 2.8V$. Consequently, the system sets the MCU operating frequency to $12MHz$ and the operating voltage to $2.8V$.

Note that the order to set the operating voltage and frequency is crucial to ensure correct operations: setting the operating voltage to $2.8V$ as first operation would cause the system to shut down, as it is below the minimum operating voltage required by $16MHz$, that is, $3.3V$. When scaling the frequency down, the system needs to scale down the frequency and then the voltage. Counversely, when scaling the frequency up, the system needs to scale up the voltage and then the frequency.

The system repeats the same behaviour when the DEL changes to the one associated to $2.8V - 2.2V$ ($2.2V - 1.8V$), by setting the operating voltage to $2.2V$ ($1.8V$) and the MCU operating frequency to $8MHz$ ($1MHz$).

The system has a similar behaviour when the energy buffer level increases: the operating frequency and voltage are increased instead of decreased. Here, however, to avoid flactuations of performace windows, the system needs to delay the increase of the operating frequency and voltage. This is required because, when increasing

to a higher performace window, that is, a performace window with a higher frequency, the system energy consumption increases. If the device is unable to harvest enough energy, the energy level decreases due to the increaase in the energy consumption, and the system would scale the operating setting back to the previous performace window. As now the system energy consumption decreases, the energy level may increase, repeating this pattern all over again. Consequently, the system would be stuck in constantly chaning the performace window instead of executing useful computation.

The delay necessary to avoid flactuations depends on the system design, and may require the system to offset the changes to higher performace windows by the span of an entire DEL. We describe in Sec. 4 two different approaches that we consider in our DVFS designs for the MSP430-G2553 [26].

4 IMPLEMENTATION

We use our framework to devise two different system designs, Discrete Dynamic Voltage and Frequency Scaling (D^{2}VFS) and Fixed Boot Threshold Circuit (FBTC). We describe in this section the two designs, their components, and their logic. Both system designs consider the MSP430-G2553 [26] as target MCU and the TPS62740 [28] as voltage regulator.

4.1 D^{2}VFS

Fig. 5 shows the system design of D^{2}VFS, where Fig. 5(a) depicts the core logic of D^{2}VFS and Fig. 5(b) shows the corresponding schematics.

We consider four performance windows for the MSP430-G2553 [26]: (i) $16MHz$ at $3.3V$ when the capacitor voltage (V_{cap}) is in the range $3.6V-3.3V$ (ii) $12MHz$ at $2.8V$ when V_{cap} is in the range $3.3V-2.8V$, (iii) $8MHz$ at $2.2V$ when V_{cap} is in the range $2.8V-2.2V$, and (iv) $1MHz$ at $1.8V$ when V_{cap} is in the range $2.2V-1.8V$.

First, D^{2}VFS discretizes the energy level at the Energy Buffer Level Detection stage of Fig. 5(a). In such step, D^{2}VFS relies on four voltage comprator of the BU49XXG [53] series from Texas Instruments, one for each voltage associated to the performance windows, as shown in Fig. 5(b). Each comparator takes as input the capacitor voltage V_{cap} and outputs a signal that specifies if V_{cap} is higher than the comparator threshold.

D^{2}VFS has a hardware interrupt driver that signals the MCU when the performance window changes. The interrupt driver consists in three components: (i) a flip flop D (SN74LV175A [24] of Fig. 5(b)) that keeps track of the current performance window (Previous Energy Buffer State of Fig. 5(a)), (ii) a 4-bit magnitude comparator (74HC85 [49] of Fig. 5(b)) that compares the saved performance window against the one detected in the energy buffer level detection step (Energy Buffer State Comparator of Fig. 5(a)), and (iii) a AND gate (SN74AUP1G08 [29] of Fig. 5(b)) that triggers the update of the performance window saved in the Previous Energy Buffer State stage (Save Energy Buffer State Controller of Fig. 5(a)).. The voltage detectors of the energy buffer level detection send their signals to the interrupt driver. The energy buffer state comprator compares the saved energy buffer state against the current one, that is, the one detected in the energy buffer level detection.

When a change is detected, the energy buffer state comprator triggers an interrupt to the MCU, which triggers the execution of the DVFS driver at software level. The DVFS driver verifies the current energy buffer level by looking at the signals of the energy buffer level detection and it identifies the new performance window parameters. Then, it sets the new performance window by setting the new MCU operating frequency and updating the GPIOs controlling the voltage regulator output.

Note that, when the performance window increases to a higher frequency, the voltage regulator output is updated before switching to a higher frequency. Instead, the DVFS driver does the opposite when switching to a lower frequency.

D^{2}VFS runtime behaviour. Fig. 6 shows an example of the system behaviour. The capacitor voltage V_{cap} is initially set to $3.6V$, and the DVFS driver sets the voltage regulator to $3.3V$ and the MCU operating frequency to $16MHz$. V_{cap} start falling and when it reaches $3.3V$, the interrupt driver fires an interrupt. The DVFS driver identifies the new performance window by checking the discrete energy level from the signals of the voltage detectors. Thus, it sets the voltage regulator to $2.8V$ and the frequency to $12MHz$. This same behaviour is repeated when V_{cap} drops below $2.8V$ and $2.2V$.

To avoid the flactuation problem we describe in Sec. 3, the DVFS driver delays the performance windows when scaling the frequency up. Let us now focus on Fig. 6, when V_{cap} is at $1.8V$ and rising. The MCU is set at the operating frequency of $1MHz$ and the voltage regulator is at $1.8V$. V_{cap} rises to $2.2V$ and the interrupt driver fires an interrupt. The DVFS driver identifies the new performance window by checking the discrete energy level from the signals of the voltage detectors. However, to avoid flactuations, it waits for the next interrupt to set the new performance window setting. Thus, when V_{cap} rises to $2.8V$, the interrupt driver fires a new interrupt and the DVFS driver sets the voltage regulator to $2.2V$ and the MCU frequency to $8MHz$.

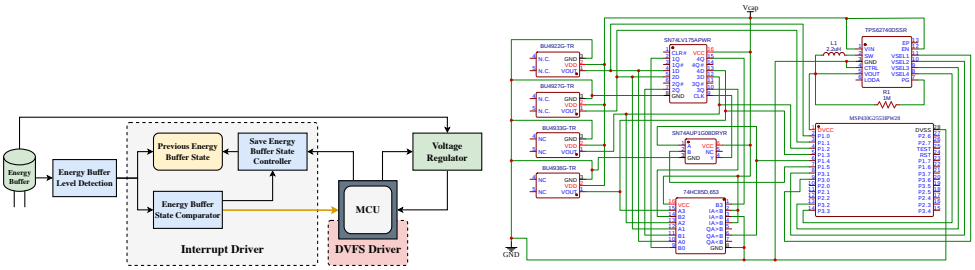
4.2 FBTC

Fig. 7 shows the system design of FBTC, where Fig. 7(a) depicts the core logic of FBTC and Fig. 7(b) shows the corresponding schematics.

We consider the same four performance windows of our D^{2}VFS system design.

FBTC Power State Controller of Fig. 7(a) turns on the system only when the energy buffer voltage is inside the MCU operating range. In the Operating Range Detection step of Fig. 7(a), FBTC identifies if V_{cap} is within such operating range, relying on two BU49XXG [53] voltage detectors, as shown in Fig. 7(b). The first voltage detector triggers when V_{cap} reaches the MCU minimum operating voltage V_{min}, that is $1.8V$, whereas the second voltage detector triggers when V_{cap} reaches a pre-defined power on voltage V_{on}. Note that in our schematics we report a $3.6V$ voltage detector as second voltage detector. However, in our fabricated board, we allow users to select among four different voltage detectors to configure V_{on}, as the *PVComp* and *PVT* ports of Fig. 8 show. To save energy, only the selected voltage detector is powered on and active.

Next, in the System Enable step of Fig. 7(a), FBTC combines the signals of the Operating Range Detection step and decides when



(a) D^{2}VFS logic representation (b) D^{2}VFS circuit schematics

Figure 5: D^{2}VFS system design.

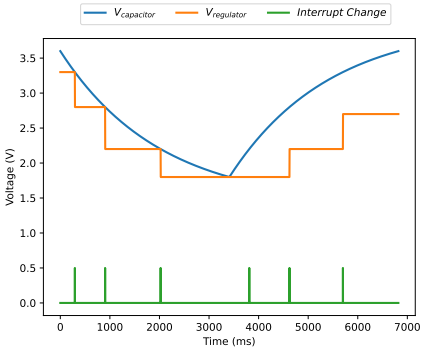


Figure 6: Example of D^{2}VFS behaviour.

the voltage regulator must be powered on. The System Enable step relies on two components, a SN74AUP1G04 [27] NOT gate and on a SN74AUP2G02 [25] 2-input NOR gate used as a flip-flop set-reset, as shown in Fig. 7(b). The NOT gate takes as input the signal of the first voltage detector, that is, the voltage detector that identifies if V_{cap} exceeds V_{min}. Hence, the NOT gate verifies if V_{cap} goes below V_{min} and such signal is used to reset the flip-flop output. Instead, the signal of the second voltage detector sets the flip-flop output.

When V_{cap} exceeds the configured V_{on}, the flip-flop output is set to a logical high and the voltage regulator is powered on. When V_{cap} goes below V_{min}, the flip-flop output is reset to a logical low and the voltage regulator is powered off.

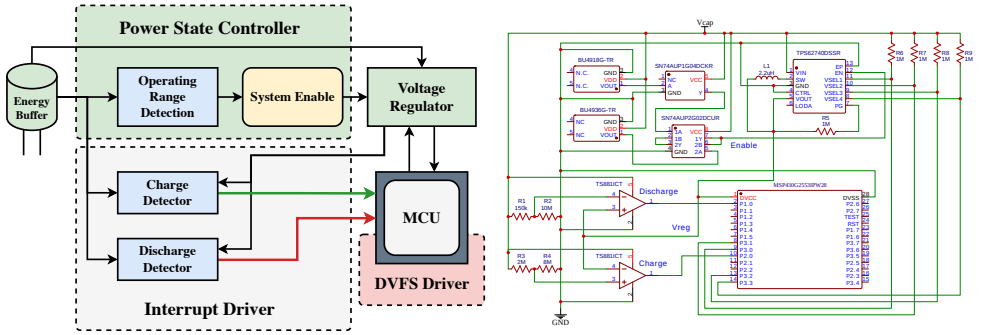
To set the voltage regulator output on system startup, we use four user-configurable pull-up resistors that selects the correct power on voltage. Such resistors are $R6 - R9$ in the schematics of Fig. 7(b) and $R1 - R4$ in our fabricated board shown in Fig. 8. Note that this is required as the voltage regulator output is controlled by the MCU, which cannot set the voltage regulator output before completing its startup phase.

Changepoint detector. FBTC hardware components do not keep track of the current performance window and, instead of explicitly discretizing the energy level, FBTC directly identifies when the performance window changes by comparing the capacitor voltage (V_{cap}) against the voltage regulator output (V_{reg}). The Interrupt Driver of Fig. 7(a) provides such functionality through a Charge Detector and a Discharge Detector. These two hardware components are based on the same logic, which we identify as Changepoint Detector. A Changepoint Detector identifies when the performance window changes and relies on two components: (i) a voltage divider to reduce V_{cap} signal, that is, the $R1 - R2$ ($R3 - R4$) resistors of Fig. 7(b) and (ii) a TS881 [54] operational amplifier that compares the reduced V_{cap} signal against V_{ref}. Depending on the configuration of the operational amplifier inputs, the Changepoint Detector can identify when the energy buffer is charging (Charge Detector) or discharging (Discharge Detector). To detect the energy buffer discharge, V_{reg} is connected to the non-inverting input of the operational amplifier and the V_{cap} voltage divider output at the inverting input, as shown near the *discharge* label of Fig. 7(b). To detect the energy buffer charge, the connections to the operational amplifier inputs are inverted.

The Changepoint Detector compares V_{reg} against V_{cap} and fires an interrupt to the MCU whenever a change is detected. To understand its functionality, let us focus on Fig. 9. The blu curve represents V_{cap}, whereas the orange one represents V_{reg}. The two voltage dividers of the Charge and Discharge Detectors offset the V_{cap} signal in such a way that the considered input voltage equals V_{reg} whenever the conditions to change the performance window are detected.

Let us focus on the red curve, that is, the voltage divider output of the Discharge Detector. In Fig. 9, the MCU frequency is initially set to $16MHz$, V_{reg} to $3.3V$, V_{cap} is $3.6V$, and the energy buffer is discharging. When V_{cap} reaches $3.3V$, the V_{reg} line (orange line) exceeds the scaled V_{cap} curve (red curve). Hence, the Discharge Detector outputs a logical high (brown line), triggering an interrupt to the MCU. We can notice that the scaled V_{cap} curve is used as reference point for V_{reg} to scale the MCU frequency.

