POLITECNICO DI MILANO
DEPARTMENT OF ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

SYSTEM SUPPORT FOR TRANSIENTLY-POWERED EMBEDDED SENSING SYSTEMS

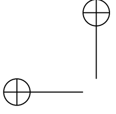
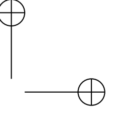
Doctoral Dissertation of:
Naveed Anwar Bhatti

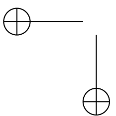
Supervisor:
Prof. Luca Mottola

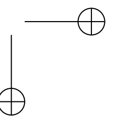
Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Andrea Bonarani

2017 – XXX





Acknowledgement

I would like to start by saying, I feel that I am truly blessed by the Almighty to be standing where I am today.

To begin with, I owe my most profound gratitude to my advisor, Luca Mottola, for his advice, ideas, funding, and unfailing optimism since before I arrived in Milan. From igniting in me a spark of interest for transiently-powered systems to guiding my Ph.D. to its end, Luca has been a friend and mentor during these last three years and I am grateful for the enthusiasm, energy and time Luca has devoted in my work.

I want to offer my gratitude to my tutor Luciano Baresi. I also want to thank my co-authors for their efforts at LUMS: Muhammad Hamad Alizai, Saad Ahmed, and Junaid Haroon.

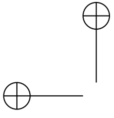
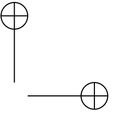
I appreciate my colleagues at Politecnico di Milano who provided me a warm and welcoming social climate for research. Many ideas that resulted in the research, presented in this thesis, had their origin in (late night) discussions with Kapal Dev (my roommate). I would also like to thank Mikhail Afanasov, Koustabh Doloui, Paolo Manca and Riccardo Paccagnella. These are the amazing people I have shared the office with over the span of three years.

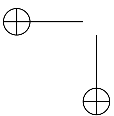
I would like to thank my parents for their love, encouragement, and support throughout my entire education. Without their motivation, I wouldn't be here today. Also, my wife's parents deserve my gratitude for all their support and encouragement throughout.

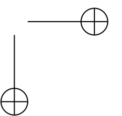
And Finally, I would like to express my deepest gratitude towards my dearest wife Hala, who has had been my backbone in this entire journey. I believe her love and motivation has had me achieve many milestones I couldn't have reached otherwise.

Thanks to all of you!

`This research was partly funded by by the Cluster Projects ”Zero-energy Buildings in Smart Urban Districts" (EEB), ”ICT Solutions to Support Logistics and Transport Processes" (ITS), and ”Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.`





Abstract

TRANSIENTLY powered embedded systems are becoming popular because of their self-sustainable, no maintenance and easily deployable nature. However, there is an intrinsic challenge with these systems: they can be unpredictably interrupted, as energy harvesting by no means can ensure a predictable supply of energy. Reboots will frequently happen, which translates into a waste of resources, including energy, as applications need to re-initialize and re-acquire the state. As a result, the overall performance inevitably suffers. To allow an application to cross the boundaries of periods of energy unavailability, prior solutions, either save only a portion of program memory (avoiding the heap) [171] limiting developers to employ sophisticated programming techniques, or resort to hardware modifications by replacing SRAM with FRAM [93], that may not only impact cost but also processing speed.

This thesis aims to design software techniques for transiently-powered embedded devices, allowing an application to make progress, with a minimum possible energy spent on saving the system state and without resorting to hardware modifications. In the first part of the thesis, we present the detailed analysis of the existing energy harvesting and wireless energy transfer solutions for wireless sensor networks (WSNs) [27]. We define desirable properties, classify existing solutions, and argue about their applicability in different deployment environments. Later, we conduct a comprehensive survey of the state of the art for transiently-powered embedded systems. We discuss challenges, define goals and classify transiently-powered embedded system solutions into different categories based on the techniques they use to ensure forward progress of the application.

In the second part of the thesis, we develop three different techniques for saving system state quickly and in an energy-efficient manner, exploiting different properties of non-volatile memory. Key to their efficiency is the way the state information is organized on non-volatile memory. Our re-

sults, through extensive evaluation, crucially indicate that there is no "one-size-fits-all" solution. It is the application’s memory characteristics that will make one technique preferable over another. These evaluation results also lead us to design an additional technique, DICE, in which, instead of reading non-volatile memory to compute changes in the main memory (which is energy-hungry operation and used in previous techniques), we track changes in the main memory through just code instrumentation. This makes DICE not only further reduce the amount of data to write onto non-volatile memory to ensure forward progress of the application, but also helps existing system support complete a given workload with better energy efficiency and reduced execution latency.

Finally, we present HARVOS that decides when to save the system state by looking at the worst-case energy cost required to reach the next opportunity to save system state, depending on the program structure as represented in the control-flow graph. HARVOS allows the system to make an informed decision, at every opportunity to save system state, on whether to continue with the normal execution or save the system state. Our evaluation indicates that HARVOS allows transiently-powered embedded systems to complete a given workload with 68% fewer restarts, compared to existing literature.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Questions 3
1.3 Thesis Statement and Contribution 5
1.4 Thesis Roadmap 6

I Understanding Transiently Powered Embedded Sensing Systems **9**

2 Energy Harvesting and Wireless Energy Transfer in WSNs **11**
2.1 Introduction 12
2.2 Energy Harvesting: Overview and Desirable Properties . . . 16
2.3 Energy Harvesting → Kinetic Sources 18
2.3.1 Kinetic Energy as Vibrations 19
2.3.2 Kinetic Energy as Air or Water Flows 22
2.3.3 Kinetic Energy as Human or Animal Motion 23
2.4 Energy Harvesting → Radiant Sources 25
2.4.1 Extracting Energy from Visible Light 25
2.4.2 Extracting Energy from Radio-frequency (RF) Transmissions 28
2.5 Energy Harvesting → Thermal Sources 29
2.6 Energy Harvesting → Biochemical and Chemical Sources . 30
2.7 Energy Harvesting: Discussion 34

2.8 Wireless Energy Transfer: Overview and Desirable Properties 36
2.9 Wireless Energy Transfer → Mechanical Waves 39
2.10 Wireless Energy Transfer → Magnetic Fields 40
2.11 Wireless Energy Transfer → Electromagnetic Radiations . . 42
2.11.1 Visible Light 43
2.11.2 Microwaves or RF Transmissions 44
2.12 Wireless Energy Transfer: Discussion 46
2.13 Mapping WSN Environments to Harvesting and Transfer Techniques 49
2.13.1 Outdoor Environments 49
2.13.2 Indoor Environments 52
2.14 Research Agenda 54
2.15 Summary 57

3 Transiently-Powered Embedded Systems 59
3.1 Introduction 59
3.2 Challenges 60
3.3 Taxonomy of Transiently Powered Computing Solutions . . 62
3.3.1 Out-of-place Checkpointing 62
3.3.2 In-place Checkpointing 70
3.3.3 Non-volatile Processor 75
3.4 Summary 75

II System Support for Transiently Powered Embedded Sensing Systems 77

4 "*How?*" :Designing Checkpointing Mechanism 79
4.1 Introduction 80
4.2 Background 81
4.2.1 Target Platforms 82
4.2.2 Prior Art 83
4.3 Fundamental Operation 85
4.4 Storage Modes 87
4.4.1 Split 87
4.4.2 Heap Tracker 89
4.4.3 Copy-If-Change 91
4.5 Evaluation 91
4.5.1 Contiguous Data 94
4.5.2 Non-contiguous Data 98
4.5.3 Fragmented Data 102

4.6 Discussion 106
4.6.1 The role of memory *span* 106
4.6.2 The role of *fragmentation* 107
4.7 Outlook and Summary 108

5 "*What?*" : Differential Checkpointing **109**
5.1 Introduction 110
5.2 Background 111
5.3 Overview 114
5.4 Recording Differentials 115
5.4.1 Global Context 115
5.4.2 Call Stack 120
5.5 Implementation 122
5.5.1 Precompiler 122
5.5.2 **`record()`** 122
5.5.3 Checkpoint 123
5.6 Evaluation 123
5.6.1 Settings 124
5.6.2 Results → Update Size 126
5.6.3 Results → Smallest Energy Buffer 128
5.6.4 Results → Number of Checkpoints 129
5.6.5 Results → Execution Latency 132
5.7 Related Work 133
5.8 Summary 135

6 "*When and Where?*" : HarvOS **137**
6.1 Introduction 138
6.2 Related Work 139
6.3 Overview 140
6.3.1 Challenge 141
6.3.2 Rationale 142
6.3.3 Operation 142
6.3.4 Generalization 145
6.4 Placement Rules 146
6.4.1 Branching 146
6.4.2 Loops 148
6.4.3 Function Calls and Interrupt Handlers 151
6.5 Evaluation 152
6.5.1 Settings 153
6.5.2 Results 157

6.6 Summary 162

7 Conclusion and Future Directions **165**
7.1 Future Directions 167

Bibliography **169**

CHAPTER 1

Introduction

The increasing demand for manufacturing smaller and cheaper computing devices triggered a new trend in embedded sensing to design battery-less systems, with miniaturized energy storage capacity, in combination with the small scale energy harvesting system. However, energy harvesting system by no means can ensure a predictable supply of energy for battery-less embedded sensing devices. Their output is both highly variable and unpredictable. Subsequently, because of the variability of energy harvesting system and miniaturized energy storage capacity, these battery-less systems need to save the system state to ensure forward progress of the application across power cycles.

1.1 Motivation

The progress in micro electro-mechanical systems and micro electronics are redefining the scope of the energy constraints of portable embedded devices, such as home automation, ambient sensors, wearable/implantable medical and fitness devices, and other smartphone-connected accessories. Technologies that harvest ambient energy can integrate with these devices to refill their energy buffers. Wireless energy transfer can complement these

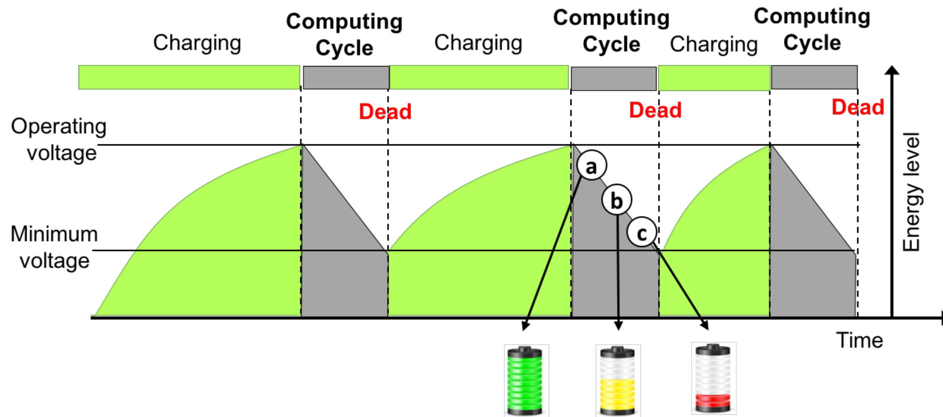


Figure 1.1: *Basic illustration of the working of transiently-powered embedded systems*

techniques by allowing users to opportunistically recharge their devices. Several techniques recently appeared that enable practical wireless energy transfer at scales suitable for the aforementioned class of devices [27]. In a few years, as much as WiFi connectivity is currently found in public areas and commercial locations, we expect citizens to find wireless recharging as well.

Energy harvesting and wireless energy transfer, however, can by no means ensure a predictable supply of energy for embedded sensing devices. Computing under such transient energy conditions becomes a challenge. Thus, such devices need to cope with highly variable, yet unpredictable energy supplies across both space and time, and be prepared to survive periods of energy unavailability. They will experience frequent shutdowns, to later reboot as soon as energy is newly available, as shown in Figure 1.1 where ⓐ, ⓑ and ⓒ represent different energy levels of energy buffer.