The MCU runs a software-level DVFS driver that is triggered whenever the Discharge and Charge Detectors fire an interrupt.



(a) FBTC logic representation (b) FBTC schematics

Figure 7: FBTC system design.

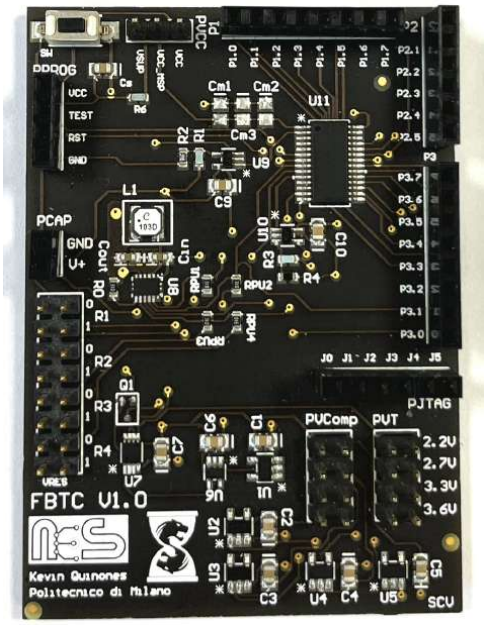


Figure 8: FBTC board.

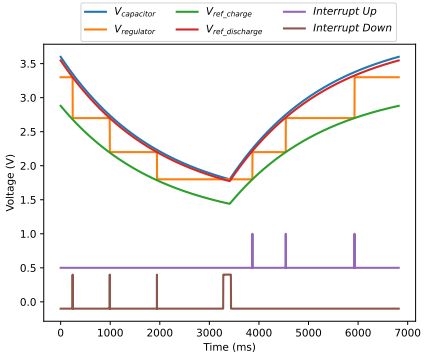


Figure 9: Example of FBTC behaviour.

Depending on the component that is firing an interrupt, the DVFS driver increases or decreases the MCU frequency and voltage regulator output. Similarly to D^{2}VFS, whenever the DVFS driver needs to decrease the MCU frequency, it first decreases the MCU operating frequency and then the voltage regulator output. Instead, the DVFS driver executes these two operations in reverse order whenever it needs to increase the MCU frequency.

Let us focus back on Fig. 9. The Discharge Detector fired an interrupt, and the DVFS driver changes V_{reg} to $2.8V$ and scales the MCU operating frequency to $12MHz$. The scaled V_{cap} curve (red curve) is now far from the V_{reg} line (orange line). The energy buffer continues discharging and when it reaches $2.8V$, the V_{cap} curve exceeds the V_{reg} line. As such, the Discharge Detector fires an interrupt and the DVFS driver changes V_{reg} to $2.2V$ and scales the MCU operating frequency to $8MHz$. This behaviour is repeated when V_{cap} reaches $2.2V$.

Note that, in Fig. 9, when V_{cap} is approaching $1.8V$, V_{reg} (orange line) constantly exceeds the scaled V_{cap} signal (red curve). However, here there is no performance window to change, as the MCU is already at $1MHz$ and V_{reg} at $1.8V$, that is, the settings of the lowest possible performance window. To avoid unexpected behaviours and ignore this situation, the DVFS driver disables the interrupts

from the Discharge Detector when it sets the lowest possible performance window and enables them back whenever scaling to a higher performance window.

The green curve in Fig. 9 represents the voltage divider for V_{cap} used by the Charge Detector. It is crucial that, whenever the performance window changes, the new voltage regulator output V_{reg} does not trigger a change back to the previous performance window. For example, during the discharge of the energy buffer in Fig. 9, the V_{reg} line (orange line) never exceeds the green curve. Note that the graph resolution is not sufficient to verify such situation. If this condition is not met, we have a bouncing problem: the Discharge (Charge) Detector changes V_{reg} and its new value triggers the Charge (Discharge) Detector as the V_{reg} line exceeds the green (red) curve of the V_{cap} voltage divider for the Charge (Discharge) Detector, resulting in FBTC being stuck in constantly changing the performance window back and forth. To avoid this problem, we need to carefully select the values for the resistors used in the voltage dividers for the Charge and Discharge Detectors, as we later describe.

FBTC behaviour during the energy buffer charge is similar to the one we describe for the discharge, where V_{reg} line (orange line) is instead compared against the green curve, that is, the scaled V_{cap} signal of the Charge Detector. However, during the energy buffer charge of Fig. 9, FBTC does not need the same delay of D^{2}VFS to avoid flactuations between two performance windows. This is thanks to the possibility of tuning the voltage at which the Charge Detector identifies the new performance window, which allows the Charge Detector to signal the new performance window to the MCU only when V_{cap} exceeds by ϵ_c the minimum operating voltage of the next performance window. We explain how to tune this parameter later in this section.

Voltage divider configuration. We now describe how to identify the optimal values for the resistors used in the voltage dividers of the Charge and Discharge Detectors. Considering the schematics of Fig. 7(b), the operational amplifiers inputs are:

$$V_{ref\_discharge} = \frac{R2}{R1 + R2} \cdot V_{cap} = \delta_d \cdot V_{cap} \quad (1)$$

$$V_{ref\_charge} = \frac{R4}{R3 + R4} \cdot V_{cap} = \delta_c \cdot V_{cap} \quad (2)$$

Note that we call δ_c (δ_d) as the charge (discharge) voltage regulator resistance ratio.

An interrupt to change the performance window is raised whenever $V_{ref\_discharge} < V_{reg}$ or $V_{ref\_charge} > V_{reg}$. Hence, the bouncing problem happens whenever one condition is verified and the change to V_{reg} verifies the other condition:

$$\text{when } V_{ref\_charge} > V_{reg}[i] \rightarrow V_{ref\_discharge} < V_{reg}[i+1] \quad (3)$$

$$\text{when } V_{ref\_discharge} < V_{reg}[i] \rightarrow V_{ref\_charge} > V_{reg}[i-1] \quad (4)$$

Eq. (3) refers to a scenario where the energy buffer is charging, whereas Eq. (4) refers to a discharging scenario. Note that $V_{reg}[i]$ identifies the voltage regulator output of the performance window i and the performance windows are ordered by ascending operating frequency/voltage. We rewrite Eq. (1) and Eq. (2) by sobstituting Eq. (3) and Eq. (4):

$$\text{when } \delta_c \cdot V_{cap} > V_{reg}[i] \rightarrow \delta_d \cdot V_{cap} < V_{reg}[i+1] \quad (5)$$

$$\text{when } \delta_d \cdot V_{cap} < V_{reg}[i] \rightarrow \delta_c \cdot V_{cap} > V_{reg}[i-1] \quad (6)$$

To avoid the bouncing problem, for each performance window i, Eq. (3) and Eq. (4) must never be satisfied. This means that we need to satisfy the following constraints:

$$\text{when } \delta_c \cdot V_{cap} > V_{reg}[i] \rightarrow \delta_d \cdot V_{cap} \geq V_{reg}[i+1] \quad (7)$$

$$\text{when } \delta_d \cdot V_{cap} < V_{reg}[i] \rightarrow \delta_c \cdot V_{cap} \leq V_{reg}[i-1] \quad (8)$$

Eq. (7) refers to a scenario where the energy buffer is charging and gives us a set of constraints on the discharge voltage regulator resistance ratio δ_d, whereas Eq. (8) refers to a discharging scenario and gives us a set of constraints on the charge voltage regulator resistance ratio δ_c.

To identify the list of constraints, for each performance window i with an associated voltage range (V_{max}, V_{min}):

- we consider Eq. (7) with $V_{cap} = V_{max} + \epsilon_d$ if there exists a performance window $i + 1$
- we consider Eq. (8) with $V_{cap} = V_{min} + \epsilon_c$ if there exists a performance window $i - 1$

Note that ϵ_d (ϵ_c) is the minimum voltage sensitivity we want to obtain for the discharge (charge) detector. We consider these parameters to compensate for irregularities of real circuit components and to avoid flactuations when the energy buffer is charging. Note that we use ϵ_d (ϵ_c) with the constraints for δ_c (δ_d), as they refer to a condition for the charge (discharge) detection.

Let us now consider the four performance windows for the MSP430-G2553 [26]:

1) $1MHz$ with $V_{reg} = 1.8V$ and V_{cap} in $(2.2V, 1.8V)$
2) $8MHz$ with $V_{reg} = 2.2V$ and V_{cap} in $(2.8V, 2.2V)$
3) $12MHz$ with $V_{reg} = 2.8V$ and V_{cap} in $(3.3V, 2.8V)$
4) $16MHz$ with $V_{reg} = 3.3V$ and V_{cap} in $(3.6V, 3.3V)$

We consider $\epsilon_c = \epsilon_d = 50mV$. We explain later the selection of these values.

We consider the performance windows 1,2, and 3 with Eq. (7), obtaining the following constraints on δ_d:

- $V_{cap} = 2.20V + 50mV = 2.25V$, $V_{reg}[1] = 1.8V$, $V_{reg}[2] = 2.2V \rightarrow \delta_d \geq \frac{2.2V}{2.25V}$
- $V_{cap} = 2.80V + 50mV = 2.85V$, $V_{reg}[2] = 2.2V$, $V_{reg}[3] = 2.8V \rightarrow \delta_d \geq \frac{2.8V}{2.85V}$
- $V_{cap} = 3.30V + 50mV = 3.35V$, $V_{reg}[3] = 2.8V$, $V_{reg}[4] = 3.3V \rightarrow \delta_d \geq \frac{3.3V}{3.35V}$

The constraints obtained from Eq. (7) provide a lower bound for δ_d. Hence, we need to consider the constraint with the highest value for δ_d, that is, $\delta_d \geq \frac{3.3V}{3.35V} = 0.9851$. Note that we want to select the lowest possible value for δ_d, that is, $\delta_d = 0.9851$, as otherwise we would increase the minimum voltage sensitivity ϵ_c. Recalling that $\delta_d = \frac{R2}{R1+R2}$, we can identify that $R1 = 150k\Omega$ and $R2 = 10M\Omega$.

We consider the performance windows 2,3, and 4 with Eq. (8), obtaining the following constraints on δ_c:

- $V_{cap} = 2.20V + 50mV = 2.25V$, $V_{reg}[1] = 1.8V$, $V_{reg}[0] = 1.8V \rightarrow \delta_c \leq \frac{1.8V}{2.25V}$
- $V_{cap} = 2.80V + 50mV = 2.85V$, $V_{reg}[2] = 2.8V$, $V_{reg}[1] = 2.2V \rightarrow \delta_c \leq \frac{2.2V}{2.85V}$
- $V_{cap} = 3.30V + 50mV = 3.35V$, $V_{reg}[3] = 3.3V$, $V_{reg}[2] = 2.8V \rightarrow \delta_c \leq \frac{2.8V}{3.35V}$

The constraints obtained from Eq. (8) provide a upper bound for δ_c. Hence, we need to consider the constraint with the lowest value for δ_c, that is, $\delta_c \geq \frac{1.8V}{2.25V} = 0.8$. Note that we want to select the highest possible value for δ_c, that is, $\delta_c = 0.8$, as otherwise we would decrease the minimum voltage sensitivity ϵ_d. Recalling that $\delta_c = \frac{R4}{R3+R4}$, we can identify that $R3 = 2M\Omega$ and $R4 = 8M\Omega$.

Selecting ϵ_c. When selecting ϵ_c, we need to consider a value that allows the energy buffer to keep sufficient energy to sustain the computation in the new performance window for a reasonable amount of instructions. An extra voltage of ϵ_c in a capacitor corresponds to $\frac{1}{2}C{\epsilon_c}^2$ energy. By considering a maximum energy consumption per clock cycle of e_{cc}, the number of extra clock cycles n_{clock_cycles} that ϵ_c allows to execute is:

$$n_{clock\_cycles} = \frac{\frac{1}{2}C{\epsilon_c}^2}{e_{cc}} \tag{9}$$

In the worse case, the DVFS driver of FBTC requires 18 machine-code instructions to change the performance window, that is, 18 clock cycles. Hence, to justify switching to a higher frequency, we need to satisfy the following equation:

$$n_{clock\_cycles} * p_{lower} \geq 18 + n_{clock\_cycles} \tag{10}$$

where p_{lower} represents the energy consumption increase of a lower operating frequency with respect to a higher one that can be sustained at the same voltage level. For the MSP430-G2553 [26], the average p_{lower} among the three switching points (i.e., $1MHz - 8MHz$, $8MHz - 12MHz$, and $12MHz - 16MHz$) is 1.17. Note that this means that, switching to a higher frequency provides, on average, a 17% better energy efficiency. Hence, for the MSP430-G2553 [26], $n_{instr} \geq 106$ clock cycles.