The problem is exacerbated as the complexity of applications grows. Many modern applications are effectively stateful [28]. In these settings, for example, whenever actuation becomes part of the application logic, actuators must retain their operating settings after a power failure to safely resume their functionality. Even in stateless implementations, the application processing deployed on embedded devices might not execute entirely on a single charge of the limited energy buffers typically employed, impeding forward progress. For example, accelerometer sensors may need to apply complex signal processing algorithms before reporting the data, which typically requires a few seconds of intense MCU utilization.

The frequent shutdown and reboot represents a waste of computing re-

sources and therefore of energy, as applications will need to re-initialize, re-acquire state, and perform re-synchronization with other nearby devices. The reboot process will take some non-negligible time; as a result, even if the application ultimately manages to make progress, the user experience will inevitably suffer.

We aim at allowing an application's processing to cross the boundaries of periods of energy unavailability, without resorting to hardware modifications, for example, replacing SRAM with FRAM [93], that may not only impact cost but also reduce the processing speed as FRAM is 3x slower than SRAM.

To that end, we must answer three questions. First is how to enable an embedded system to save the state of the program on stable storage, i.e., checkpointing, along with its later recovery with minimal latency and energy consumption. Second is how to optimize the overall system by checkpointing the minimal state needed to ensure forward progress of the application – thereby reducing the energy and time overhead of saving system state. The third is how to determine when to perform checkpointing and how to intertwine that with the main application's processing.

1.2 Questions

This thesis addresses the following questions:

1. ***How* to enable efficient checkpointing mechanism?**

   Crucially, we need to efficiently checkpoint system's state on stable storage, where it can be later retrieved to resume the computation. The whole processes should be quick and performed in an energy-savvy manner, to minimally perturb the user experience and not to affect the duration of the next computing cycle as shown in Figure 1. We need to do so without resorting to hardware modifications which otherwise may greatly impact costs, especially at scale. Additionally, the mechanism should not exclude any data segment, for example, MementOS [171] excludes heap, as the part of the checkpoint and restore. Otherwise, excluding heap will not only limit the programmer capability in terms of using dynamic data structures. Also, modern embedded network applications made the dynamic memory allocation mandatory for applications' design as their average memory requirement varies widely from one configuration to another [115].

2. ***What* to checkpoint?**

   The answer of the first question leads us to a more important question: what is the minimal state needed to checkpoint to ensure forward progress of the application? To answer this question, we need to design an effective checkpointing mechanism which can save only those memory areas of the system state which are altered compared to the previous checkpointed system state in non-volatile memory (NVM) with minimal energy and time wastage. Furthermore, the mechanism should be transparent where the developer is oblivious to the checkpointing issues and focuses only on the main coding problem. The process of detecting altered memory locations between current system state in main memory and previously checkpointed state in NVM needs to be done intelligently, otherwise it may increase the overhead instead of decreasing it.

3. ***Where and when* to trigger checkpointing mechanism?**

   Understanding *where* and *when* to trigger the checkpoint within the application code is also crucial for the overall performance of the system. As for "where", ideally one would checkpoint after every change to a variable's value, not to loose any progress in case of a shutdown, but the associated overhead would be prohibitive because some checkpointing mechanism need to probe the energy buffer, for example, through ADC operations, which not only consumes energy but also changes execution times. Therefore, frequently performing this operation may become prohibitive because of the energy cost. As for "when", doing checkpointing too early i.e. at position ⓐ or ⓑ in Figure 1, would essentially correspond to a waste of energy that could be usefully employed in further computations. In contrast, excessively postponing a checkpoint may yield a situation where insufficient energy is left to complete the operation. Because of the unpredictable supply of energy from the environment and the varying run-time execution of programs, striking an efficient trade-off is challenging.

Answers to the aforementioned questions are explored in detail in three key chapters (Chapters 4, 5, and 6)

1.3 Thesis Statement and Contribution

The focus of this doctoral research is:

Design a software-based solution, which enables transiently-powered modern wireless embedded devices to make progress across periods of energy unavailability, without resorting to any hardware modification.

To this end, our **first contribution** is to investigate different checkpointing schemes to efficiently checkpoint system's state on stable storage, where it can be later retrieved to resume the computation. This thesis describes three robust checkpointing techniques [28], i.e., *Full-Stack*, *Heap-Tracking* and *Copy-if-Change*, which ensure to checkpoint and restore a device's state on stable storage quickly and in an energy-efficient manner. Furthermore, checkpoints are restored only if they are complete, that is, the device has not shut down while taking the checkpoint. This thesis describes the trade-offs among the three schemes and two baselines taken from the literature through real experiments using two different platforms. It concludes that there is no "one-size-fits-all" solution. The performance depends on factors such as the amount of data to handle, how in memory the data is laid out, as well as an application's read/write patterns.

The lessons we learned from the *Copy-if-Change* checkpointing technique lead us to our **second contribution**: DICE (DIfferential ChEckpointing). DICE is a set of techniques to evaluate differentials between the previous checkpoint data in NVM and the current system state. These differentials originate from the use of programming constructs that mutate the system state, such as variable assignments and memory references. DICE automatically instruments existing code to track such changes and uses this information to refrain from writing to NVM the slices of the previous checkpoint that remained unaltered. This way, we reduce both the energy spent during and the time taken for checkpoints.

The **third contribution** is to address the problem of *where* and *when* to trigger the checkpoint within the application code. This thesis describes HARVOS [29], a code instrumentation strategy, which insert systems calls called **`trigger`**s at specific locations in the code. These triggers, based on the control-flow graph of the program and remaining energy, decide whether to perform the checkpoint before continuing the execution. HARVOS operates at compile-time with limited developer invention and adapts to varying levels of remaining energy and possible program executions at runtime. The underlying design rationale also allows the system to spare energy-intensive probing of the energy buffer, for example, through ADCs, whenever possible.

1.4 Thesis Roadmap

The remainder of this thesis is structured as follows.

- **Chapter 2** contains an extensive survey of the existing energy harvesting (EH) and wireless energy transfer (WET) techniques. The survey includes: *a*) in-depth, yet intuitive understanding of the phenomena enabling energy harvesting and wireless energy transfer; *b*) relations and complementary aspects of energy harvesting and WET; and *c*) detail discussion that maps existing energy harvesting and wireless transfer solutions to the characteristics of the target deployments.

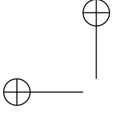
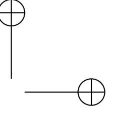
- **Chapter 3** gives an overview of the state of the art transiently-powered embedded systems. This includes solutions which: *a*) checkpoint application state on to external NVM *b*) employ NVM, especially FRAM, as the only memory space for both instructions and data; *c*) use nonvolatile processors (NVP) that relieve the system from checkpointing altogether.

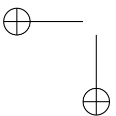
- **Chapter** 4 contains the set of three techniques to checkpoint the transiently-powered embedded system device state onto stable storage. It describes the trade-offs among the three schemes through real experiments and provides insights on what kind of application may benefit most from what schemes.

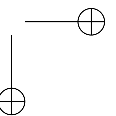
- **Chapter** 5 describes DICE, a code instrumentation strategy to automatically track changes in system state. The chapter describes the code instrumentations rules, implementation, and results based on a combination of three benchmarks across three different existing system support and two different hardware platforms. For example, using DICE, HARVOS can complete the execution of RSA algorithm with 86% fewer checkpoints and a 34% reduction in execution latency.

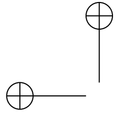
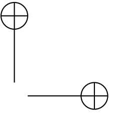
- **Chapter** 6 describes HARVOS, a code instrumentation strategy to automatically place trigger calls, which initiate checkpointing mechanism depending on remaining energy in the buffer, within programmers code. The chapter describes the design rationale, compile-time rules we apply to decide on the placement of trigger calls depending on the program structure, and experimental results. Compared to existing approaches, our evaluation indicates that HARVOS allows transiently-powered devices to complete a given workload with 68% fewer checkpoints, on average.

- **Chapter 7** concludes the thesis and presents some possible future research directions.

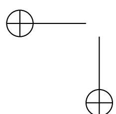
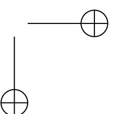


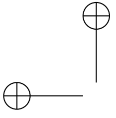
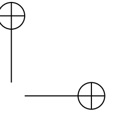


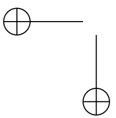



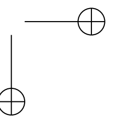
Part I

Understanding Transiently Powered Embedded Sensing Systems





CHAPTER 2

Energy Harvesting and Wireless Energy Transfer in WSNs

Advances in micro-electronics and miniaturized mechanical systems are redefining the scope and extent of the energy constraints found in battery-operated wireless sensor networks (WSNs). On one hand, ambient *energy harvesting* may prolong the systems' lifetime, or possibly enable perpetual operation. On the other hand, *wireless energy transfer* allows systems to decouple the energy sources from the sensing locations, enabling deployments previously unfeasible. As a result of applying these technologies to WSNs, the assumption of a finite energy budget is replaced with that of potentially *infinite*, yet *intermittent* energy supply, profoundly impacting the design, implementation, and operation of WSNs. This chapter discusses these aspects by surveying paradigmatic examples of existing solutions in both fields, and by reporting on real-world experiences found in the literature. The discussion is instrumental to providing a foundation for selecting the most appropriate energy harvesting or wireless transfer technology based on the application at hand. We conclude by outlining research directions originating from the fundamental change of perspective that energy harvesting and wireless transfer bring about. This chapter is published in

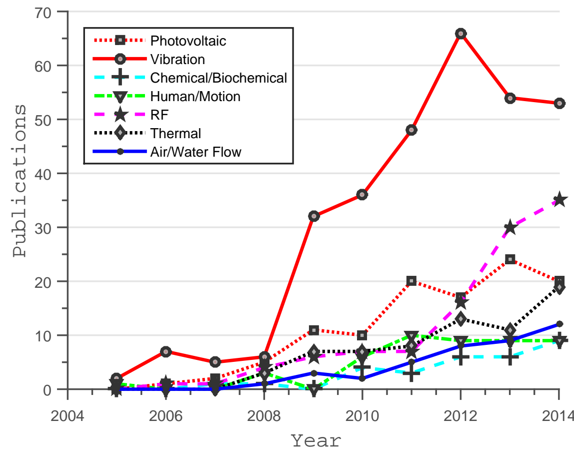


Figure 2.1: *Number of journal articles published by ACM, IEEE, or Elsevier in the last decade revolving around micro-level energy harvesting.*

extended form as *"Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences"* in [27]

2.1 Introduction

Wireless sensor networks (WSNs) have become a viable tool to gather information from the environment and to act on it non-invasively. Typical deployments employ battery-powered nodes. As a result, a whole body of work appeared to provide staple networking functionality within limited energy budgets. Simultaneously, as the form factor of WSN nodes shrunk, the battery size had to reduce as a consequence. Notwithstanding recent developments in battery fabrication, this resulted in increasingly smaller amounts of available energy.

Energy harvesting and wireless transfer To counteract this trend, recent advances in micro-electronics and miniaturized mechanical systems are finding their way in WSNs along two lines. On one hand, technologies to *harvest energy* from the ambient may integrate with WSNs to prolong the system's lifetime, or to enable perpetual operation whenever possible. A variety of techniques appeared that apply to, for example, light, vibrations, and thermal phenomena, while matching the constraints of common WSN nodes. These technologies are naturally attractive wherever replacing batteries is unfeasible or impractical, as in biomedical implants [42].

The applicability of energy harvesting remains a function of the deployment environment. Depending on its characteristics, suitable sources of

ambient energy may simply not be at disposal. Several scenarios also bear constraints on node locations that render the benefits of energy harvesting not able to pay back the additional complexity and costs. For example, WSN nodes equipped with light sensors are employed to realize adaptive lighting in road tunnels [38]. Potential sources of ambient energy are lacking in this environment, thus the opportunities for energy harvesting are minimal. However, the nodes cannot be re-arranged in space to favor energy harvesting, as their locations are dictated by application requirements.

The issues above mainly stem from the coupling between the location of energy harvesting and where sensing needs to occur. *Wireless energy transfer* (WET)—that is, the ability to move energy across space—can break such coupling, allowing WSN designers to exploit abundant energy sources available at places other than the locations of sensing. In addition, WET may distribute the available energy from energy-rich locations to energy-poor ones, improving the overall energy balance. Several techniques recently appeared that enable WET in WSN applications as well, such as laser [26], power LEDs [122], or radio transmissions [35, 165]. Notably, energy harvesting is an integral part of WET, as the latter occurs by intentionally spreading energy in the ambient, which is gained back at the receivers' end using harvesting techniques.

Energy harvesting and wireless transfer fundamentally redefine the traditional design assumptions in battery-operated WSNs: from considering a *finite* energy budget to relying on an *infinite*, yet *intermittent* energy supply. This change of perspective spurred a plethora of research works. As an example, Figure 2.1 depicts the number of articles published in the last decade in ACM, IEEE, and Elsevier journals dealing with micro-level energy harvesting, that is, the kind most directly applicable to WSNs. The figure accounts only for the lowest-level enabling technology and does not include, for example, works about network protocols in energy harvesting scenarios. The overall trend is markedly *increasing*[1].

Prior literature Because of the growing interest, a few surveys about energy harvesting and wireless transfer recently appeared. These may be broadly classified in three categories.

A few works mainly focus on integrating energy harvesting into the

[1]Figure 2.1, which is only taken as an example of the trends at stake, solely considers journals as they constitute a well-defined (sub)set of publication venues. This is certainly not comprehensive, but representative of the growing interest in the topic. Accurately identifying relevant papers at other venues, such as conferences, would likely be more difficult and arguably lead to similar considerations. We obtain the statistics by searching through the ACM Digital Library, IEEEXplore, and Elsevier's Sciencedirect based on subject categories and relevant keywords. The complete list of papers used to obtain the statistics in Figure 2.1 is available at `http://goo.gl/9Fzlve`.

node design; for example, by discussing different energy storage solutions possibly coupled with WSN nodes [101, 201]. However, low-power embedded technology is quickly evolving, and *both* complete uW-level platforms [108] and 32-bit MCUs with standby currents of a few hundred nA [104] are appearing. Thus, the state of affairs is in a situation of rapid change.

Other works discuss similar aspects for WET, and yet without explicitly linking it with energy harvesting. For example, [12] investigate the use of super capacitors as opposed to rechargeable batteries, whereas [228] describe a history of WET together with considerations on how to leverage the omni-directionality of electromagnetic radiations to power WSNs. Similar to the above, these considerations are likely to be a function of currently available embedded hardware, which is quickly evolving.

Special attention is devoted to WET through radio-frequency (RF) transmissions. [216] study the feasibility of RF-based WET, as well as the hardware design and dimensioning of systems to match exposure limits in specific application scenarios, such as smart buildings. [30] as well as [129] specifically report on systems based on the "harvest-then-transmit" paradigm, employed by RFID-scale devices to operate in a battery-less fashion. The authors discuss circuit models, signal processing techniques, and network architectures, reaching into systems that combine energy and data transfer in the same waveform.

Contribution and road-map In contrast to prior literature, we aim at providing an in-depth, yet intuitive understanding of the *phenomena* enabling energy harvesting and wireless transfer. Further, unlike works that only focus on one or a few technologies [30, 129, 216], our goal is to comprehensively cover *multiple technologies*. This allows us to elicit the relations and complementary aspects of energy harvesting and WET, while creating a foundation to analyze the trade-offs of different technologies and compare them against each other. Our investigation is instrumental to the selection of the most appropriate solution based on *application* requirements. Confronted to trends in embedded hardware, the relation of energy harvesting and wireless transfer to an application's traits is likely far less volatile. We therefore expect our analysis to enjoy long-term validity.

In this chapter we limit ourselves to the lowest-level technology that enables energy harvesting or WET. Other works that are not specific to a given energy harvesting or WET solution, such as network protocols for rechargeable WSNs or scheduling algorithms for mobile recharging stations, are often generally applicable regardless of the energy provisioning technology. We also intentionally only focus on solutions possibly applica-

Table 2.1: *Structure of the article.*

Subject	Topic	Section
Energy harvesting	Overview and desirable properties	2.2
	Kinetic sources	2.3
	Radiant sources	2.4
	Thermal sources	2.5
	Biochemical and chemical sources	2.6
	Discussion	2.7
Wireless energy transfer	Overview and desirable properties	2.8
	Mechanical waves	2.9
	Magnetic fields	2.10
	Electromagnetic radiations	2.11
	Discussion	2.12
Overarching considerations	Mapping WSN environments	2.13
	Research agenda	2.14

ble in low-power WSNs of resource-constrained devices, as they represent a significant domain with well-defined requirements. Energy harvesting and wireless transfer are applicable to other sensing platforms as well, such as personal mobile devices. These, however, exhibit different requirements compared to WSNs, and would require a markedly different conceptual framework.

Notwithstanding the specific scope of our work, the subject matter is vast. As a result, this chapter is broadly structured in three parts, as illustrated in Table 2.1:

1. **Energy harvesting:** Section 2.2 introduces energy harvesting by explaining the fundamental distinction between energy source and corresponding extraction technique, by illustrating the desirable properties that harvesting solutions must present to be employed in WSN applications, and by introducing a classification of existing solutions to guide the reader. Sections 2.3 through 2.6 focus on the individual energy harvesting solutions by illustrating the energy sources, their extraction techniques, and reported experiences of applying them to WSNs. Section 2.3 discusses kinetic energy sources, that is, the energy of motion, and ways to convert it into electrical energy. Section 2.4 illustrates radiant energy sources, which is the energy carried by electromagnetic radiations when they disperse. Energy sources due to thermodynamic gradients are described in Section 2.5, whereas Section 2.6 discusses the emerging forms of energy harvesting from biological or chemical sources. Section 2.7 wraps up the discussion by

providing a summarizing view on the application of energy harvesting in WSNs; in particular, Table 2.3 compares the different solutions along the desirable properties previously illustrated.

2. **Wireless energy transfer:** Section 2.8 introduces WET by eliciting the underlying relation to energy harvesting, and by discussing desirable properties that WET techniques should present to be fruitfully employed in WSN applications. The individual WET techniques are illustrated in Sections 2.9 to 2.11 by focusing on the different transfer mechanisms and by reporting on real-world experiences. Section 2.9 illustrates the use of mechanical waves, and especially of acoustic ones, to transport energy through the oscillation of matter, that is, in the form of kinetic energy. Magnetic fields, which may serve for WET by means of inductive coupling or inductive resonant coupling, are discussed in Section 2.10. The use of electromagnetic waves such as visible light, microwaves, and RF transmissions to transfer energy over space is described in Section 2.11. Section 2.12 concludes the treatment by offering a high-level view on the use of WET in WSN applications; in particular, Table 2.5 confronts different solutions along the properties introduced earlier.

3. **Overarching considerations:** Section 2.13 maps existing energy harvesting and wireless transfer solutions back to the characteristics of the target deployments, distilling a set of general guidelines to gauge the most appropriate technology. Section 2.14 describes open problems generated by the change of perspective brought by energy harvesting and wireless transfer, illustrating avenues for future research in areas as diverse as hardware design and environment models.

We end this chapter in Section 6.6 with brief concluding remarks.

2.2 Energy Harvesting: Overview and Desirable Properties

Research in WSNs mainly concentrated on realizing advanced functionality within finite energy budgets, dictated by the battery capacities. Because of this, existing works mainly focus on finding optimal lifetime-performance trade-offs at all layers of the stack. Harvesting energy from the ambient may potentially resolve this conflict, as already demonstrated in a number of concrete deployments [184, 185, 215].

To reason systematically about energy harvesting solutions in WSN applications, we distinguish between the relevant *energy source* against the

corresponding *extraction techniques*. The energy source indicates what environmental phenomena one may exploit to harvest energy. Examples are kinetic energy sources such as different forms of vibrations, and radiant energy sources such as solar or artificial light. Every source bears an intrinsic content of energy that may only partly be taken out, depending on the *extraction technique*. This is the specific technical solution to convert an environmental phenomena into electric energy. As a matter of fact, energy harvesting is indeed yet another form of energy conversion.

WSNs are a specific breed of networked system, with proper characteristics dictated by application requirements, hardware/software constraints, and deployment environments. When applied to WSN applications, energy harvesting solutions should present a set of desirable properties, discussed next. These properties equally apply to a large set of deployment environments:

- **EH1: high energy density.** Sources should bear an intrinsically high energy content; because of the limited efficiency of current extraction techniques, harvesting is useful only whenever the energy density of the source can compensate it.
- **EH2: high efficiency.** To justify the added system complexity, a certain extraction technique should be able to take out the highest possible fraction of the energy density offered by a given source.
- **EH3: small form factor.** The extraction technique should operate at micro-level and the harvesting device be realizable in small form factors, ideally at most on the scale of the WSN node, not to complicate the deployments.
- **EH4: high robustness.** The harvesting equipment should be sufficiently reliable and require limited maintenance, even if exposed to stressful environmental phenomena; ideally, it should not further constrain the WSN lifetime.
- **EH5: low cost.** The harvesting equipment should be attainable at low cost, not to significantly impact the system's total cost of ownership.

Besides examining harvesting solutions based on these properties, we also consider off-the-shelf *availability* as an attribute of a particular extraction technique. This is relevant for determining whether the corresponding device can be seamlessly interfaced with a WSN device and readily employed. Differently, the design of custom integration hardware is often a function of whether an extraction technique is approximated by a voltage or a current source. The former type of source provides constant voltage as long as the current drawn from the source is within the source's capabili-

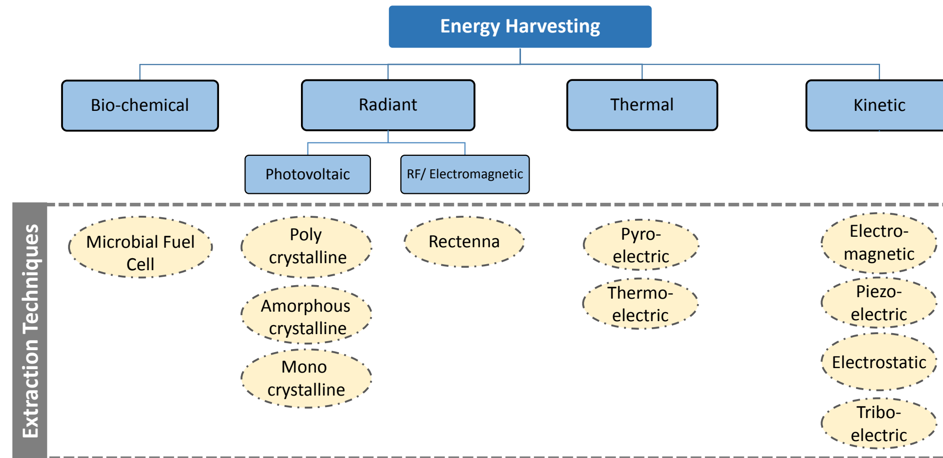


Figure 2.2: *Energy sources (rectangular blocks) and their extraction techniques (oval blocks).*

ties, and dually for the case of current sources. We analyze this aspect as well.