FBTC sets the MCU to operate at the minimum possible voltage for each operating frequency. Hence, to identify the highest energy consumption per clock cycle of the MCU, we need to consider the operating frequency with the highest energy consumption at its minimum operating voltage. Among the four operating frequencies of the MSP430-G2553 [26], $16MHz$ is the one with such highest energy consumption and it consumes $0.85nJ$ per clock cycle, as shown in Fig. 1.

By sobstituting these values in Eq. (9) and by considering a target capacitor of $100\mu F$, ϵ_c must be at least $0.042V$.

5 EVALUATION

We evaluate the performance of D^{2}VFS and FBTC under different system settings and energy harvesting scenarios. We describe next the experiments and system setup, the considered energy scenarios, and the experiments results.

5.1 Setting → System and Experiments

Reproducing energy harvesting sources using real hardware is extremely challenging, as their behavior is highly non-deterministic [17, 20]. For this reason, we opt for software-based system emulation to execute our experiments, as this not only ensures experiments reproducibility, but also allows us to better control the system setting.

System emulation. We run our experiments using ScEpTIC [45, 46], a versatile and easily-extendable emulator for intermittent programs. To execute our experiments, we extend ScEpTIC to emulate (i) the energy consumption of common intermittent system components, (ii) the energy harvested from the environment, and (iii) the logic and energy consumption of custom circuitry.

Our extension of ScEpTIC allows us to configure and design the emulated system components, including the energy harvesting source, the MCU, and components external to the MCU, such as the energy buffer, peripherals, non-volatile memories, voltage regulators, and custom circuits. Note that we emulate energy harvesting sources by reproducing a voltage trace [3, 20, 51], which can be either synthetic or gathered from a real harvester. Further, we rely on our extension to implement the system and circuitry components that emulate the energy consumption and logic of D^{2}VFS and FBTC boards.

To run our experiments, we implement a ScEpTIC simulation that emulates intermittent executions of the configured system. During the emulation of the program execution, we make ScEpTIC track the level of the energy buffer by emulating the energy consumption of the configured system and the energy harvested from the configured energy source. Then, whenever the level of the energy buffer falls below a user-specified threshold, such as the minimum MCU operating voltage, ScEpTIC emulates the occurrence of a power failure.

The code and documentation of our ScEpTIC extension are available as open-source release [42].

Platform and system setting. We consider as target platform the MSP430-G2553 [26], a MCU from the MSP430 family of extremely-low-power MCUs from Texas Instruments [32], as this is the same platform used to evaluate D^{2}VFS [3]. We implement into ScEpTIC an energy model of the MCU which accounts for its different operating modes. Note that we use existing measures [4] to model the MCU energy consumption in active mode, as in such operating mode the MSP430-G2553 experiences flactuations in its power consumption that are not represented in its datasheet [4]. We instead rely on datasheet information [26] to model the MCU energy consumption in low-power mode, and the energy consumption and latency of peripheral accesses, such as ADC probing. Further, we model FBTC energy consumption using both its datasheet information and real measures taken from the fabricated board. We compare the two models in Sec. 5.3.

To store the state across power failures, we equip the system with a MB85RC64V [36] non-volatile memory, that is, a $8Kbyte$ FRAM accessible thorugh the I^2C protocol at a maximum speed of $1MHz$.

We evaluate the performance of D^{2}VFS and FBTC considering two different well-established system supports for intermittent computing: Hibernus [8] and Mementos [51]. Both mechanisms save the program state, consisting in register file, special registers, and main memory, onto a non-volatile memory location whenever the voltage of the energy buffer V_{buff} falls below a specified threshold V_{save}. For doing so, Hibernus relies on system interrupts that fire whenever the V_{save} is reached; Mementos instead relies on special function calls, statically placed at specific program locations, that probe V_{buff} through an ADC and then decide whether to save the state.

To reproduce the behaviour of these two forward progress mechanisms, we extend ScEpTIC to automatically identify the optimal

V_{save}. Then, for Hibernus, we consider an external voltage divider of $200K\Omega$ [8] and we model into ScEPTIC the execution of state-saving operations whenever V_{buff} falls below the optimal V_{save}. Instead, for Mementos, we use its *loop-latch* placement strategy [51] to statically place probe function calls in the benchmarks source code. Then, we model into ScEPTIC the probe function call logic, and the timing and energy consumption of the ADC using the data found in the MSP430-G2553 datasheet [26].

Note that, to reproduce the setup of a real-world deployment, we assume that, similarly to Mementos [51], Hibernus state-saving operations save only the used portion of main memory (i.e., the one delimited by the stack pointer) instead of its whole content [8], as this removes an unnecessary overhead of state-saving operations.

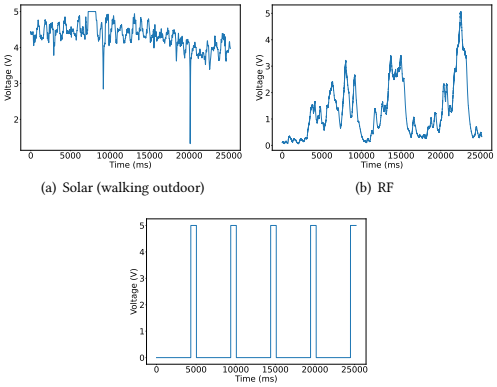
Finally, we must note that at a clock frequency of $1MHz$, the MSP430-G2553 minimum operating voltage is $1.8V$, whereas the ADC minimum operating voltage is $2.2V$. As a consequence, whenever the voltage of the energy buffer falls below $2.2V$, the ADC may return a value that does not reflect the current level of the energy buffer, causing an unexpected behaviour of Mementos, which may execute state-saving operations when not necessary or not execute them when necessary. To account for this situation, we consider three possible scenarios for Mementos: (i) `Default`, where function calls do not probe the ADC and instead directly save the state whenever $V_{buff} < 2.2V$, (ii) `NOADCOFF`, where we assume the ADC can operate in the same voltage range of the MCU, and (iii) `ADCMINV`, where we set the MCU to power off at $2.2V$.

Benchmarks and baselines. Battery-less devices usually act as sensors of a Wireless Sensor Network and their common workload can be summarized into three phases [2]: (i) environment sensing, (ii) data processing and computation, and (iii) data communication. The computation executed during environment sensing and data communication (phase i and iii) is usually off-loaded to peripherals external to the MCU, whereas data processing and computation (phase ii) usually executes on the MCU. D^{2}VFS and FBTC techniques change the operating frequency of the MCU, but not the one of peripherals, thus affecting only the performance of the computation executed on the MCU itself. For these reasons, we focus only on benchmarks that represent data processing and computation (phase ii), as this phase usually executes entirely on the MCU.

We thus select a set of benchmarks that summarize different types of data processing computations commonly used in intermittent computing [46, 51, 57]: (i) Dijkstra algorithm for computing the shortest path between two nodes of a graph, (ii) Fast Fourier Transform (FFT) for signal analysis, and (iii) RSA for data encryption. We consider the open-source implementation of each benchmark available in the MiBench2 [19, 22] benchmark suite and we compile them using Clang [37] version 8.0.1 with the default compiler settings.

We consider as baseline a group of static frequency configurations for the MSP430-G2553 [26]: $1MHz$, $8MHz$, $12MHz$, and $16MHz$.

Metrics. To evaluate the performance of D^{2}VFS and FBTC, we mainly focus on the *time required to complete a given workload* and on the *device energy consumption*, as these are the metrics affected by the voltage and frequency changes of D^{2}VFS and FBTC.



(a) Solar (walking outdoor) (b) RF

(c) Synthetic

Figure 10: Energy sources voltage trace

When measuring the time required to complete a given workload, we differentiate between the time of active periods, that is, the *execution time*, and the time of inactive periods, that is, the *re-charge time*. This allows us to identify (i) where performance is lost, (ii) how different configurations of voltage and frequency affect the execution time, and (iii) how the external circuitry of D^{2}VFS and FBTC affect the recharge time. Moreover, this allows us also to identify how different voltage operational ranges affect performance, as different frequencies have different voltage ranges that affect both the execution and re-charge time.

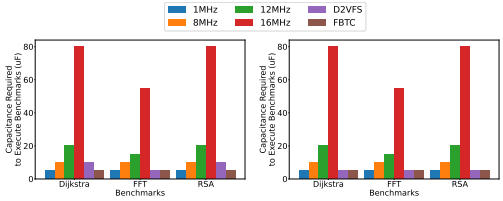
Moreover, to compare different energy consumption levels, we also measure the *number of power failures* happening during the execution of a workload. We consider this metric as an indicator showing how energy consumption affect performance. In fact, a higher energy consumption implies a significantly worse performance level when it causes additional power failures, as a power failure further increases both the execution and recharge time due to additional restore operations and energy buffer recharge.

Finally, to identify the energy overhead intoduced by D^{2}VFS and FBTC, we measure the energy consumption of their circuitry.

5.2 Setting → Energy

To evaluate D^{2}VFS and FBTC under different energy conditions, we must account for the knobs that influence the energy and power capabilities of an intermittent system, such as the considered energy harvesting source, the energy buffer size, and the power-on voltage. We discuss next how we select these parameters.

Energy harvesting sources. The energy supply capability of an energy harvesting source defines the timespan of the active and inactive periods of a power cycle, influencing the number of power cycles required to complete a given workload. In our experiments, we consider a group of energy sources that represents three possible energy supply scenarios.



(a) Hibernus (b) Mementos

Figure 11: Minimum capacitance required to execute benchmarks at a given frequency.

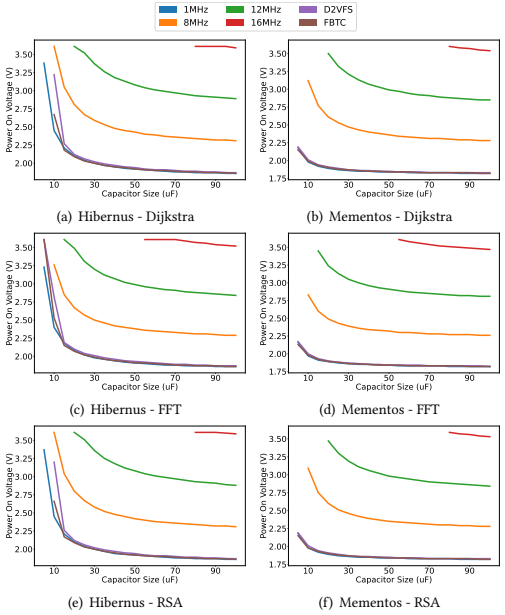
First, we consider a *high-capability* energy supply scenario, characterized by long active periods and a low power failure rate. A high-capability energy source supplies sufficient energy to sustain the device workload during its active period, significantly extending the active period timespan and consequently reducing or completely removing the number of power failures happening during the execution of a given workload. In our experiments, we reproduce this scenario by considering the voltage trace of a solar energy source, measured from a solar panel outside our lab while walking [3]. Fig. 10(a) depicts its voltage trace.

Next, we consider a *medium-capability* energy supply scenario, characterized by medium active periods and a medium power failure rate. A medium-capability energy source sporadically supplies energy during the device active period, increasing the length of the active period. However, the supplied energy is usually not sufficient to sustain the device workload, causing a medium number of power failures. In our experiments, we reproduce this scenario by considering the voltage trace of a RF energy source, taken from the voltage traces used for the evaluation of Mementos [1, 51]. Fig. 10(b) depicts its voltage trace.

Finally, we consider a *low-capability* energy supply scenario, characterized by short active periods and a high power failure rate. A low-capability energy source rarely supplies energy during the device active period, causing short active periods and very frequent power failures. As such, the supplied energy usually recharges the device energy buffer only during its inactive period. In our experiments, similarly to previous works [33], we reproduce this scenario by implementing a synthetic $5V$ energy source that supplies energy only when the device is powered off, that is, during its inactive period. Fig. 10(c) depicts a possible example of its voltage trace. Note that the source supplies $0V$ when the MCU is not in active mode of operation (e.g., low-power mode, powered off).

Note that, similarly to previous works [3, 51], we consider a $30K\Omega$ equivalent resistance for the energy harvesting circuit of each energy source.

Energy buffer size and V_{on}. We consider a capacitor as energy buffer. The capacitor size and the power-on voltage influence the length of power cycles and the time required to resume the computation after a power failure. High-capacitance capacitors increase the duration of a power cycle, as they store more energy. However, they also increase the time required to recharge the energy buffer. This not only affects the duration of a power cycle, as the harvested



(a) Hibernus - Dijkstra (b) Mementos - Dijkstra
(c) Hibernus - FFT (d) Mementos - FFT
(e) Hibernus - RSA (f) Mementos - RSA

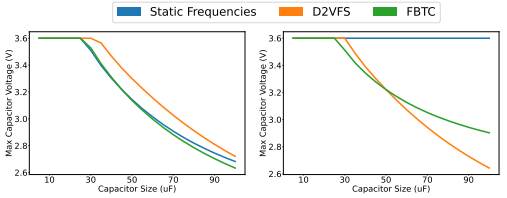
Figure 12: Minimum V_{on} required for benchmark execution.

energy produces a lower increase in the capacitor's voltage, but also increases the recharge time, that is, the time required to reach V_{on} while the device is powered off. Similarly, a high V_{on} extends the timespan of a power cycle active period, as it results in a higher energy availability during the computation, but it also increases the recharge time. Note that the identification of the optimal capacitance C and V_{on} is not the scope of this paper.

However, there are lower bounds for C and V_{on} that cannot be exceeded, which are frequency and workload dependent. C and V_{on} delimit the energy available in the active period of a power cycle, that is, e_{active}, which must be higher than the energy consumed by state-saving and state-restoring operations. Otherwise a device would not achieve forward progress across power failures, as e_{active} would not be sufficient to progress in the program execution or complete state-saving operations.

Hence, to evaluate the behaviour of D^{2}VFS and FBTC under different conditions, we consider multiple combinations of lower bounds for C and V_{on}. We extend ScEPTIC to identify the lower bounds of C and V_{on} for a given workload, which we use to select a set of values for C and V_{on} that reflect a reasonable choice in a real-world deployment.

Fig. 11 shows the lower bound of C for the baselines, D^{2}VFS, and FBTC across all the benchmarks and forward progress mechanisms. In general, the execution of all the benchmarks at $16MHz$ and $12MHz$ require at least a $80\mu F$ and $20\mu F$ capacitor, respectively. Instead, $1MHz$, $8MHz$, D^{2}VFS, and FBTC require no more than a



(a) Hibernus (b) Mementos

Figure 13: Maximum capacitor voltage using the RF energy source.

$10\mu F$ capacitor, that is, the minimum decoupling capacitance of the MSP430-G2553 suggested by Texas Instruments is $10\mu F$ [31]. Note that Fig. 11(a) depicts that D^{2}VFS requires a $10\mu F$ capacitor to execute the Dijkstra and RSA benchmarks with Hibernus, whereas $1MHz$ and FBTC require only a $5\mu F$ capacitor. This is a consequence of D^{2}VFS higher quiescent current consumption compared to FBTC.

Following these results, we choose to execute our experiments using two capacitor sizes: (i) $80\mu F$ to run experiments using all baseline operating frequencies, and (ii) $20\mu F$ to run experiments using all baselines operating frequencies except $16MHz$.

Then, we identify the V_{on} lower bound for each possible capacitor size. Fig. 12 shows the minimum values of V_{on} across each benchmark and capacitor size. In general, the minimum V_{on} trend follows the voltage operating range of the various frequencies: $16MHz$ has the highest V_{on}, whereas $1MHz$ has the lowest. Note that D^{2}VFS and FBTC have a V_{on} within the same one of $1MHz$, as they have the same voltage operating ranges.

To ensure that all the baselines selected for a given capacitor size can achieve forward progress, we choose the highest V_{on} across the ones of the baselines. Otherwise, a baseline may not have sufficient energy to save the state and it may be forever stuck in re-executing the same portion of a workload [13]. We thus select $3.6V$ for all the capacitor sizes.

Note that, despite some configurations have smaller lower bounds for C or V_{on}, we consider the same values for C and V_{on} across the baselines, D^{2}VFS, and FBTC, as this give us comparable results within the same experiment. Moreover, in our experiments we assume that the capacitor voltage cannot exceed $3.6V$ and each experiment starts with the capacitor voltage equals to V_{on}.

Quiescent currents and energy harvesting. The capacitor leakage current I_{leak} and the quiescent current consumption $I_{quiescent}$ of external circuitry, such as the one of Hibernus [8], D^{2}VFS, and FBTC, cause a capacitor discharge even when the MCU is powered off. Depending on these current consumptions, the capacitor size C, and the power-on voltage V_{on}, the energy harvesting source may not supply sufficient energy to recharge the energy buffer to V_{on}, potentially leading to a scenario where the MCU never powers on. Recalling that a capacitor is recharged whenever the energy harvesting source supplies a voltage higher than the capacitor voltage, this problem may happen with energy sources that supply short bursts of energy with a voltage higher than V_{on}. With a high value of C or V_{on}, such energy sources are unable to recharge the capacitor to V_{on} in a single burst, as these parameters highly affect the capacitor voltage increase and consequently its recharge time. As such, after an energy burst partially recharges the capacitor, the MCU is still powered off, and I_{leak} and $I_{quiescent}$ consume the previously harvested energy stored in the capacitor, potentially depleting it. If subsequent energy bursts do not supply sufficient energy to compensate for the energy consumed by I_{leak} and $I_{quiescent}$, or do not recharge the capacitor to V_{on}, the MCU never powers on. Note that that is the case for the RF source of Fig. 10(b) with $C = 100\mu F$ and $V_{on} = 3.6V$.

To avoid this problem, one may lower C or V_{on}, allowing the energy harvesting source to recharge the capacitor to V_{on} within a single energy burst. However, this is not possible when C or V_{on} are already set to their lower bound to execute a given workload, that is, our experiments configuration with all the baselines included.

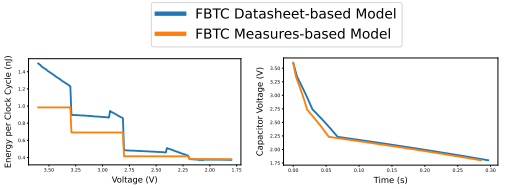
To overcome the problem in such situations, we instead consider the addition of a voltage doubler between the energy buffer and the energy harverster, as in the WISP platform [48], consisting in a circuit that doubles the voltage reaching the capacitor. By doubling the voltage of an energy source, energy bursts whose voltage exceeds V_{on} are both more frequent and longer, allowing the capacitor to reach V_{on}, despite the energy consumed by I_{leak} and $I_{quiescent}$. Note that we implement the voltage doubler logic in our extension of ScEpTIC.

To investigate where a voltage doubler may be necessary, we implement in our extension of ScEpTIC an analysis that identifies the maximum voltage reached by the capacitor during the inactive period of a power cycle for a given energy harvesting source. Note that such analysis accounts only for the quiescent current consumption $I_{quiescent}$ of D^{2}VFS, FBTC, and Hibernus circuitry, whereas we ignore the capacitor leakage voltage, as this is significantly lower than $I_{quiescent}$.

From our experiments, the *high-capability* and *low-capability* energy sources do not require any voltage doubler for any configuration. Instead, the *medium-capability* energy source requires one voltage doubler for the configuration with $C = 80\mu F$ and $V_{on} = 3.6V$. In fact, as Fig. 13 depics, starting from a capacitance of $30\mu F$, the maximum voltage reached by the capacitor drops below $3.6V$, that is, the V_{on} we selected for every experiment configuration.

However, note that using a voltage doubler may not always be an option, as (i) voltage doublers usually require AC input current [14], whereas an energy harverster may output DC current [9], and (ii) similarly to voltage regulators, voltage doublers never have a 100% efficiency [14], wasting a portion of harvested energy. These are the reasons behind our choice of executing the experiments with two capacitor configurations, as one requires a voltage doubler, that is, $80\mu F$, and the other does not, that is, $20\mu F$.

Finally, D^{2}VFS has a higher quiescent current draw than FBTC and static frequency configurations, which reults in a lower equivalent resistance. The quiescent current consumption of Hibernus circuitry is dominant with respect to the one of D^{2}VFS and FBTC. Hence, when using Hibernus, D^{2}VFS lower equivalent resistance results in a faster recharge of the capacitor, which reaches higher voltages compared to FBTC and static frequencies, as Fig. 13(a) shows. Instead, in Mementos, starting from a capacitance of $50\mu F$, the higher quiescent current of D^{2}VFS overcomes the advantage of a lower equivalent resistance, nullyfing such benefit, as Fig. 13(b)



(a) Energy per cycle (b) Capacitor discharge

Figure 14: Comparison of FBTC datasheet-based model against FBTC measures-based model.

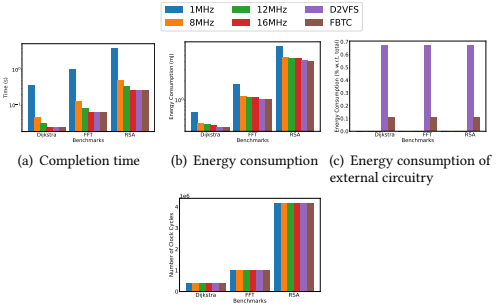
shows. Note that, for Mementos, we estimate that in the static frequencies configuration the capacitor always reaches $3.6V$, as we assume no $I_{quiescent}$.

5.3 Energy Model Validation

We model D^{2}VFS and FBTC energy consumption using real measures of a MSP430-G2553 [26] MCU and the datasheet information for the various circuitry components of DVFS and FBTC boards. To validate our model, we measure the energy consumption of the FBTC board we fabricated. We use a PeakTech 6225A [50] variable power supply to vary the voltage of the FBTC board between $3.6V$ and the minimum operating voltage for the considered clock frequency, using steps of $0.01V$. We measure the FBTC board current draw using a UNI-T UT61E multimiter [56]. We repeat the measures for each one of the operating frequencies we consider, namely, $16MHz$, $12MHz$, $8MHz$, and $1MHz$, and we use them to create a measures-based model for the FBTC board.

Fig. 14 compares FBTC datasheet-based model against the fabricated FBTC board. Fig. 14(a) compares the energy consumption per clock cycle of the datasheet-based FBTC model against our measures. Our model assumes an average efficiency of 90% for the TPS62740 [28] voltage regulator, as indicated in the datasheet [28]. However, this does not represent the actual behaviour of the voltage regulator. The measures of Fig. 14(a) show that the voltage regulator has a non-linear behaviour and its efficiency depends on the input/output voltages. In particular, in the range between $3.6V$ and $3.3V$, that is, the operating voltage range of $16MHz$, our model underestimates the energy consumption by up to 50% and, on average, by 38%. This discrepancy decreases down to 34% (23%) in the voltage range associated to $12MHz$ ($8MHz$), that is, between $3.3V$ ($2.8V$) and $2.8V$ ($2.2V$), with an average underestimation of 28% (13%). Note that in the range between $2.2V$ and $1.8V$, that is, the voltage range associated to $1MHz$, our model overestimates the energy consumption by up to 2%.

To evaluate the effects of the differences between the datasheet-based model and the real measures of the FBTC board, we create a measures-based model and we compare the workload achieved in a single discharge of a $100\mu F$ capacitor. The lower energy consumption of the datasheet-based model results in the execution of 16% more clock cycles than the measures-based model, as the datasheet-based model executes 907365 (763751) clock cycles. The capacitor discharge time depicted in Fig. 14(b) shows an interesting behaviour. The significant difference in the energy estimation between $3.6V$ and $3.3V$ barely affects the discharge time. The the overall difference between the discharge times is 4%, which is mainly caused by the differences in the energy estimation between $3.3V$ and $2.2V$. This is due to the non-linear relation between the capacitor voltage and the capacitor energy, which makes the MCU sustain lower frequencies for longer periods. Consequently, the discrepancy in the energy estimation of higher frequencies has a low impact on the overall simulation.



(a) Completion time (b) Energy consumption (c) Energy consumption of external circuitry

(d) Simulated clock cycles

Figure 15: Benchmarks results with the high-capability energy source, Hibernus, C = 80μF, and V_{on} = 3.6V.

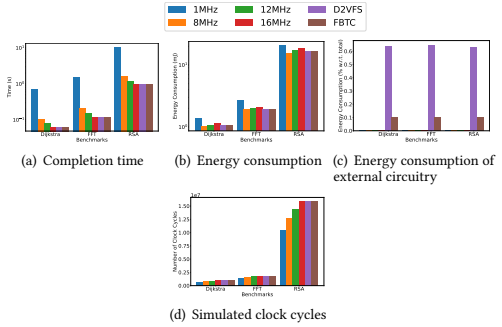
For these reasons, despite the energy estimation difference between the two FBTC models, there is no difference in the performance trend of the FBTC models against static frequencies and D^{2}VFS across our experiments, which we run considering both the datasheet-based and measures-based FBTC models. The only significant difference stands in the RSA benchmark with the low-capability energy source, where the measures-based FBTC model experiences an extra power failure than the datasheet-based model.