In the following sections, we study the nature of energy sources, their corresponding extraction techniques, along with relevant real-world WSN experiences. Our description is driven by the energy source, as it largely represents a determining factor for employing energy harvesting as a function of application requirements. Throughout the discussion, we cast the different solutions in the conceptual framework of Figure 2.2, which helps relate energy sources with the corresponding extraction techniques.

2.3 Energy Harvesting → Kinetic Sources

Kinetic energy is the energy of motion, and one of the most fundamental forces of nature. It is formally defined as the work needed to accelerate a body of a given mass from rest to a certain speed. The body gains the energy during its acceleration, and maintains this amount of energy unless its speed changes. The same amount of work is performed by the body when decelerating to a state of rest. Beyond the computing domain, leveraging kinetic energy to power various devices is an established practice. One example is that of self-winding watches, where the mainspring is wound automatically as a result of the natural motion of one's arm.

Kinetic energy may take numerous forms. In the following, we discuss popular forms of kinetic energy together with the corresponding, most commonly employed extraction techniques. These, however, should not be

intended as mutually-exclusive categories. A given form of kinetic energy may, for example, easily transform into a different one. As a result, extraction techniques employed for one form of kinetic energy are sometimes applicable when kinetic energy manifests in different ways.

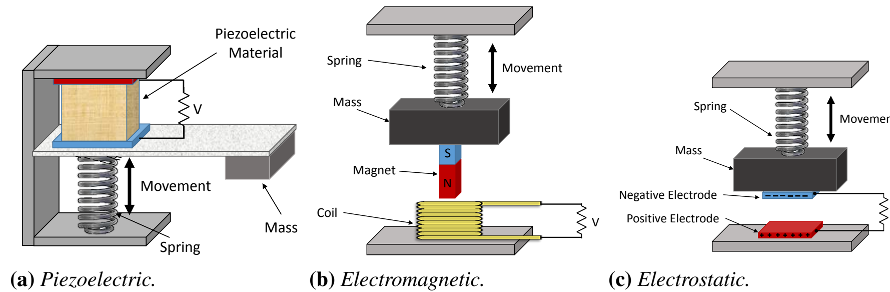
2.3.1 Kinetic Energy as Vibrations

Vibrations from manufacturing machines, mechanical stress, and sound waves are popular sources of kinetic energy. Vibration energy is typically of high density **EH1** and devices based on extraction techniques, discussed in this section, are readily available off-the-shelf at fairly low costs **EH5**. Moreover, these extraction techniques enjoy small form factors **EH3** together with useful harvesting efficiency **EH2**. As a result, these solutions are extensively explored in the WSN literature, as shown in Figure 2.1.

Extraction techniques apt to these sources and applicable to WSNs are often based on *piezoelectric* [137], *electromagnetic*, or *electrostatic* effects [43, 53]. On a conceptual level, these solutions share the fundamental mechanism to convert vibrations into electric energy. Two sub-systems are involved: a *mass-spring* system and a *mechanical-to-electrical* converter. The mass-spring system transforms vibrations into motion between two elements relative to a single axis; the mechanical-to-electrical converter transforms the relative motion into electric energy by exploiting either of the three aforementioned effects.

Piezoelectric effect Solutions exploiting this effect are based on a property of some crystals that generate an electric potential when they are twisted, distorted, or compressed [163]. Whenever a piezoelectric material is put under some external force, it causes a deformation of the internal molecular structure that shifts positive and negative charge centers. This produces a macroscopic polarization of the material directly proportional to the applied force. The resulting potential difference across the material generates an alternating current (AC), which is then converted into direct current (DC); for example, using a full-bridge diode rectifier. Although a piezoelectric harvesting device naturally resembles a current source, it may also be leveraged as a voltage source [99].

Although piezoelectric materials are not just used for harvesting energy from vibrations [156, 239], they are most often employed with a cantilever-like structure, shown in Figure 2.3a. The cantilever acts as the mass-spring system. When the beam bends because of vibrations, it creates stress on the piezoelectric film, generating alternating current. The cantilever's resonant frequency is key in determining the efficiency, and can be adjusted



(a) *Piezoelectric.* **(b)** *Electromagnetic.* **(c)** *Electrostatic.*

Figure 2.3: *Simplified models of different vibration energy harvesters.*

by changing the mass at the end of the beam and the material. Due to their mode of operation, these systems not only capture periodic ambient vibrations, but also sudden or sporadic motion.

The choice of piezoelectric material greatly influences the harvesting efficiency **EH2**. Several materials, both natural and artificial, exhibit a range of piezoelectric effects and are operational at micro-level, thus also helping achieve small form factors **EH3**. One of the most commonly used materials for piezoelectric energy harvesting is Lead Zirconate Titanate (PZT), which is however brittle in nature and susceptible to cracks upon high stress. This may negatively impact the device robustness **EH4**. Another commonly used material is Polyvinylidene Fluoride (PVDF), which is comparatively more flexible than PZT. Although PZT and PVDF are capable of generating high voltage, the output current is low due to their high impedance, thus limiting the harvesting efficiency.

Electromagnetic and electrostatic effects The electromagnetic effect is ruled by Faraday's and Lenz's laws, stating that a change in the magnetic conditions of a coils surroundings generates electromotive force. This causes voltage to be induced in the coil. To produce the change in the magnetic conditions around a coil, a magnet acts as the mass in a mass-spring system that produces movement parallel to the coils axis [107, 144], as in Figure 2.3b. This is not the only means to exploit the electromagnetic effect. For example, Samuel DeBruin et al. [54] develop a current sensor that leverages the changes in the magnetic field induced by an AC current line. This, in turn, induces an AC signal on the secondary coil, producing sufficient energy to power the sensor device.

In the case of mass-spring systems, besides the magnet's mass, its material and the coils characteristics concur to determine the efficiency **EH2** of the harvesting device. Most solutions use Neodymium Iron Boron (Nd-

FeB) for the magnet, as it provides the highest magnetic field density **EH1**. The number of turns and the material used for the coil help tune the resonant frequency according to the expected ambient vibrations. Although electromagnetic generators are more efficient than piezoelectric ones, their fabrication at micro level is difficult: the assembly is complex and care must be taken to align the magnet with the coil. These aspects negatively impact robustness **EH4**.

Differently, electrostatic transducers produce electric energy due to the relative motion of two capacitor plates, as in Figure 2.3c. When the ambient vibrations act on a variable capacitance structure, its capacitance starts oscillating between its maximum and minimum values. An increase in the capacitance decreases voltage and vice versa If the voltage is constrained, charges start flowing towards a storage device, converting the vibration energy into an electrical one. Consequently, electrostatic energy harvesters are modeled as current sources [212]. The main benefit of electrostatic transducers over piezoelectric and electromagnetic ones is the small form factor **EH3**, as they can be easily fused into a micro-fabrication process [179]. However, they require an initial charge for the capacitor.

Reported experiences Vibrations generated by vehicles are often used as sources of kinetic energy. For example, Edward Sazonov et al. [185] employ electromagnetic-based transducers to run a structural health monitoring (SHM) system for bridges, generating up to 12.5 mW through vibrations due to vehicular traffic. Similarly, K. Vijayaraghavan et al. [215] implement a traffic-flow monitoring system through a self-sustained WSN that uses vibrations generated by passing vehicles.

Vibrations from vehicles may also power on-board sensors. I.M. Tolention and M.R. Talampas [209] design a self-powered vehicular tracking system that uses vehicle vibrations to generate energy through piezoelectric harvesters. Markus Lohndorf et al. [125] evaluate the possibility of a self-sustained tire pressure monitoring system by harvesting vibrations through electrostatic harvesters. To ensure the safety of industrial machinery in trailer trucks, Daniele Dondi et al. [56] design a WSN of accelerometers and magnetometers using piezoelectric energy harvesters, powering the nodes from trailer-induced vibrations.

In other settings, Shad Roundy et al. [179] experimentally verify that vibrations generated from a small microwave oven suffice to run a WSN node. They also empirically show that vibrations from various appliances, such as cloth dryers and blender casings, suffice to run small electronic equipment.

Figure 2.4: *Different forms of vertical air flow harvester.*

2.3.2 Kinetic Energy as Air or Water Flows

Wind mills and hydroelectric turbines are among the oldest known mechanisms to extract energy from air or water flows, respectively. Their operating principles are similar to those applied for vibrations. In most cases, some form of turbine converts a flow into a rotational movement. This movement subsequently drives an electromagnetic generator. These techniques are gaining popularity, as visible in Figure 2.1 from the recent increase of works about energy harvesting from air or water flows.

The physical characteristics of the turbine—specifically the number and type of blades, and the axis of rotation—heavily influence the efficiency of the energy harvester. Whenever the air or fluid flows at high speeds, a few short and thin blades are more efficient, whereas at low speeds, numerous long and wide blades achieve better efficiency. Generators with vertical axis rotation, as shown in Figure 2.4, are popular as harvesting devices, because they do not need to be pointed accurately in the direction of the flow, as opposed to horizontal-axis devices. These often employ a tail rotor to adjust their orientation along the direction of flow.

At micro-level, energy harvesters for air or water flows use miniaturized versions of traditional spinning turbines. In addition, the devices often take advantage of drag forces causing vibrations of the device itself, through an additional piezoelectric or electromagnetic harvester [239]. Despite the wide variety of turbines available in the market, the large form factor **EH3** due to turbine components, and the limited robustness **EH4** due to degradation that also induces high maintenance costs **EH5**, limit their application to specific scenarios.

Reported experiences Yen Kheng Tan and S.K. Panda [205] design a self-sustained forest fire monitoring system that uses micro wind turbines to harvest energy and to sense wind speed. To measure the latter, they use

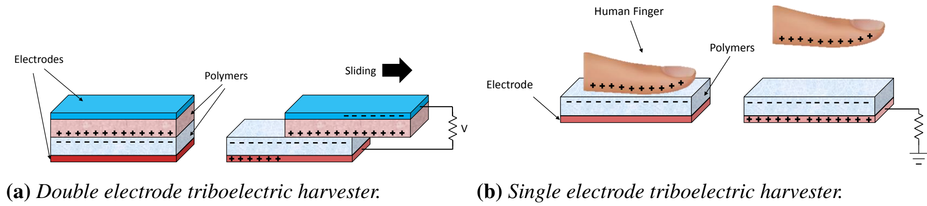
horizontal-axis turbines with a tail rotor for orienting the device along the direction of air flow. Similarly, Xuan Wu and Dong-Weon Lee [225] design a self-sustained forest fire monitoring system that solely runs on micro wind turbines, but uses a temperature sensor to trigger the fire alarm. Emilio Sardini et al. [184] design an energy harvesting wireless sensor to monitor temperature and air velocity in a building's air ducts. Sensed data is then used to automatically control HVAC systems. In general, the use of air flows as sources of kinetic energy in indoor scenarios is increasingly popular [157, 226, 239].

Besides air flows, James L Chen et al. [40] use a vertical-axis water turbine for powering WSN nodes in water pipes to monitor leakages and quality. Thad Starnes [199] proposes the idea of energy harvesting from blood pressure, later realized by Martin Detrre et al. [55] by creating a self-sustained pacemaker. This device, even if not explicitly designed for networked sensing, demonstrates the feasibility of energy harvesting from flows at extremely small scales, which may be similarly applied in WSNs.

2.3.3 Kinetic Energy as Human or Animal Motion

As mentioned earlier, kinetic energy may easily transform from one form to another. This increases the general applicability of extraction techniques commonly employed for one form to a different form of kinetic energy. An illustrative example is that of kinetic energy from human motion. Extraction techniques employed for vibrations and described in Section 2.3.1—including those based on piezoelectric, electromagnetic, and electrostatic effects—are also applicable to harvest energy from human motion. For example, Joseph A Paradiso et al. [156] design a system able to harvest energy from the push of a button through a piezoelectric material. The device then transmits a digital identifier wirelessly, which can be used to control other electronic equipment.