For this reason, we report in this paper the experiments results for the datasheet-based FBTC model, as the lack of real measures for the D^{2}VFS board would make a comparison against the FBTC measures-based model not fair. Further, the datasheet-based model of FBTC allows us to identify the energy consumption of the FBTC components external to the MCU, whereas the measures-based FBTC model does not allow such distinction.

5.4 Results → High-capability Energy Source

Experiments with the high-capability energy source experience no power failures, as such energy source supplies sufficient energy to sustain the workload within a power cycle for any experiment configuration. For this reason, we do not report the number of power failures and the recharge time, as they are always equal to zero. Similarly, we do not report the execution time, as it corresponds to the completion time.

Further, in our experiments, the energy source always keeps the capacitor at its maximum voltage, independently of its size. Consequently, we discuss here only the experiment results with a $80\mu F$ capacitor, as experiments with a $20\mu F$ capacitor do not produce different results.



(a) Completion time (b) Energy consumption (c) Energy consumption of external circuitry

(d) Simulated clock cycles

Figure 16: Benchmarks results with a high-capability energy source, Mementos, C = 80μF, and V_{on} = 3.6V.

Hibernus. Fig. 15 shows the experiment results with Hibernus and a 80μF capacitor, where D^{2}VFS and FBTC demonstrate the best performance. Fig. 15(a) depicts the completion time of each benchmark. D^{2}VFS and FBTC require the same time of the static 16MHz configuration to complete the benchmarks and are up to 16x faster than the other baselines. This behaviour is a consequence of having always a full energy buffer: the harvested energy keeps the energy buffer in the operating range of 16MHz, which not only is the fastest and most efficient operating frequency, but also is the frequency that D^{2}VFS and FBTC set.

In fact, the combination of voltage and frequency scaling technique of D^{2}VFS and FBTC keeps the MCU in the most efficient configuration, resulting in up to 1.7x lower energy consumption, as Fig. 15(b) shows. Moreover, despite executing benchmarks at 16MHz, D^{2}VFS and FBTC have a 9% lower energy consumption than the static 16MHz configuration. This is a consequence of the voltage scaling technique of D^{2}VFS and FBTC, which, despite (i) the energy lost due to the voltage regulator and (ii) the energy consumed by their additional components, supplies the lowest possible voltage to the MCU, keeping the energy consumption lower than the static 16MHz configuration.

Finally, D^{2}VFS and FBTC additional components have a very low impact on the overall energy consumption. Fig. 15(c) shows that, across all benchmarks, D^{2}VFS and FBTC additional components are responsible only for the 0.67% and 0.1% of the overall energy consumption, respectively. As such, FBTC has a 0.57% lower energy consumption than D^{2}VFS, while demonstrating the same completion time.

Mementos. As we previously argue, the high-capability energy source supplies sufficient energy to ensure that the capacitor voltage never drops below the minimum ADC operating voltage. This causes the three configuration scenarios for Mementos to produce the same results, as the voltage is always in the ADC operating voltage range. As such, we report only the results of the `Default` Mementos configuration.

Fig. 16 shows the experiment results with Mementos in `Default` configuration and a 80μF capacitor. Fig. 16(a) depicts that the completion time shows the same pattern of the experiments with Hibernus: D^{2}VFS and FBTC require the same time of the static 16MHz configuration to complete the benchmarks and they are up to 12x faster than the other baselines.

However, as Fig. 16(b) shows, D^{2}VFS and FBTC no longer have the same advantage of energy consumption that have in the experiments with Hibernus. This is a consequence of the executions of Mementos' probe function, which turns the ADC on, waits for a sample of the capacitor voltage, and turns the ADC back off. These operations have an access latency and introduce a penalty in terms of wait cycles, consisting in clock cycles where the MCU wait for the operation completion by executing null operations (NOPs).

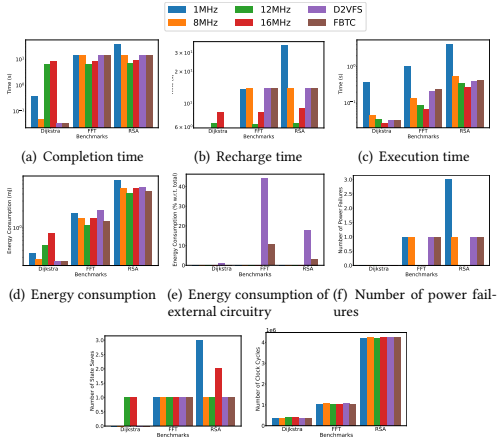
The number of NOPs is proportional to the MCU operating frequency. In the MSP430G2553, the ADC access latency is 1.32μs, consisting in a power-on latency of 0.1μs [26] and a sampling time of 1.22μs [26]. At 16MHz (8MHz), the MCU needs to wait 22 (11) clock cycles to retrieve an ADC sample. As a consequence, higher frequencies are subject to a higher penalty, as they execute a higher number of wait cycles. As Fig. 16(d) shows, the MCU at 16MHz executes, on average, 40% (20%) more clock cycles than at 1MHz (8MHz), with a maximum of 52% (25%) more clock cycles in RSA, where probe function calls happen more frequently. The higher is the operating frequency, the higher is the number of wait cycles, and the higher is the energy that wait cycles consume, potentially wasting all the energy saved by operating at a more efficient clock frequency.

Despite the penalty of ADC accesses, D^{2}VFS and FBTC still consume less energy than the static 1MHz, 12MHz, and 16MHz configurations across all benchmarks, as Fig. 16(b) shows. This is thanks to D^{2}VFS and FBTC voltage scaling technique, which lower the voltage to the mimimum possible to operate at 16MHz. Note that the significantly lower access penalty that 1MHz has, that is, 2 clock cycles instead of 22, is still not sufficient to overcame the better energy per cycle of 16MHz. Instead, FBTC (D^{2}VFS) consumes, on average, 3.7% (4.29%) more energy than the static 8MHz configuration, with a maximum of 7.6% (8.22%) more in RSA. However, FBTC and D^{2}VFS are, on average, 67% faster than 8MHz, with a minimum of a 61% faster completion time in RSA. We believe that this significant decrease in completion time completely justifies the small increase in energy consumption, especially considering that the energy source supplies more energy than the device can buffer and thus an increase of energy consumption does not cause any power failure.

Finally, D^{2}VFS and FBTC additional components have a very low impact on the overall energy consumption, showing the same ratio we see in the experiments with Hibernus, that is, 0.64% and 0.1% of the total energy consumption, respectively, as Fig. 16(c) shows. Similarly, FBTC has a 0.55% lower energy consumption than D^{2}VFS, while demonstrating the same completion time.

5.5 Results → Medium-capability Enegy Source

We discuss the results for the experiments with the medium-capability energy source. Note that V_{on} is set to 3.6V in each experiment.

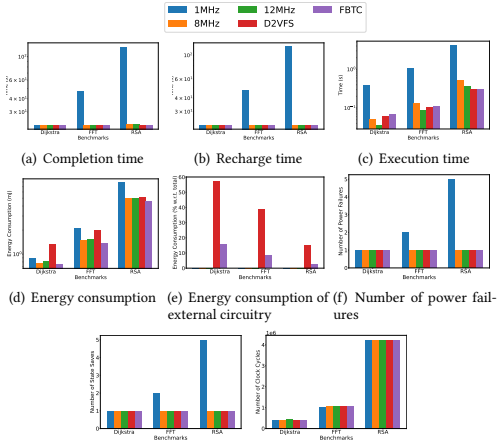


(a) Completion time (b) Recharge time (c) Execution time

(d) Energy consumption (e) Energy consumption of external circuitry (f) Number of power failures

(g) Number of state saves (h) Simulated clock cycles

Figure 17: Benchmarks results with a medium-capability energy source, Hibernus, C = 80μF, and V_{on} = 3.6V.



(a) Completion time (b) Recharge time (c) Execution time

(d) Energy consumption (e) Energy consumption of external circuitry (f) Number of power failures

(g) Number of state saves (h) Simulated clock cycles

Figure 18: Benchmarks results with a medium-capability energy source, Hibernus, C = 20μF, and V_{on} = 3.6V.

Hibernus with C = 80μF. Fig. 17 shows the experiment results with Hibernus and a 80μF capacitor. From the completion times shown in Fig. 17(a) we can identify two different performance trend for D^{2}VFS and FBTC: the first in Dijkstra, where they outperform all baselines, and the second in FFT and RSA, where they demonstrate an average performance against the baselines.

In Dijkstra, D^{2}VFS and FBTC show a better performance than the best performing baseline, that is, the static 8MHz configuration: they demonstrate a 42% and 41% faster completion time (Fig. 17(a)), respectively, and consume 8% and 11% less energy (Fig. 17(d)), respectively. Moreover, D^{2}VFS and FBTC are up to two orders of magnitude faster than the baselines and consume up to 3x less energy than the static frequency configurations.

D^{2}VFS and FBTC voltage and frequency scaling technique is the reason behind their better performance in Dijkstra. Fig. 17(c) shows that scaling the frequency grants D^{2}VFS and FBTC a faster execution time than the static 8MHz configuration, as they are able to execute a portion of Dijkstra at faster operating frequencies.

Further, the operating frequency changes of D^{2}VFS and FBTC ensure that their minimum operating voltage is the lowest possible, extending the amount of MCU instructions executed within a single power cycle. This, in combination of operating at the minimum possible voltage for the selected operating frequency, grants D^{2}VFS and FBTC a lower energy consumption than the baselines, as shown in Fig. 17(d). The lower energy consumption of D^{2}VFS and FBTC allows them to complete Dijkstra within a single power cycle and without entering Hibernus' hibernation mode, as Fig. 17(f) and Fig. 17(g) show, respectively. We recall that Hibernus enters hibernation mode when the voltage of the energy buffer drops below a pre-defined threshold [8], and it saves the state before entering hibernation mode. Note that the static 1MHz and 8Mhz configurations demonstrate a similar behaviour. However, thanks to frequency scaling, D^{2}VFS and FBTC have a faster execution time, demonstrating to be a better solution than these two static configurations.

Let us now focus on FFT and RSA, where we can notice a change in the performance trend. Here D^{2}VFS and FBTC no longer perform better than all the static configurations. Compared to the best performing baseline, that is, 12MHz, D^{2}VFS and FBTC are 2.1x slower (Fig. 17(a)) and consume, on average, 56% and 15% more energy (Fig. 17(d)), respectively. The reason behind this performance change is the reduced voltage range that the lower static frequency configurations, D^{2}VFS, and FBTC have when entering hibernation mode. When Hibernus enters hibernation mode, it sets the MCU to a low-power mode whose minimum operating voltage is 1.8V. The voltage threshold that triggers hibernation mode depends on the minimum operating voltage of the MCU, which is frequency-dependent. As such, the higher static frequencies configurations, such as 12MHz and 16MHz, enter low-power mode at a higher voltage than D^{2}VFS, FBTC, and the lower static frequency configurations, as the former group has a higher minimum operating voltage than the latter. Note that D^{2}VFS and FBTC eventually scale the frequency down to 1MHz, extending the power cycle length at the expenses of a more depleted energy buffer. Consequently, the static 12MHz and 16MHz configurations enter hibernation mode with a higher energy buffer level than D^{2}VFS, FBTC, and the static 1MHz and 8MHz configurations, thus resulting in a longer period waiting for new incoming energy.

The medium-capability energy source we consider for these experiments supply short energy bursts that are 5s apart from each other, as shown in Fig. 10(b), which lead to long periods where the MCU is either powered off or in low-power mode, waiting for new incoming energy. Consequently, D^{2}VFS and FBTC enter hibernation

mode later than the baselines and the nature of the energy source does not allow them to wait in hibernation mode, as the energy bursts are too far from each other, causing a power failure. The number of state saves and power failures shown in Fig. 17(g) and Fig. 17(f), respectively, provide evidence of this behaviour. Both in FFT and RSA, D^{2}VFS and FBTC enter hibernation mode and then experience one power failure, whereas $12MHz$ and $16MHz$ have sufficient energy to wait for new incoming energy bursts. Further, in RSA, the static $16MHz$ configuration enters hibernation mode twice, without experiencing any power failure.