In addition, it is possible to harvest electric energy from movements such as footfalls or fingers' motion through the *triboelectric* effect. This is one of the most natural phenomena, which occurs when different materials come into frictive contact with each other, becoming electrically charged. Rubbing glass with fur, or passing a comb through the hairs can, for example, yield triboelectricity. Various forms of energy harvesting using the *triboelectric* effect are possible. For example, Te-Chien Hou et al. [85] show a triboelectric energy harvester embedded within a shoe able to power 30 light-emitting diodes. These devices are typically composed of two films of different polymers laid in a "sandwich" structure, as in Figure 2.5a. The



(a) *Double electrode triboelectric harvester.* (b) *Single electrode triboelectric harvester.*

Figure 2.5: *Two modes of energy harvesting through the triboelectric effect.*

charge is generated by the rubbing of the two polymer films when they slide against each other, then captured by two electrodes.

To capture triboelectric energy generated by the touch of human skin, a single-electrode device is used, as in Figure 2.5b. When a positively charged surface, such as a human finger, comes in contact with the polymer, it induces a negative charge on it. The overall circuit remains neutral as long as the two surfaces remain in contact, producing zero voltage on the load, as indicated on the left-hand side of Figure 2.5b. However, as the positively charged surface moves away, as shown on the right-hand side of Figure 2.5b, the polymer induces a positive charge on the electrode to compensate the overall charge. This makes the free electrons flow from the polymer towards the electrode, and then to ground producing output voltage on the load. Once the polymer comes back to the original state, the voltage drops back to zero.

The use of polymers in both these forms of triboelectric devices makes the structure flexible, improving robustness **EH4**, and allows one to achieve small form factors **EH3**. The latter feature, for example, facilitates the integration into wearable WSNs, where the skin rubbing against the harvesting device triggers the triboelectric effect. However, the scenarios where these technologies may be applied are vastly different, requiring significant customizations, as a solution is difficult to obtain off-the-shelf.

Reported experiences Extracting energy from human motion is a natural choice in a number of scenarios, especially within the health-care domain. As an example, there exist implantable sensors able to harvest energy from muscular movements to transmit useful information about prosthetic implants [14, 192]. Besides the aforementioned work by [156], [67] also use piezoelectric energy harvesters embedded within shoes to power active RFID tags for indoor localization. The range of application is not limited to scenarios involving humans. Nelson I. Dopico et al [58], for example, demonstrate a herd localization system powered solely by harvesting energy from cattle movement.

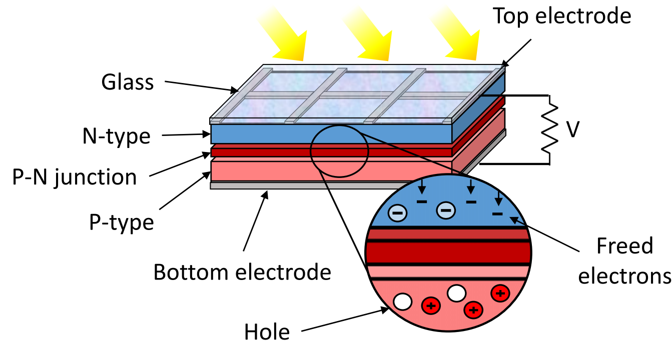


Figure 2.6: *Simplified model of a photovoltaic cell.*

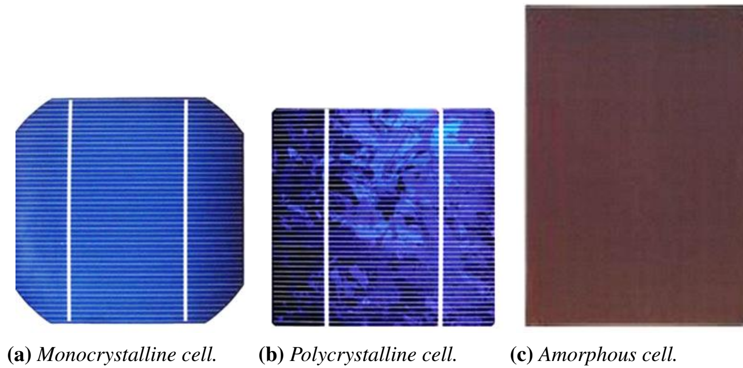
2.4 Energy Harvesting → Radiant Sources

Radiant energy is the energy carried by electromagnetic radiations when they disperse from a source to the surrounding environment. The most common form of radiant energy is, of course, solar light. However, light radiation may or may not be visible. Besides light, a source of radiant energy that recently received increased attention are pre-existing radio-frequency (RF) transmissions.

2.4.1 Extracting Energy from Visible Light

The *photoelectric* effect allows one to extract energy from electromagnetic radiations below the infrared spectrum, with solar (visible) light as the primary example. Photovoltaic cells, leveraging the photoelectric effect, are one of the most mature energy harvesting technologies. As Figure 2.6 depicts, a photovoltaic cell—also known as solar cell—consists of a minimum of two layers of semi-conducting material, mostly silicon. One of the layers, called N-type layer, is doped with impurities to increase the concentrations of electrons. Likewise, freely moving positive charges called *holes* are introduced by doping the silicon of the other layer, thus called P-type layer. Together, P-type and N-type layers form so called P-N junctions.

When light hits the N-type layer, due to the photoelectric effect, the material absorbs the photons and thus releases the free electrons. The electrons travel through the P-N junction towards the P-type layer to fill the holes in the latter. Among the freed electrons, few of them do not find a hole in the P-type layer, and move back to the N-type layer. This occurs through an external circuit, whereby the generated current is directly proportional to the intensity of light. Thus, photovoltaic cells are generally modeled as



(a) *Monocrystalline cell.* **(b)** *Polycrystalline cell.* **(c)** *Amorphous cell.*

Figure 2.7: *Different types of solar cells.*

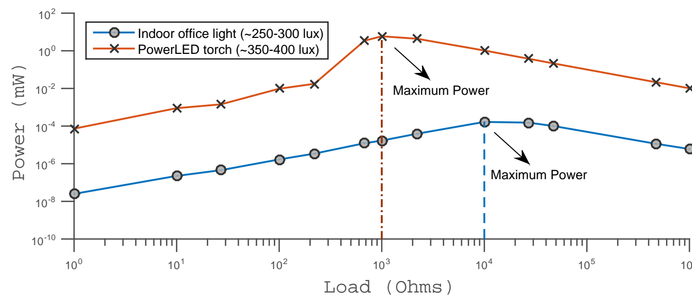


Figure 2.8: *Harvested power using amorphous silicon cells against different loads.*

current sources [99].

The material used in a cell's construction mainly determines its efficiency **EH2**, along with form factor **EH3** and cost **EH5**. Monocrystalline cells, shown in Figure 2.7a, use a single crystal of silicon, and are both most expensive and most efficient. However, they are not employed at micro level because of their cost. Polycrystalline cells, shown in Figure 2.7b, use multiple silicon crystals and can be recognized by their shattered-glass look. They are less efficient than monocrystalline cells, but their lower cost outweighs the efficiency losses. Amorphous silicon cells—also known as *thin film* cells, shown in Figure 2.7c—are manufactured by depositing vaporized silicon on a substrate of glass or metal and may be flexible, which simplifies installation. These are both the least performing and cheaper type of cell; however, under typical indoor lighting conditions, they may outperform monocrystalline ones.

A characteristic of solar cells is the relation between their efficiency against the actual load and operating temperature. Figure 2.8 illustrates

this relation for an amorphous cell. In conditions of office lighting, the cell gives maximum power with a load of 10,000Ω; when the same cell is under a PowerLED torch, it generates maximum power with a load of 1000Ω. To leverage this phenomena, solar cells usually incorporate a Maximum Power Point Tracking (MPPT) module, which tracks the output power and applies the appropriate load to optimize the output performance. In principle, MPPT modules may similarly assist the functioning of other energy harvesting techniques [232], but their use is most commonly reported with solar cells.

Due to the high energy density of solar light **EH1** and off-the-shelf availability of existing solutions at relatively low cost **EH5**, light-based energy harvesting is a popular means to power WSN nodes. The greatest advantage at micro level is the lack of moving parts, which increases robustness **EH4**, and the absence of emissions or noise. Nevertheless, the overall performance strongly depends on environmental conditions, that is, on light intensity and duration.

Reported experiences Many cases of light-based energy harvesting can be found in the literature. For example, Julian Gutierrez et al. [73] design an automated irrigation system by extending the battery lifetime of WSN nodes using solar cells. A self-sustained transmit beacon is also reported [178], which uses both light and vibration energy harvesting to achieve up to 100% duty cycle.

The need to employ solar energy harvesting at micro-level also motivates novel solutions compared to the mainstream use of solar cells. For example, Davide Brunelli et al. [32] improve the efficiency of solar cells by designing a very low power (1 mW) MPPT module for micro-level energy harvesting. Lohit Yerva et al. [231] show a sensor device able to harvest sufficient indoor-solar energy to acquire and transmit sensor readings every minute along a collection routing tree. Differently, EnHANTs [70] leverages solar radiations using organic semiconductors, enabling the use of bendable cells that facilitate deployments. The use of organic semiconductors allows the system to keep constant efficiency over different lighting levels, avoiding the need of MPPT modules. Finally, Prometheus [95] and Everlast [193] use a super-capacitor as a primary buffer to store energy; the main advantage being a reduction of the recharge cycles of the main battery, which prolongs its lifetime.

2.4.2 Extracting Energy from Radio-frequency (RF) Transmissions

Extracting energy from pre-existing radio-frequency (RF) transmissions recently received much attention, as shown in Figure 2.1, due to the increasing pervasiveness of cellular stations, FM radios, and WiFi networks.

The key element of an RF energy harvesting device is the "rectenna", that is, a special type of antenna able to convert the energy carried by electromagnetic waves directly into electrical current. A rectenna comprises a standard antenna and a rectifying circuit. The antenna captures the electromagnetic waves in the form of AC current; the rectifier performs the AC-to-DC conversion, making a rectenna resemble a voltage-controlled current source [150]. To design an efficient rectenna, different types of physical antennas, such as patch, dipole, planar, microstrip, and uniplanar antennas, may be incorporated with different types of rectifying circuits.

RF energy harvesting evidently bears a significant potential, yet it still requires evidence of practical applicability. Several efforts are undergoing towards this end; for example, studies exist to assess the feasibility of RF energy harvesting across different bands, such as those for digital TV, GSM900, GSM1800, 3G, and WiFi [162, 217]. These works report power levels in harvested energy from nW to mW depending on the distance from the nearest base station.

The conversion of RF transmissions into electrical current through the rectenna does not involve any mechanical process as compared to other techniques, and thus provides higher robustness **EH4**. Small form factors **EH3** are attainable by employing specific types of antennas, such as microstrip ones. In general, rectennas require highly customized solutions for a particular frequency band.

Reported experiences RF energy harvesting is gaining momentum in a number of WSN applications. For example, several successful attempts exist in smart-health applications for powering wearable and implantable medical sensors from pre-existing RF transmissions. Peng Cong et al. [45] design a cubic millimeter-scale battery-less wireless sensor to monitor blood pressure powered by external RF sources. Similarly, P. Anacleto et al. [15] build a cubic micrometer-scale rectenna able to harvest 1 uW, which can be used to power wireless implantable sensors.