Due to the nature of the medium-capability energy source, we can notice from Fig. 17(a), Fig. 17(c), and Fig. 17(b) that the recharge time represents most of the completion time, whereas the execution time only represents a small portion of the latter. Note that we consider the recharge time as the time where the MCU is not executing any code and it is waiting for new energy to be harvested to resume the execution, whereas the execution time as the time where the MCU is performing active computation. In RSA, the recharge time of the best static frequency confiuration, that is, $12MHz$, is 95% of the total completion time, whereas in D^{2}VFS and FBTC the recharge time is 97% of the completion time. Note that the increase of recharge time is another consequence of entering hibernation mode with little energy to wait in low-power mode. The more depleted energy buffer of D^{2}VFS and FBTC results in a $2.1x$ higher recharge time than the static $12MHz$ configuration, as D^{2}VFS and FBTC need to recharge the capacitor to V_{on} starting from a lower voltage than the static $12MHz$ configuration.

Finally, on average, FBTC shows a 0.01% faster completion time than D^{2}VFS, while showing a 24% lower energy consumption than D^{2}VFS across all benchmarks. Futher, Fig. 17(e) shows that D^{2}VFS components have a higher impact on the energy consumption than FBTC components. D^{2}VFS components are responsible for up to 44% of the total energy consumption, whereas FBTC components responsible only for up to 11% of the energy consumption, consuming on average $6.6x$ less energy than the components of D^{2}VFS.

Hibernus with C = 20μF. In Sec. 5.1 we argue that a system with a $80\mu F$ capacitor requires a voltage doubler to ensure program forward progress, as the medium-capability energy source is not able to recharge a $80\mu F$ capacitor to $3.6V$, that is, V_{on}. However, we also point out that using a voltage doubler may not always be an option. As such, we execute our experiments using also a smaller capacitor, that is, $20\mu F$, which does not require the use of a voltage doubler. Note that here we do not consider the static $16MHz$ configuration, as it is not able to achieve forward progress with such small capacitor size.

Fig. 18 depicts the experiment results with the $20\mu F$, showing a change in the performance trend when lowering the capacitor size: D^{2}VFS and FBTC now outperform all the baselines. The key reson behind such performance change stands in the energy buffer size: entering low-power mode with a depleted capacitor no longer pones a disadvantage for D^{2}VFS and FBTC when used with Hibernus. In fact, as Fig. 18(b) shows, the recharge time of D^{2}VFS and FBTC now is within the one of the best performing baseline. Note that now the recharge time represents up to 99% of the total completion time.

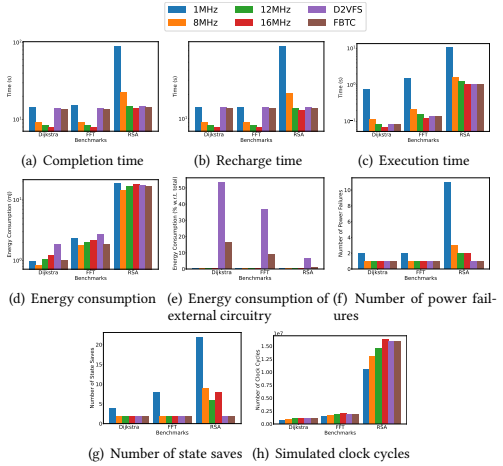
Fig. 18(a) shows that D^{2}VFS and FBTC complete the benchmarks $5.4x$ times faster than the static $1MHz$ configuration, that is, the worse performing baseline. Moreover, D^{2}VFS and FBTC show a similar performance against the two static $8MHz$ and $12MHz$ configurations. On average, D^{2}VFS and FBTC are are 0.68% and 0.4% faster than these two static configurations, respectively. Further, FBTC now demonstrates the lowest energy consumption: on average, it consumes 22% less energy than all the baselines and up to 72% less than the worse performing baseline, that is, $1MHz$. Instead, D^{2}VFS higher quiescent current results, on average, in a 22% higher energy consumption than all the baselines. Note that, thanks to its lower quiescent current, FBTC consumes, on average, 44% less energy than D^{2}VFS.

These are all consequences of D^{2}VFS and FBTC voltage and frequency scaling technique, as D^{2}VFS and FBTC are able to set the MCU operating frequency to $16MHz$, operating in a more efficient condition than all the baselines. Moreover, operating at $16Mhz$ overcompensates for the active portion of power cycles where D^{2}VFS and FBTC operate at $1MHz$. Compared to the $80\mu F$ case, this results to an overall faster execution time, as Fig. 18(c) shows. Further, the higher is the number clock cycles in the workload, the faster D^{2}VFS and FBTC complete the benchmarks with respect to static frequency configurations. This is the case of the RSA, compared to Dijkstra and FFT.

The change in the energy consumption trend affects the number of power failures, shown in Fig. 18(f): all the baselines now experience a power failure, wheras with a $80\mu F$ capacitor the static $12MHz$ configuration was exempted from power failures. Moreover, the baselines,D^{2}VFS, and FBTC no longer complete Dijkstra benchmark in one power cycle. This is a consequence of a smaller energy buffer, as the capacitor now can store less energy that can be used to progress in the program execution while the energy source is not supplying sufficient energy to sustain the computation. A similar trend happens in the number of state saves, shown in Fig. 18(g).

Finally, we notice the same performance difference between D^{2}VFS and FBTC that happens with the $80\mu F$ capacitor. On average, FBTC is 0.27% slower than D^{2}VFS, while consuming 44% less energy. However, there is now an increase of the overall energy consumption of D^{2}VFS and FBTC components due to higher recharge times than the experiments with a $80\mu F$ capacitor. Fig. 18(e) shows that D^{2}VFS circuitry is now responsible for up to 57% of the total energy consumption, whereas FBTC circuitry is responsible only for up to 15% of it, consuming an average of $4.7x$ less energy than the circuitry of D^{2}VFS.

Mementos with C = 80μF. We discuss now the experiment results with Mementos. We run the experiments considering the three Mementos configurations, namely `Default`, `NOADCOFF`, and `ADCMINV`, which affect only the behaviour of D^{2}VFS, FBTC, and the static $1MHz$ configuration. Our experiments show no change in the performance trend between the techniques affected by Mementos configuration and the other baselines. The only significant difference across the three configurations stands in the number of state saves for such frequencies. `ADCMINV`, on average, shows a 31% ($25x$) higher (lower) number of state saves than `NOADCOFF` (`Default`). However, this difference does not have a significant impact on performance: `ADCMINV`, on average, shows a 4.5% (5.1%) higher (lower) energy consumption than `NOADCOFF` (`Default`) and a 5.2% (3.2%) lower completion time than `NOADCOFF` (`Default`). For these reasons,

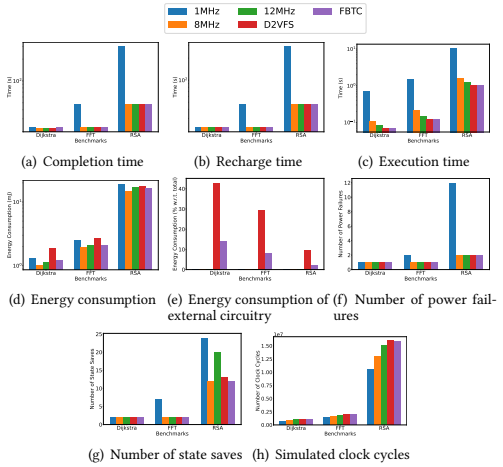


(a) Completion time (b) Recharge time (c) Execution time

(d) Energy consumption (e) Energy consumption of external circuitry (f) Number of power failures

(g) Number of state saves (h) Simulated clock cycles

Figure 19: Benchmarks results with a medium-capability energy source, Mementos in ADCMINV configuration, C = 80μF, and V_{on} = 3.6V.



(a) Completion time (b) Recharge time (c) Execution time

(d) Energy consumption (e) Energy consumption of external circuitry (f) Number of power failures

(g) Number of state saves (h) Simulated clock cycles

Figure 20: Benchmarks results with a medium-capability energy source, Mementos in ADCMINV configuration, C = 20μF, and V_{on} = 3.6V.

we report here only the results of ADCMINV, as (i) the performance trend is unchanged and the performance difference is not significant, despite the variance in the number of state saves, and (ii) across the three configurations, this represents the most reasonable choice for a real-world deployment.

Fig. 19 shows the experiment results with Mementos in ADCMINV configuration and a 80μF capacitor.

Fig. 19(a) depicts the completion time of each benchmark, where the static 16Mhz configuration demonstrates the fastest completion time. Similarly to what we argue in the Hibernus experiments, the reason of such better performance stands in the small operating range of the static 16MHz configuration, which allows the MCU to resume the computation faster, as the capacitor require less energy to reach V_{on} from a power-off state. Conversely to Hibernus, in these experiments the MCU does not enter low-power mode and the MCU directly powers off when the minimum operating voltage is reached. Further, here the effect of powering off with a higher capacitor voltage is more beneficial than with Hibernus: in the baselines, the capacitor does not discharge due to quiescent current consumption of Hibernus external comparators, as Mementos does not require any external component. Consequently, for these reasons, the baselines recharge back to V_{on} faster, thus resuming the computation faster than D^{2}VFS and FBTC, which instead are either operating in a slower and less efficient frequency or powered off waiting for the energy buffer to be recharged to V_{on}.

The static 16MHz configuration terminates both Dijkstra and FFT benchmarks 72% (78%) faster than FBTC (D^{2}VFS). A similar case happens also for the static 12MHz configuration, which terminates both Dijkstra and FFT benchmarks 64% (70%) faster than FBTC (D^{2}VFS). However, the behaviour we previously describe represents an advantage only with short-length workloads, which in this case are Dijkstra and FFT. RSA requires the MCU to execute 14x (7x) more clock cycles than Dijkstra (FFT), as Fig. 20(h) shows. Both 12MHz and 16MHz now experience twice the number of power failures that D^{2}VFS and FBTC experience. As such, the static 16MHz configuration almost loses all the advantage, as in RSA it is only 4% (8%) faster than FBTC (D^{2}VFS). Instead, 12Mhz loses all the performance advantage of its reduced operating voltage range: it is now 1.4% slower than FBTC, while remaining 2% faster than D^{2}VFS. This change in the trend is thanks to D^{2}VFS and FBTC ability to operate at a more efficient voltage and frequency range.

Note that, similarly to the Hibernus experiments, the completion time is mainly affected by the recharge time, as Fig. 20(a) and Fig. 20(b) show. As we can see from the performance trend across the three benchmarks, the more clock cycles are executed, the more the effect of a reduced operating voltage range is attenuated. Fig. 20(c) provides evidence of this behaviour: the higher the number of clock cycles executed, the faster is the execution time of D^{2}VFS and FBTC. In Dijkstra and FFT, D^{2}VFS and FBTC execution time is within the execution time of 12MHz, whereas in RSA they match the one of 16MHz. Considering that a deployed system runs the same workload forver, we believe that in the long run D^{2}VFS and FBTC have significantly faster completion time compared to the baselines, as the execution has an infinite number of clock cycles.

Let us now focus on Fig. 19(d), which depicts the energy consumed to complete the benchmarks. The static 8MHz configuration has the most efficient energy consumption, which however does not translate to the fastest completion time, as we described in Fig. 19(a). Among the three benchmarks, D^{2}VFS always shows one of the highest energy consumption, consuming on average 66% more energy than the static 8MHz configuration. Instead, on average, FBTC consumes 45% less energy than D^{2}VFS and 12% more

energy than the static $8MHz$ configuration, always resulting among the most efficient configuration across each benchmark.

Two factors influence D^{2}VFS and FBTC higher energy consumption than the static $8MHz$ configuration. First, as we point out in our experiments with the high-capability energy source, ADC probing introduces a clock cycle penalty that increases with the MCU operating frequency. The higher is the frequency, the higher is the number of executed clock cycles to complete the benchmark. For example, as Fig. 19(h) shows, $16MHz$ need to execute, on average, 44% more clock cycles than $1MHz$ to complete the benchmarks. Hence, ADC probing makes D^{2}VFS and FBTC to pay a higher penalty than the static $8MHz$, as D^{2}VFS and FBTC execute a portion of the program at $16MHz$ and $12MHz$, which have a higher penalty than the static $8MHz$ configuration. Second, D^{2}VFS and FBTC have a quiescent current draw that is not present in the baselines.

Despite the higher energy consumption, in Dijkstra and FFT, D^{2}VFS and FBTC experience the same number of power failures of the baselines, as Fig. 19(f) shows. Instead, in RSA, D^{2}VFS and FBTC experience only one power failure, whereas the baselines experience at least twice this number. This demonstrates that, despite the higher energy consumption, D^{2}VFS and FBTC are able to manage energy more efficiently, as they experience less power failures. This translates to a similar trend in the number of state saves, as Fig. 19(g) shows.