Prototypes of RF-harvesting wireless sensors for HVAC control and building automation are also reported [109, 152]. These harvest energy from the unlicensed 2.4 GHz ISM band, that is, the one used by WiFi, Bluetooth, and other commodity wireless devices. Being one of the most crowded portions of the spectrum, harvesting energy from these frequencies

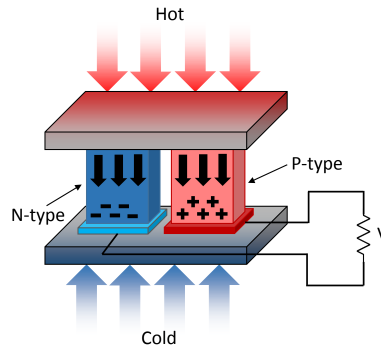


Figure 2.9: *Simplified illustration of thermoelectric effect leading to electrical current.*

is typically very effective. General-purpose WSN platforms harvesting RF energy also exist [151, 158]. These provide a stepping stone to validate the practical applicability of the related harvesting techniques.

Worth noticing is that the use of RF energy harvesting goes beyond merely powering WSN devices. For example, Vinvent Liu et al. [123], combine RF energy harvesting with backscatter communications, and create a communication primitive where devices communicate by backscattering ambient RF signals. This eliminates the need for both dedicated power sources and wires for communication.

2.5 Energy Harvesting → Thermal Sources

Thermal energy refers to the internal energy of an object in conditions of thermodynamic equilibrium, that is, in the absence of macroscopic flows of matter or energy, and in conditions of constant temperature. Whenever the conditions of thermodynamic equilibrium cease to exist, the resulting matter or energy flows become usable to harvest electric energy through *thermoelectric* or *pyroelectric* techniques.

Thermoelectric techniques are based on Seebeck's effect, which is conceptually similar to the photovoltaic techniques in Section 2.4.1. N-type and P-type materials are employed here as well, as shown in Figure 2.9. As the temperature difference between opposite segments of the materials increases, charges are driven towards the cold end. This creates a voltage difference across the base electrodes, which is proportional to the temperature difference. Thus, thermoelectric harvesters can be modeled as voltage sources [99]. Silicon wafer or aluminum oxide are typically used as a substrate material, because of their large thermal conductivity.

In contrast, pyroelectric energy harvesters leverage materials with the

ability to generate a temporary voltage as their temperature is made continuously varying, much like piezoelectric materials generate a potential when they are distorted, as discussed in Section 2.3.1. Specifically, the temperature changes cause the atoms to re-position themselves in a crystalline structure, changing the polarization of the material. This induces a voltage difference across the crystal, which gradually disappears due to leakage currents if the temperature stays constant.

Generally, the extraction techniques operating from thermal sources have no moving parts, and are therefore more robust **EH4** to environment factors compared to other micro-level harvesters. The devices may thus achieve operational lifetimes of several years without maintenance. Because of their mode of operation, it is also relatively simple to achieve small form factors **EH3**. Both thermoelectric and pyroelectric based harvesters are becoming increasingly available off-the-shelf.

Reported experiences Several works exist reporting on the use of energy harvesting from thermal sources to power WSN devices. For example, Chen Zhao et al. [237] design a self-powered WSN node that harvests energy from temperature fluctuations in the environment. Similarly, structural health monitoring systems for oil, gas, and water pipes [136, 234] leverage thermal sources such as hot water and steam. Luca Rizzon et al. [175] use heat dissipated from CPUs in data centers to run a WSNs for environment monitoring. Energy harvesters based on thermal phenomena may operate both as sensors and energy harvesters [37]; for example, by measuring temperature differences in the environment as a function of the produced energy. Also, the warmth of humans or animals may be used to power medical sensors [127], even though the applications in these areas do not appear to leverage networking functionality.

2.6 Energy Harvesting → Biochemical and Chemical Sources

Energy harvesting may leverage forms of *biological* or *chemical* energy in specific environments. Biochemical energy is the potential of a chemical substance to produce electrical energy through a chemical reaction or through the transformation of other chemical substances. Humans embody the biochemical harvesting pathway, as we power ourselves by the conversion of food into energy through biochemical processes. Differently, the electric battery is an example that uses chemical processes to convert naturally occurring chemical reactions into electricity [167].

Among biochemical extraction techniques, microbial fuel cells (MFC) use biological waste to generate electrical energy, as schematically shown

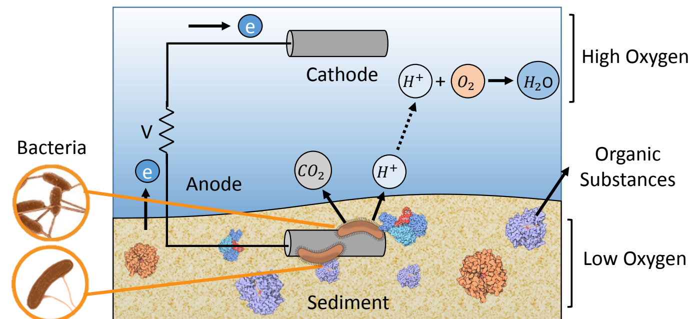


Figure 2.10: *Working of a microbial fuel cell: as part of their natural nutrition processing, bacteria remove electrons from organic material; these flow through an anode-cathode, battery-like, structure where the two compartments are separated in terms of their O_2 concentration.*

in Figure 2.10. Bacteria in water metabolize biological waste by breaking it down in a process of oxidization. This results in the creation of free electrons, along with CO_2 and H^+ ions. If the environment is deficient in oxygen, an anode picks up the free electrons and transports them to a corresponding cathode where, in an environment abundant in oxygen, the reduction process completes, yielding a water molecule.

The efficiency of an MFC is thus a function of the difference in oxygen concentration across the anode and the cathode sections. This is why, for example, MFCs deployed in marine environments have the anode partly embedded in soil and the cathode placed close to the surface, at a higher oxygen concentration, as in Figure 2.10. MFCs are thus modeled as a voltage source in series with a resistance. As energy harvesting devices, MFCs are usually robust **EH4** and require little maintenance, as long as the renewable resource, such as biological waste, is available. Recent results also indicate the potential to harvest energy from voltage differences across the xylem of a tree bark and the soil [128], using similar principles.

MFCs are usually employed to power medium-to-large scale electronic devices. For miniaturized devices, such as biomedical implants, enzymatic biofuel cells are also considered [133]. Enzymes are proteins generated by living organisms to catalyze chemical reactions. Unlike MFCs, enzymatic biofuel cells do not use living organisms to trigger the oxidation process, but only some of the enzymes that specific microorganisms produce, which are carefully extracted and purified for use. This results in higher energy efficiency **EH2** than MFCs, at the expense of higher production costs **EH5** [17]. Moreover, the power densities of enzymatic biofuel cells

vary significantly compared to MFCs, as they are typically several times smaller [36].

Extraction techniques based on chemical processes typically focus on taking advantage of corrosion phenomena; for example, those occurring on steel bars used to reinforce concrete structures. The two main factors responsible for the corrosion of steel in cement are carbonation and oxidation under the presence of water that seeps through cement pores. These elements, when reacting with iron (Fe), form new compounds like hydrated iron oxides ($Fe_2O_3\hat{A}{\cdot}nH_2O$) or iron oxide-hydroxide (FeO(OH), also known as rust), while releasing electrons. The portion of the steel that releases electrons acts as anode, whereas the portion of metal that accepts the electrons acts as cathode, with water acting as an electrolyte. The resulting flow of electrons is harvested in the form of electrical current.

Reported experiences Several WSN applications exist where the sensed phenomena is a naturally occurring biochemical or chemical process. Clare et al. [174] establish the viability of powering small oceanographic sensors by demonstrating a constant output power of 0.05 W/m^2 from MFCs in a marine settings. Donovan et al. [57] design a power management system to boost the voltage of MFCs for powering wireless temperature sensors over a period of one year in natural water. Gong et al. [69] use a tiny 0.25 m footprint MFC with average output power of 44 mW/m^2 to power an acoustic modem for transmitting temperature measures over 50 days. Similarly, Kaku et al. [98] and Dai et al. [49] place MFCs in rice paddies and forests to power wireless sensors. Building upon the work by Love et al. [128], Voltree Inc. devises a custom sensor platform that harvests energy from voltage differences across tree barks and soil, demonstrating a large-scale forest fire detection system [100, 218].

As for chemical processes, corrosion phenomena often cater for an opportunity for in-situ sensing. Qiao et al. [167] design a steel corrosion monitoring system for reinforcing concrete structures where they harvest the needed energy from the corrosion process. Similar cases are reported in the design of sensing systems for waste water treatment [65, 145]. Several results indicate that the corrosion process can produce energy in the range of a few mW over a period of several hours, which is sufficient to power small electronics [154, 168, 202].

Table 2.2: *Representative WSN deployments employing energy harvesting and their key characteristics.*

Energy Source		Extraction Techniques	Application	Harvested Power	Energy Availibilty	Operation	Literature Reference
Kinetic	Vibration	Electromagnetic	Structural health monitoring	12mW	Intermittent	Intermittent	[185]
	Vibration	Piezoelectric	Safety of vehicles	13.5uW	Continuous	Continuous	[125]
	Vibration	Piezoelectric	Safety of agriculture machinery	724uW	Intermittent	Intermittent	[188]
	Airflow	Electromagnetic	Detection of forest fire	7.7mW (@ 3.6m/s)	Continuous	Continuous	[205]
	Airflow	Electromagnetic	Automatic HVAC	45mW (@ 9m/s)	Intermittent	Intermittent	[184]
	Water flow	Electromagnetic	Water pipe monitoring	18mW	Continuous	Continuous	[147]
	Human motion	Piezoelectric	Biomedical implants	1mW	Intermittent	Intermittent	[14]
	Animal motion	Electromagnetic	Herd localization	N/A	Intermittent	Intermittent	[58]
Radiant	Indoor light	Amorphous crystalline	Indoor application	180uW	Intermittent	Intermittent	[178]
	Sun light	Amorphous crystalline	Smart irrigation	240mW	Intermittent	Continuous	[73]
	Ambient RF	Rectenna	Outdoor Sensing	60uW	Continuous	Continuous	[183]
Thermal		Thermoelectric	Environment monitoring	218uW	Continuous	Continuous	[175]
		Thermoelectric	Water metering	250u	Continuous	Continuous	[37]
Bio-Chemical		Microbial fuel cell (MFC)	Precision agriculture	310.24uW	Continuous	Continuous	[161]

2.7 Energy Harvesting: Discussion

Table 2.2 illustrates a summary view on the use of energy harvesting in WSN applications, based on a set of representative examples. Key observations we draw are:

- Energy harvesting from kinetic sources is vastly employed in WSN applications. The harvesting performance varies greatly, from few uW to tens of mW, as it depends on ambient characteristics and efficiency **EH2** of the specific technique. In many cases, the harvested energy suffices to run mainstream WSN platforms, such as the TelosB node that requires less than 10 mW to achieve 10% duty cycle [26]. In several applications, the ambient can supply energy in a *continuous* manner. In these cases, harvesting techniques for kinetic sources also enable continuous operation of the WSN nodes, that is, whenever ambient energy is available, the system can run using only tiny energy buffers, helping to cut down costs **EH5**. This is contrast to the larger energy buffers, or supplementary batteries, employed whenever the ambient energy is insufficient to fully sustain the system's operation.

- Radiant sources are also popular in WSN applications; for example, when relying on solar light. The harvesting performance is in the mW range for visible light, but lowers by several orders of magnitude when harvesting from RF transmissions. This limits the application of the latter to extremely low-power settings. In both cases, the performance is strongly influenced by the size of the harvesting device, being it a solar cell or a rectenna, and by the distance from the source. The behavior of radiant sources is also more predictable as compared to others; sunlight can be forecast and RF transmissions are almost omnipresent nowadays. These aspects facilitate achieving perpetual operation using these harvesting techniques. Even when ambient energy is not continuously available, the intermittent supply of energy from solar light can be counteracted by equipping the nodes with suitably dimensioned energy buffers [73], or by adapting the duty cycle depending on the expected energy availability [34].