Finally, the more efficient voltage and frequency scaling circuitry of FBTC demonstrates, on average, a 45% lower energy consumption and a 3.6% faster compleition time than D^{2}VFS. Note that the higher quiescent current draw of D^{2}VFS components is responsible, on average, for the 33% of the overall energy consumption, whereas FBTC components only for the 9%, as shown in Fig. 19(e). This causes D^{2}VFS to consume $4.5x$ more energy than FBTC when the MCU is powered off and recharges its energy buffer, causing the recharge time of D^{2}VFS to be 4% higher than the one of FBTC, as Fig. 19(b) shows.

Mementos with C = 20μF. In Sec. 5.1 we argue that using a $80\mu F$ capacitor with the medium-capability energy source we consider is not possible in cases not compatible with using a voltage doubler. To account for this situation, we also run our experiments with a $20\mu F$ capacitor, which does not require a voltage doubler to ensure forward progress across power failures. Note that, for similar reasons to the experiments with the $80\mu F$ capacitor, we discuss only the results for the `ADCMINV`configuration of Mementos.

Fig. 20 depicts the experiment results with a $20\mu F$ capacitor and Mementos in `ADCMINV`configuration. Similarly to the experiments with Hibernus and a $20\mu F$ capacitor, we notice a performance trend change for the completion time with respect to the experiments with a $80\mu F$ capacitor. Fig. 20(a) shows that there is negligible difference in the completion time between D^{2}VFS, FBTC and the static $8MHz$ and $12MHz$ configurations. Compared to these two baselines, on average, FBTC (D^{2}VFS) has a 0.13% (0.25%) slower (faster) completion time. Note that, in RSA, that is, the benchmark with the highest number of executed clock cycles, FBTC and D^{2}VFS are 0.11% and 0.37% faster than the static $12MHz$ configuration, respectively, that is, the fastest baseline.

The reason behind such changes is the same for the Hibernus case: the capacitor size no longer pones a disadvantage for D^{2}VFS and FBTC extended voltage range. Consequently, D^{2}VFS and FBTC recharge time is now in par with the one of the baselines, as Fig. 20(b) shows. This demonstrates that the quiescent current of D^{2}VFS and FBTC have a low impact on performance while the MCU is powered off. In fact, the recharge time of D^{2}VFS and FBTC is similar to the one of the baselines, which in contrast do not have such quiescent current draw.

Further, Fig. 20(c) shows that D^{2}VFS and FBTC demonstrate an execution time that is, on average, $3.5x$ faster than all the baselines and at least 16% faster than the best performing baseline, that is, the static $12MHz$ configuration. Key behind such greater performance is D^{2}VFS and FBTC voltage and frequency scaling technique. In fact, despite the inability of the static $16MHz$ configuration to execute with the $20\mu F$ capacitor, D^{2}VFS and FBTC set the MCU to operate at $16MHz$ for a portion of each power cycle, which is the fastest and most efficient operating frequency. This makes D^{2}VFS and FBTC able to extract the most possible performance out of harvested energy.

Compared to the experiments with a $80\mu F$ capacitor, there is very little difference in the performance trend for the energy consumption, number of power failures, and executed clock cycles. D^{2}VFS and FBTC experience the same number of power failures than the baselines except the static $1MHz$ configuration, which experiences a up to $6x$ higher number of power failures, as Fig. 20(f) shows.

Finally, we notice the same performance difference between D^{2}VFS and FBTC that happens with the $80\mu F$ capacitor, as FBTC is only 0.28% slower than D^{2}VFS, while demonstrating a 30% lower energy consumption. Similarly to the experiments with the $80\mu F$ capacitor, the higher quiescent current draw of D^{2}VFS components is responsible, on average, for the 27% of the overall energy consumption, whereas FBTC components only for the 8%, as shown in Fig. 20(e).

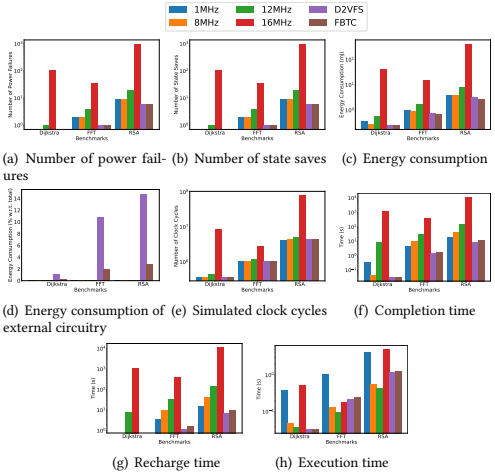
5.6 Results → Low-capability Energy Source

We discuss here the results for the experiments with the low-capability energy source. Note that V_{on} is set to $3.6V$.

Hibernus. Fig. 21 shows the experiment results with Hibernus and a $80\mu F$ capacitor. D^{2}VFS and FBTC demonstrate the best overall performance against all the baselines.

Due to its nature, the low-capability energy source does not supply any energy during the computation. Hence, conversely to what happens with the high-capability and medium-capability energy sources, the length of the active portion of a power cycles is fixed and strictly depends on the minimum possible operating voltage of the selected MCU operating frequency. The voltage and frequency scaling techniques of D^{2}VFS and FBTC represent the key factor behind their performance, as they ensure that the MCU operates at the maximum possible frequency and minimum possible voltage, thus ensuring the maximum possible operating voltage range without the drawbacks of executing at a low operating frequency. This extends significantly the number of clock cycles executed within a single power cycle, providing a significant benefit on the overall performance.

Fig. 21(f) depicts the completion time of each benchmark. D^{2}VFS and FBTC are, on average, three orders of magnitude faster than the baselines. Compared to the best performing baseline of each

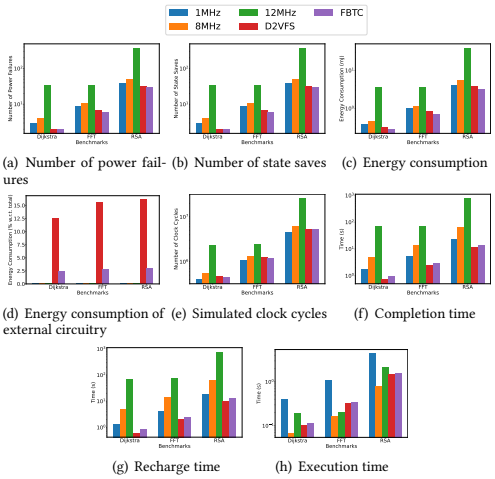


(a) Number of power failures (b) Number of state saves (c) Energy consumption

(d) Energy consumption of external circuitry (e) Simulated clock cycles (f) Completion time

(g) Recharge time (h) Execution time

Figure 21: Benchmarks results with the low-capability energy source, $C = 80\mu$F, $V_{on} = 3.6V$, and Hibernus.



(a) Number of power failures (b) Number of state saves (c) Energy consumption

(d) Energy consumption of external circuitry (e) Simulated clock cycles (f) Completion time

(g) Recharge time (h) Execution time

Figure 22: Benchmarks results with the low-capability energy source, $C = 20\mu$F, $V_{on} = 3.6V$, and Hibernus.

benchmark, D^{2}VFS and FBTC demonstrate, on average, a 134% and 87% faster completion time, respectively.

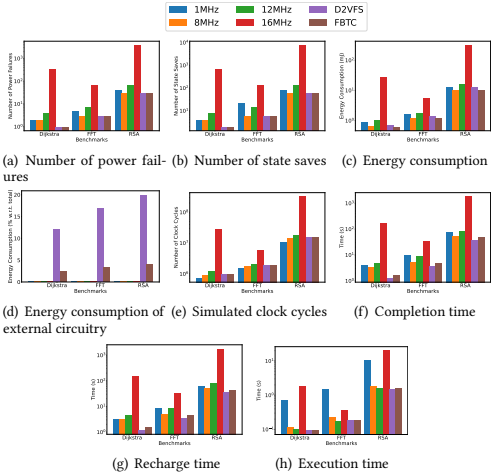
We must note that extending the lifespan of a power cycle by lowering the clock frequency increases the time required to execute a portion of the instructions executed within a single power cycle, as Fig. 21(h) depicts. Consequently, D^{2}VFS and FBTC have a slower active cycle than some baselines, resulting in a slower execution time. For example, in FFT and RSA, D^{2}VFS and FBTC, on average, are respectively 91% and 111% slower than the static $12MHz$ configuration, that is, the baseline with the fastest execution time. However, such increase in the active cycle execution time cames at the significant advantage of a higher number of instructions executed within a single power cycle, which significantly lowers the number of power cycles required to compelte a given workload, as we previously argue. Moreover, D^{2}VFS and FBTC waste less time than the baselines in waiting for new incoming energy, significantly reducing the recharge time, as shown in Fig. 21(g). Note that here D^{2}VFS and FBTC recharge time is, on average, two orders of magnitude lower than the baselines. Consequently, as we previously show, their completion time is significantly lower than the baselines.

D^{2}VFS and FBTC demonstrate a significantly lower energy consumption than all the baselines. Fig. 21(c) shows that D^{2}VFS and FBTC consume, on average, $27x$ and $29x$ less energy than the static frequency configurations, respectively. This is thanks to D^{2}VFS and FBTC dynamic voltange and frequency scaling techniques, which allows them to operate at the most efficient conditions. Further, Fig. 21(a) shows that D^{2}VFS and FBTC are able to complete the Dijkstra benchmark within a single power cycle, whereas in FFT and RSA D^{2}VFS and FBTC experience, on average, $26x$ less power failures than the baselines. Moreover, in FFT and RSA, D^{2}VFS and FBTC experience half the number of power failures than the static $8MHz$ configuration, which is the baseline experiencing the lowest number of power failures. This behaviour is consequence of D^{2}VFS and FBTC ability to extend the number of instructions executed within a single power cycle, which also results in a reduction of the number of power cycles required to complete a given workload.

Note that the energy consumption and the number of power failures affect each other: a decrease in the energy consumption reduces the number of power failures, thus reducing the number of power cycles required to complete a given workload; a decreaase in the number of power failures reduces the energy consumed due to state-saving and state-restoring operations, thus reducing the overall energy consumption. Moreover, the reduction of power failures results in a reduction of state-save and state-restore operations, further reducing the number of clock cycles executed to complete a given workload. Fig. 22(e) shows that, on average, FBTC and D^{2}VFS execute $4.3x$ less clock cycles than all the baselines.

Finally, the lower quiescent current of FBTC results, on average, in a 9% lower energy consumption than D^{2}VFS, as shown in Fig. 21(c). Further, D^{2}VFS components are responsible for up to 16% of the total energy consumption, wheras FBTC components do not exceed 3% of it, as shown in Fig. 21(d). However, a higher energy consumption means a lower equivalent resistance that enables a faster recharge of the capacitor. Fig. 21(g) shows that the lower resistance of D^{2}VFS results, on average, in a 37% faster recharge time than FBTC. This affects the completion time, as D^{2}VFS has, on average, a 33% faster completion time than FBTC, as Fig. 21(f) shows.

When switching to a $20\mu F$ capacitor, the performance trend does not change, as D^{2}VFS and FBTC still outperform all the baselines. We show the experiment results with a $20\mu F$ capacitor in Fig. 22. D^{2}VFS and FBTC demonstrate the lowest completion time (Fig. 22(f)) and the lowest energy consumption (Fig. 22(c)), which

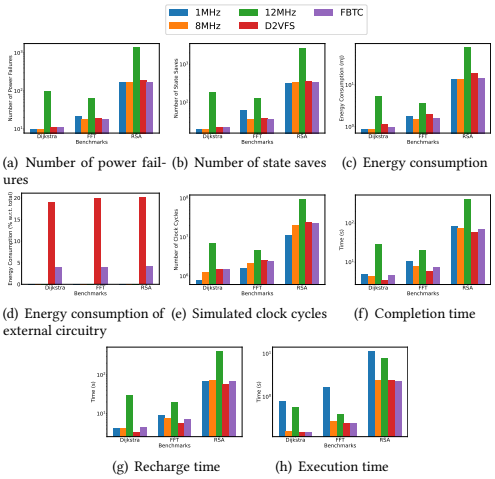


(a) Number of power failures (b) Number of state saves (c) Energy consumption

(d) Energy consumption of external circuitry (e) Simulated clock cycles (f) Completion time

(g) Recharge time (h) Execution time

Figure 23: Benchmarks results with the low-capability energy source, $C = 80\mu$F, $V_{on} = 3.6V$, and Mementos.



(a) Number of power failures (b) Number of state saves (c) Energy consumption

(d) Energy consumption of external circuitry (e) Simulated clock cycles (f) Completion time

(g) Recharge time (h) Execution time

Figure 24: Benchmarks results with the low-capability energy source, $C = 20\mu$F, $V_{on} = 3.6V$, and Mementos.

lead D^{2}VFS and FBTC to experience the lowest number of power failures (Fig. 22(a)). On average, FBTC has a 17% lower energy consumption than D^{2}VFS, which allows FBTC to experience up to 17% less power failures than D^{2}VFS, as shown in Fig. 22(a). However, similarly to what happens with the 80μF capacitor, D^{2}VFS has a 27% faster completion time than FBTC.

Mementos. We now discuss our experiment results with the low-capability energy source and Mementos.

We run each experiment considering all the three Mementos configurations. Similarly to the experiments with the medium-capability energy source, the performance trend is unchanged across the three Mementos configuration, despite the performance difference in FBTC, D^{2}VFS, and the baselines. From a performance standpoint, `ADCMINV`represent the average case between `NOADCOFF`and `Default`. `ADCMINV`, on average, shows a 39% (28.3x) higher (lower) number of state saves than `NOADCOFF` (`Default`), which results in a 15% (11%) higher (lower) energy consumption than `NOADCOFF` (`Default`) and a 14% (21%) slower (faster) completion time than `NOADCOFF` (`Default`). Moreover, as we previously argue, `ADCMINV`represents the most reasonable choice among the three configurations of Mementos. Note that a similar case happens with the 20μF capacitor. For these reasons, we discuss here only the results with the `ADCMINV`configuration for Mementos.

Fig. 23 shows the experiment results with the `ADCMINV`configuration of Mementos and a 80μF capacitor. The performance difference between D^{2}VFS, FBTC, and the baselines across each benchmark shows a trend similar to the one of the experiments with Hibernus. However, there is a performance difference introduced by the executions of Mementos' probe function. As we previously discuss, the probe function of Mementos accesses the ADC, introducing a latency that increases with the MCU operating frequency.

Considering the performance of Hibernus experiments of Fig. 21 as a reference, here 1MHz is the frequency that is mostly affected by ADC accesses penalty, as we the completion time shown in Fig. 23(c) demonstrates, whereas 8MHz is the frequency less affected. Despite a performance change, the performance trend between D^{2}VFS, FBTC and the baselines does not change.

In general, the static 8MHz configuration is the best performing baseline, and both D^{2}VFS and FBTC outperform it. On average, FBTC and D^{2}VFS show, respectively, a 42% and 84% faster completion time than the static 8MHz configuration, as Fig. 23(f) depicts. Moreover, we can notice from Fig. 23(f) that, on average, FBTC (D^{2}VFS) shows a 3.5% (0.81%) lower (higher) energy consumption than the static 8MHz configuration. Not only FBTC has a lower energy consumption and execution time than the static 8MHz configuration, but also D^{2}VFS has an overall better performance than the static 8MHz configuration, as it demonstrates a faster completion time under a similar energy consumption. As we previously argue, the key behind D^{2}VFS and FBTC performance stands in the voltage and frequency scaling technique, which allows them to operate in the most efficient setting and to extend the length of power cycles active periods.

Similarly to what we argue for the Hibernus experiments with the low-capability energy source, FBTC demonstrates, on average, a 19% lower energy consumption but a 29% higher completion time than D^{2}VFS. The reason behind such performance difference stands in the lower quiescent current of FBTC, which is responsible for no more than 4% of the total energy consumption, whereas D^{2}VFS components are responsible for up to 20% of the total energy consumption. Despite the lower quiescent current of FBTC unlocks more energy-efficient operations, it provides a higher equivalent resistance that negatively affects the recharge of the energy buffer.

Finally, Fig. 24 depicts the experiment results with Mementos in ADCMINVconfiguration and a $20\mu F$ capacitor. These results lead us to the same conclusion of the experiments with a $80\mu F$ capacitor, as there is no significant difference in the performance trend between the experiments with the two capacitor sizes. D^{2}VFS and FBTC demonstrate up to $3.1x$ and $2.4x$ faster completion time than the baselines and up to $1.7x$ and $2.1x$ lower energy consumption, respectively. Compared to the best performing baseline, that is, the $8MHz$ configuration, on average, FBTC (D^{2}VFS) has a 5% (31.78%) higher energy consumption and a 2.5% (29%) lower completion time. Similarly to the $80\mu F$ capacitor experiments, FBTC demonstrates an average of a 26% lower energy consumption and a 26% higher completion time than D^{2}VFS.

5.7 Results → Final Remarks

D^{2}VFS and FBTC show exceptional performance with high-capability and low-capability energy sources, as their ability to efficiently extend the number of instructions executed within a single power cycle demonstrates a significantly lower completion time and a lower energy consumption than all the baselines. This allows D^{2}VFS and FBTC to terminate the benchmarks in a lower number of power cycles than the baselines, providing a significant performance gain. Instead, with a medium-capability energy source, D^{2}VFS and FBTC performance is comparable to the best performing baseline, especilly when a large energy buffer is used.

We identified three different supply scenarios, which generalizes across a vast majority of possible energy supply conditions. From the results of these experiments, we can see that there is no best configuration among the static ones that demonstrates to have the best performance across all the different energy supply and workloads scenarios. For example, the static $16Mhz$ configuration is the best performing baseline with the high-capability energy source, but it becames the worse performing baseline with the low-capability energy source.

Energy harvesting sources are unpredictable and environmental changes may affect the energy supply pattern [9], resulting in a change in the energy supply scenario. For example, a cloud covering the sun causes the energy supply scenario to change from high-capability to low-capability. In such condition, D^{2}VFS and FBTC demonstrate to provide a good general setting: conversely to the static frequency configurations, their performance trend is not drastically affected by changes in the energy supply scenario. This is thanks to their voltage and frequency scaling technique, which, as we argue in our experiment results discussion, allows D^{2}VFS and FBTC to operate at the most efficient setting that provides the best possible performance. For this reason, we believe that D^{2}VFS and FBTC represent a valuable technique for the design and development of battery-less systems, as it relief system designers from profiling the energy source and running multiple and laborious experiments to identify a system setting that may not serve good performance with changes in the energy supply scenario.

REFERENCES

[1] 2011 (last access: Jul 1st, 2022). RF energy traces used in Mementos. https://github.com/ransford/mspsim/tree/mementos/traces.

[2] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys '20)*.

[3] S. Ahmed, Q. Ain, J. H. Siddiqui, L. Mottola, and M. H. Alizai. 2020. Intermittent Computing with Dynamic Voltage and Frequency Scaling. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks (EWSN '20)*.

[4] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[5] D. Balsamo, A. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Graceful Performance Modulation for Power-Neutral Transient Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[6] D. Balsamo, B. J. Fletcher, A. S. Weddell, G. Karatziolas, B. M. Al-Hashimi, and G. V. Merrett. 2019. Momentum: Power-Neutral Performance Scaling with Intrinsic MPPT for Energy Harvesting Computing Systems. *ACM Transactions on Embedded Computing Systems* (2019).

[7] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[8] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[9] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[10] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[11] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SENSYS)*.

[12] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[13] A. Colin and B. Lucia. 2018. Termination Checking and Task Decomposition for Task-based Intermittent Programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*.

[14] P. Favrat, P. Deval, and M.J. Declercq. 1998. A high-efficiency CMOS voltage doubler. *IEEE Journal of Solid-State Circuits* (1998).

[15] B. J. Fletcher, D. Balsamo, and G. V. Merrett. 2017. Power Neutral Performance Scaling for Energy Harvesting MP-SoCs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*.

[16] F. Fraternali, B. Balaji, Y. Agarwal, L. Benini, and R. Gupta. 2018. Pible: Battery-Free Mote for Perpetual Indoor BLE Applications. In *Proceedings of the 5th Conference on Systems for Built Environments (BUILDSYS)*.

[17] M. Furlong, J. Hester, K. Storer, and J. Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*.

[18] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia. 2019. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*.

[20] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*.

[21] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[22] M. Hicks. 2016 (last access: Oct 15th, 2021). MiBench2 - MiBench porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[23] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[24] Texas Instruments. 1998 (last access: Dec 22nd, 2022). SN74LV175A Quadruple D-Type Flip-Flops. https://www.ti.com/lit/ds/symlink/sn74lv175a.pdf.

[25] Texas Instruments. 2009 (last access: Dec 22nd, 2022). Low-Power Dual 2-input Positive-NOR Gate. https://www.ti.com/lit/ds/symlink/sn74aup2g02.pdf.

[26] Texas Instruments. 2013 (last access: Jul 1st, 2022). MSP430-G2553 datasheet. https://www.ti.com/lit/ds/symlink/msp430g2553.pdf.

[27] Texas Instruments. 2014 (last access: Dec 22nd, 2022). SN74AUP1G04 Low-Power Single Inverter Gate. https://www.ti.com/lit/ds/symlink/sn74aup1g04.pdf.

[28] Texas Instruments. 2014 (last access: Dec 22nd, 2022). TPS6274x Step Down Converter Datasheet. https://www.ti.com/lit/ds/symlink/tps62740.pdf.

[29] Texas Instruments. 2016 (last access: Dec 22nd, 2022). SN74AUP1G08 Low-Power Single 2-Input Positive-AND Gate. https://www.ti.com/lit/ds/symlink/sn74aup1g08.pdf.

[30] Texas Instruments. 2017 (last access: Oct 15th, 2021). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[31] Texas Instruments. 2020 (last access: Jul 1st, 2022). MSP430 Hardware Design Tips. https://www.ti.com/seclit/ml/sprpe57/sprpe57.pdf.

[32] Texas Instruments. (last access: Jul 1st, 2022). MSP430 family of MCUs. https://www.ti.com/msp430.

[33] B. Islam and S. Nirjon. 2020. Scheduling Computational and Energy Harvesting Tasks in Deadline-Aware Intermittent Systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[34] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[35] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Transactions on Embedded Computing Systems* (2017).

[36] Fujitsu Semiconductor Limited. 2015 (last access: Jul 1st, 2022). MB85RC64V 8Kb I^2C FeRAM datasheet. https://www.fujitsu.com/jp/group/fsm/en/documents/products/fram/lineup/MB85RC64V-DS501-00013-7v0-E.pdf.

[37] llvm 2003 (last access: Oct 15th, 2021). The LLVM Compiler Infrastructure. https://llvm.org/.

[38] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[39] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[40] K. Maeng and B. Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[41] K. Maeng and B. Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*.

[42] A. Maioli. 2022 (last access: Jul 1st, 2022). ScEpTIC extension implementing system energy emulation. http://sceptic.neslab.it/.

[43] A. Maioli and L. Mottola. 2020. Intermittence Anomalies Not Considered Harmful. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20)*.

[44] A. Maioli and L. Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys '21)*.

[45] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[46] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021)*.

[47] A. Y. Majid, C. Delle Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak. 2020. Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices. *ACM Transactions on Sensor Networks* (2020).

[48] R. Menon, R. Gujarathi, A. Saffari, and J. R. Smith. 2023. Wireless Identification and Sensing Platform Version 6.0. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems (SenSys '22)*.

[49] Nexperia. 2021 (last access: Dec 22nd, 2022). 74HC85 4-bit Magnitude Comparator. https://www.mouser.it/datasheet/2/916/74HC_HCT85-1541793.pdf.

[50] PeakTech. 2022 (last access: Dec 22nd, 2022). PeakTech 6225A Variable Power Supply. https://peaktech-rce.com/en/laboratory-power-supplies/441-peaktech-6225a-laboratory-switching-power-supply-dc-0-30v-0-5a-digital-meters.html.

[51] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[52] E. Sazonov, H. Li, D. Curry, and P. Pillay. 2009. Self-Powered Sensors for Monitoring of Highway Bridges. *IEEE Sensors Journal* (2009).

[53] ROHM Semiconductor. 2015 (last access: Dec 22nd, 2022). BU49XXG CMOS Voltage Detector. https://www.mouser.it/datasheet/2/348/bu48xxg_e-1874410.pdf.

[54] STMicroelectronics. 2013 (last access: Dec 22nd, 2022). TS881 Comparator. https://www.st.com/resource/en/datasheet/ts881.pdf.

[55] M. Surbatovich, L. Jia, and B. Lucia. 2021. Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*.

[56] UNI-TREND Technology. 2022 (last access: Dec 22nd, 2022). UNI-T UT61E Digital Multimeter. https://meters.uni-trend.com/product/ut61plus-series/.

[57] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[58] K. Vijayaraghavan and R. Rajamani. 2010. Novel Batteryless Wireless Sensor for Traffic-Flow Measurement. *IEEE Transactions on Vehicular Technology* (2010).

[59] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.