- The remaining cases of thermal and biochemical sources, besides being among the most natural forms of harvestable energy, are also those with the lowest reported performance in WSN applications, typically in the uW range. This might be sufficient to power individual sensors performing local data processing, but not for achieving full-fledged

Table 2.3: *Energy harvesting solutions against the desirable properties of Section 2.2. The power density ranges are taken from the the papers referenced in Table 2.2. Conversion efficiency data, whenever available, are taken from [219] and [201].*

	Type of energy harvesting						
Property	**Biochemical**	**Visible light**	**RF transmissions**	**Thermal**	**Vibrations**	**Air/water flows**	**Human/animal motion**
Power-density	Low (~300uW)	Low (180uW) to high (240mW)	Low (~100uW)	Low (218uW to 250uW)	Low (700uW) to medium (12mW)	Medium (7.7mW to 18mW)	Medium (~1mW)
Conversion efficiency	N/A	~0.1% to 15%	~33%	~0.1% to 10%	N/A	N/A	~7.5% to 11%
Form factor	Large	Small	Medium	Small	Small	Large	Small
Robustness	High	High	High	High	Low	Low	Low
Cost	Medium	Low	Medium	Low	Low	High	Low

networking functionality. Nevertheless, compared to kinetic and radiant sources, thermal and biochemical sources exhibit unique characteristics. Energy from the ambient is continuously available, and the harvesting performance is sufficient to let the system run continuously. Extraction techniques from thermal sources also enjoy high robustness **EH4** because of the lack of moving parts, whereas MFCs can power WSN applications in environments where no other energy source is at disposal, as in underwater scenarios.

Table 2.3 wraps up our discussion on energy harvesting by qualitatively indicating to what extent different harvesting techniques meet the desirable properties of Section 2.2, independent of their reported use in WSN applications. With this, we intend to foster new directions besides the trends already apparent in the literature. We can draw the following observations from the analysis:

- Highest energy density **EH1** is found in visible light and human motion. Their extraction techniques; for example, based on photoelectric and triboelectric effects, enjoy high efficiency **EH2** and low cost

EH5. As all other kinetic sources, however, the latter has moving parts, which is detrimental to robustness **EH4**.

- Extraction techniques from thermal sources and mechanical vibrations exhibit the smallest form factors **EH3** among available solutions. In the latter case, the process of miniaturization tends to negatively affect robustness **req**EH4.
- More than a single technique is available at a reasonable cost **EH5** compared to mainstream WSN technology, with the only exception of harvesting devices from air/water flows, which tend to be expensive.

The next section focuses on wireless energy transfer (WET), its relation to energy harvesting, and its use in WSN applications.

2.8 Wireless Energy Transfer: Overview and Desirable Properties

Energy harvesting is attractive to prolong the WSN lifetime and to possibly enable perpetual operation. However, harvesting is only possible if the system is deployed where a sufficiently high-density energy source is available. In some deployments, this is simply not the case. In other settings, the availability of energy sources may be inconsistent across the deployment area, creating an energy imbalance.

Figure 2.11: *WSN deployment on the Matterhorn cliffs: node C faces north-west and is rarely exposed to the sun.*

Figure 2.11 shows a real-world example of the latter situation, taken from the WSN deployment at the Matterhorn mountain complex [77]. Node C, installed on the north-west side of the mountain, is not exposed to sunlight as often as nodes A and B. Applying energy harvesting from light is thus most effective only for node A and B, whereas node C would constantly enjoy a smaller energy contribution from the harvesting device.

Recent advancements in wireless energy transfer (WET), that is, the ability to wirelessly move energy in space, can decouple the sensing location from where energy harvesting is most efficiently applied. For example, WET can transport harvested energy to locations where ambient energy is scarce, as in the case of Figure 2.11. Moreover, WET can balance energy provisioning within a WSN characterized by non-uniform workloads,

taking from energy-rich nodes and giving to energy-poor ones. Without applying WET, these situations may result in a non-functional system even when the globally-available energy would be sufficient.

Most WET techniques include two components: *i)* a *transfer mechanism*, that is, the technical solution that allows the system to move energy across space wirelessly, and *ii)* a corresponding *harvesting technique* used at the destination to gain back the energy. Therefore, energy harvesting is one, but not the only, functional component of WET. In principle, applying WET to a certain location is analogous to *artificially provisioning* an environment with *harvestable ambient energy* at that location.

Figure 2.12 shows the relation between transfer mechanism and harvesting technique, and may be taken as a reference throughout the coming material. The harvesting techniques are largely the same as those discussed earlier. In the following section, we therefore concentrate on the other functional component of WET, that is, on the transfer mechanism. Separating the treatment of energy harvesting and transfer mechanisms is instrumental to elicit the complementary aspects between the two.

Similar to Section 2.2, we identify the desirable properties that WET techniques should present. These are intended in addition to those already discussed for energy harvesting, in that the individual WET techniques necessarily include a harvesting component. Some properties might not be as relevant for WET as for energy harvesting. For example, compared to the cost **EH5** and availability of different harvesting solutions, wireless energy transmitters such as antennas, torches, and lasers, are known to be commercially available at fairly low prices. Different than Section 2.2, the following properties may take *strikingly different* forms depending on the target deployment scenario:

- **WET1: high efficiency.** To be effective, a given WET technique should maintain the highest possible ratio between the energy harvested at the receiver and the one emitted by the source.
- **WET2: small form factor.** The harvesting part should operate at micro-level; the same requirement is less stringent for the transmitter part, in that the energy source is not necessarily integrated with a WSN node.
- **WET3: long range.** The operational distance of WET without considerable losses should minimally impact the deployment configuration, most often dictated by application or networking requirements.
- **WET4: high permeability.** It defines the ability to travel through

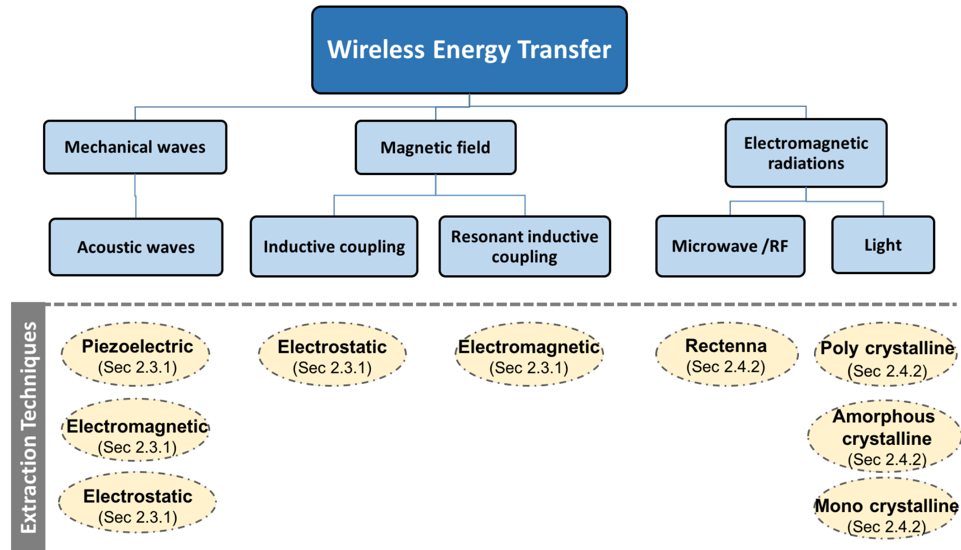


Figure 2.12: *Existing transfer mechanisms and corresponding harvesting techniques.*

obstacles of different types; certain technologies can easily traverse certain materials, such as air or water, but need line of sight or exhibit drastic losses w.r.t. other materials.

- **WET5: safety.** WET is about spreading energy in the ambient; the possibility of harming objects or persons is thus a concern, and care must be taken to determine whether a certain technique may be unsafe within its operational range.

- **WET6: routability.** It indicates technologies known to feature efficient ways to route energy across multiple hops; whereas simply harvesting received energy and using this to act as a further source is likely too inefficient.

An additional aspect is that of directionality. Existing solutions may operate in a directional fashion—with greater efficiency and range—as opposed to a omni-directional mode, which facilitates deployments by removing the need to orient the transmitters.

Next, we survey WET techniques that find application in WSNs. The discussion is driven by the transfer mechanism, as it largely determines the applicability in WSNs. The illustrative pictures shown in the next sections are *to scale* compared to each other, which enables a qualitative comparison of operational ranges.

2.9 Wireless Energy Transfer → Mechanical Waves

Mechanical waves propagate as an oscillation of matter. Such oscillation transfers kinetic energy through a medium, such as air or water. Out of the existing forms of mechanical waves, acoustic waves, that is, waves that propagate because of displacement and pressure changes of the medium, are by far the most explored for WET. Acoustic waves cause a vibrational movement onto the receiving elements. This enables the use of vibrational harvesting, discussed in Section 2.3.1, to gain back the energy. Other kinds of mechanical waves that require more sophisticated harvesting mechanisms, such as surface waves, are comparatively less explored. This represents, in fact, a consequence of the complementary aspects between the transfer mechanism used in WET and the corresponding energy harvesting technique.

As shown in Figure 2.13, acoustic waves are compressional, that is, the displacement of the medium is parallel to the direction of travel of the wave. Thus, their efficiency **WET1**, range **WET3**, and permeability **WET4** are greatly affected by the medium itself. Acoustic waves travel best through solids, then liquids, and show the most resistance through air. For example, acoustic waves are shown to transfer energy through metal walls 5 cm to 8 cm thick with efficiency of 50% to 89% [23, 191]. The efficiency in water varies from 15% to 40%; for example, Ozeri et al. [155] achieve 39% efficiency in water at a distance of 50 mm. Whenever waves travel through air, the efficiency drops to 17% at 100 mm distance [176]. We also observe ultrasound-based solutions [213] appearing on the market to charge personal devices through the air.

The propagation medium, however, is often not under the control of the WSN designers. To improve the performance in given settings, multiple vibration harvesters may be installed at the receiver's end, or transmitters may be arranged in a phased array configuration, which helps travel though objects, as shown in Figure 2.13. Acoustic waves also enjoy slower propagation speed than other kinds of waves, such as electromagnetic ones. This makes it possible to achieve more compact designs of the transmitters as well as greater efficiency **WET1** for the transmission electronics at the operational frequencies [177]. Finally, in settings where the transmission medium is animal tissue or human skin, acoustic waves naturally provide better safety **WET5** than other techniques, such as lasers.

Reported experiences A paradigmatic example is that of body sensor networks [18, 39], where safety becomes a key asset for cm- to mm-sized bio-implants. In settings where WET using electromagnetic waves is not pos-

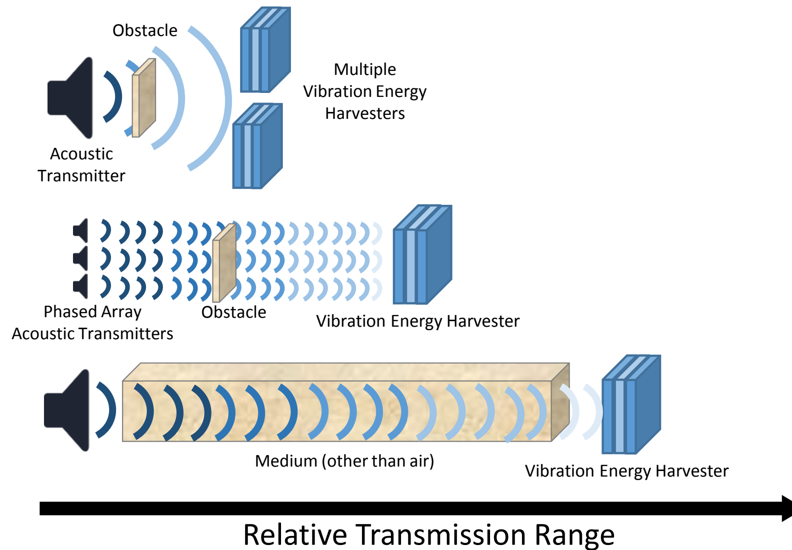


Figure 2.13: *Acoustic waves are compressional; therefore, the efficiency, range, and permeability depend on the medium of propagation. Given a specific medium, better performance is attainable by employing multiple harvesting devices at the receiver's end or by using phased arrays of transmitters.*

sible or permissible, acoustic waves are often used. Examples are deployments where acoustic waves are used to power sensors embedded in metals walls, water pipelines, and gas chambers [71,86,103,191]. Because of their permeability **WET4** properties, acoustic waves are also employed to power electronic equipment in sealed environments, such as nuclear storage facilities, where the sensors leverage piezoelectric extraction techniques to harvest the energy [87].

2.10 Wireless Energy Transfer → Magnetic Fields

WET techniques using magnetic fields mainly leverage energy harvesting based on electromagnetic effects. In Section 2.7, we already noted how the latter are vastly employed in WSN applications and enjoy good properties. Coupled with this kind of energy harvesting, two transfer mechanisms are primarily used: inductive coupling and resonant inductive coupling.

Inductive coupling is a near field—in the cm scale—WET technology that exploits two magnetically coupled coils. When alternate current is applied to the transmitter coil, this changes the magnetic field of the receiver coil, generating a potential. The mechanism is similar to the electromag-

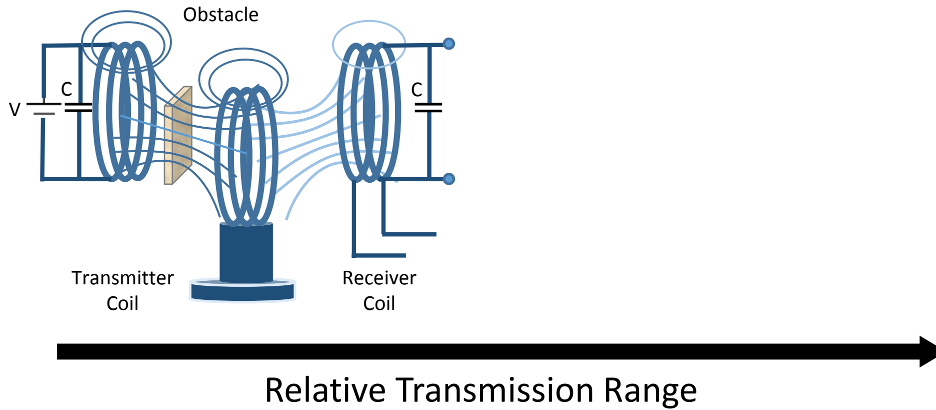


Figure 2.14: *Inductive resonant coupling allows medium-range energy transfers at high efficiency, and adding relay coils can further extend the range.*

netic extraction mechanisms of Section 2.3.1, except here we use a coil instead of a magnet to disturb the magnetic field. The efficiency **WET1** of such a system is determined by the coupling of the coils, their distance, and alignment. Over longer distances, the vast majority of the energy is lost because of resistive losses of the transmitter coil. This technique is thus solely suitable for WSN applications where the necessary operational range **WET3** is not significant.

Inductive resonant coupling, on the other hand, adds a capacitance to each coil, thus forming a tuned LC circuit, as shown in Figure 2.14. When both coils resonate at a common frequency, it is possible to attain high efficiency **WET1** over a range a few times greater than the coil's diameter. Kurs et al. [110] demonstrate this effect by powering a 60 W light bulb with approximately 40% efficiency over a 2 m distance. Recent results also make this technique more flexible in terms of reciprocal orientation of the coils. For example, Sample et al. [181] use adaptive tuning techniques to transfer energy at 2 m with high efficiency, even when coils are not properly aligned. The same system is used to run an artificial heart [223]. Sample et al. [182] show that a constant efficiency of 75% is attainable within a 65° coil rotation angle, using continuous frequency tracking and tuning of the resonant coil structure.

Inductive resonant coupling also caters for non-radiative transmission of energy, which entails the propagation of energy is not subject to phenomena such as absorption and scattering. To this end, the magnetic field is created all around the transmitter coil, with the orientation of the re-

ceiver coil impacting the efficiency. Being non-radiative, inductive resonant coupling dissipates little energy off to resonant objects [110], preventing spreading losses typical of acoustic waves. This techniques is also permeable **WET4** through metals and tissues, and remains safe **WET5** if low levels—in the range of mW to a few W—of power is transferred. Zhong et al. [238] show inductive resonant coupling to support routing **WET6** using straight-line, curved, circular, and Y-shape intermediate coils, as in Figure 2.14. Several commercial solutions based on resonant inductive coupling appeared [164, 169, 224], improving off-the-shelf availability of this technology.

Reported experiences As already mentioned, WET using inductive coupling is applicable only at very short ranges, and thus we only find experiences of resonant inductive coupling in WSNs. Nonetheless, the use of the latter technique in WSNs is arguably in its infancy, so not many works exist that report such experiences.

For example, Jonal et al. [96] use resonant inductive coupling to power sensors embedded in concrete at 40 cm depth with 5.3% efficiency. Xie et al. [227] design a WSN powered with resonant inductive coupling through a wireless charging vehicle, aiming at the optimization of the traveling path and charging times. Hancke et al. [74] demonstrate powering a MicaZ node with just a 3.2 cm diameter coil, at 10% duty cycle and a distance of 80 cm. Finally, Sample et al. [181] use resonant inductive coupling to power an underwater WSN at a depth of 1000 m.

2.11 Wireless Energy Transfer → Electromagnetic Radiations

Electromagnetic radiations are a form of radiant energy released by a certain electromagnetic process. Visible light is a common type of electromagnetic radiation; other forms are instead invisible to the human eye, such as X-rays and radio waves.

Electromagnetic radiations are distinguished based on their frequency. Such distinction also impacts the techniques to gain back the energy the radiation carries, further underlining the complementary aspects of the transfer mechanism and of the corresponding energy harvesting technique. Electromagnetic radiations below the infrared spectrum—mainly visible light—pack sufficient photonic energy to use extraction techniques based on the photoelectric effect, illustrated in Section 2.4.1. We already noted, as reported in Section 2.7, how such techniques are among the most mature means for energy harvesting. Electromagnetic radiations at higher frequen-

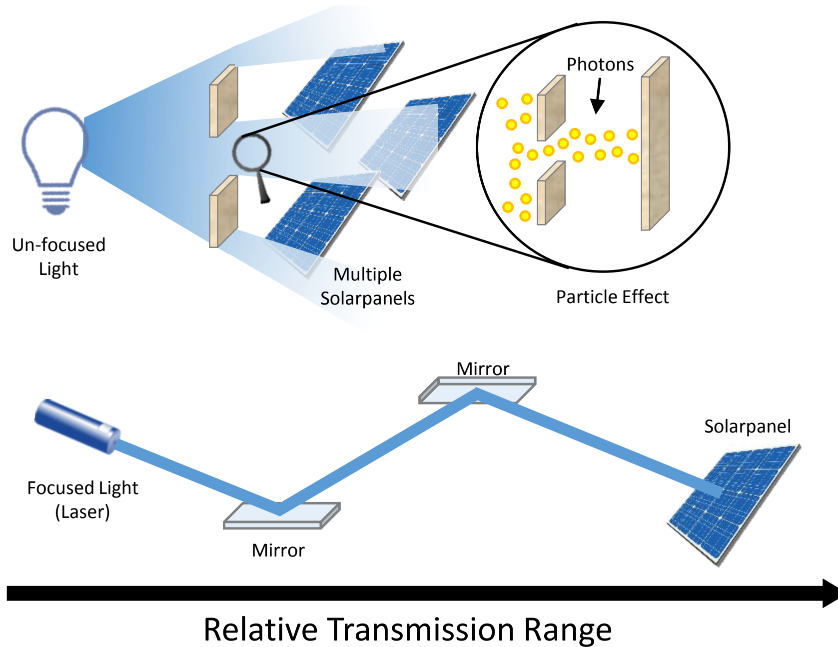


Figure 2.15: *WET using visible light. Incoherent sources suffer spreading and frequency losses, but also ameliorate the alignment requirements. Focused sources, such as lasers, require careful alignment, but also allow for greater efficiency and range.*

cies, such as microwaves and RF transmissions[2], require using rectennas, discussed in Section 2.4.2. Still based on the discussion in Section 2.7, these are comparatively less developed than energy harvesting using the photoelectric effect.

2.11.1 Visible Light

It is possible to artificially generate visible light radiations and direct them towards a photovoltaic surface to enable energy transfer at a distance. Within the visible spectrum, there exist two means to transfer energy, as shown in Figure 2.15.

One possibility is to use the light diffused from sources such as incandescent bulbs or LEDs. This type of radiation tends to be incoherent and unfocused, thus creating spreading losses due to the light beam diverging as it travels across space. The produced light also consists of several different frequencies, thus spreading the energy across the frequency domain. Thus, this approach tends to be applicable only in the short range, due to

[2]Microwaves represent the 300 MHz to 300 GHz band of the electromagnetic spectrum, while RF transmissions represents the 3 MHz to 300 MHz band.

quadratic-to-distance spreading losses and because the receiving cells often tend to be efficient only for a subset of the involved frequencies. As an example, Bhatti et al. [26] show that a 1W LED can transfer a few mW, but only at very short distances.

Laser light is, on the other hand, a stimulated emission of photons that results in a coherent and focused beam of light energy. These characteristics allow a laser beam to bear minimal losses due to spreading across space, enabling efficient energy transfer at very long ranges **WET3**, even hundreds of Km. Since the emitted light is limited to a narrow band of frequencies, the efficiency mainly rests upon the choice of the correct photovoltaic material corresponding to the used frequency bands.

Both forms of light radiation can be artificially generated from a power source, or obtained from sunlight. For example, Syed et al. [204] use mirrors to route **WET6** sunlight to locations under shade, where energy harvesting using photovoltaic techniques occurs. Sunlight can be used as the lasing source in solar-pumped lasers, sparing an artificial power source [119]. Visible light, being not permeable to most materials, requires clear line of sight. Moreover, while normal light is generally safe **WET5**, laser light may pose safety hazards, even at low-power rating, due to its focused nature that results in higher energy density.

Reported experiences Visible light is vastly explored as a WET technique to power WSN nodes. For example, reflection systems exist that use mirrors to balance the spread of sunlight in solar-harvesting sensor networks [122, 204]. LAMP [26] shows a practical laser-based long-range solution for WSNs and develops a system architecture able to power a TelosB node with 10% duty cycle at a distance of 100 m using a 0.8 W laser. Wang et al. [221] use laser light to simultaneously power multiple nodes by diffusing the light through phosphorus surfaces, eliminating the need to accurately localize the receiver nodes. Using a 3 W laser source, they are able to harvest 85 uW at a distance of 1.5 m.

2.11.2 Microwaves or RF Transmissions

Electromagnetic radiations at frequencies higher than visible light carry energy that can be gained back using a rectenna, described in Section 2.4.2. This kind of WET is extensively studied for long-distance energy transfers—in the order of several Km—and for high-power systems. Existing literature reports the use of these techniques to power airplanes [186], helicopters [31], and even satellites [138]. These techniques, however, also require line of sight because of the low permeability, as well as a track-

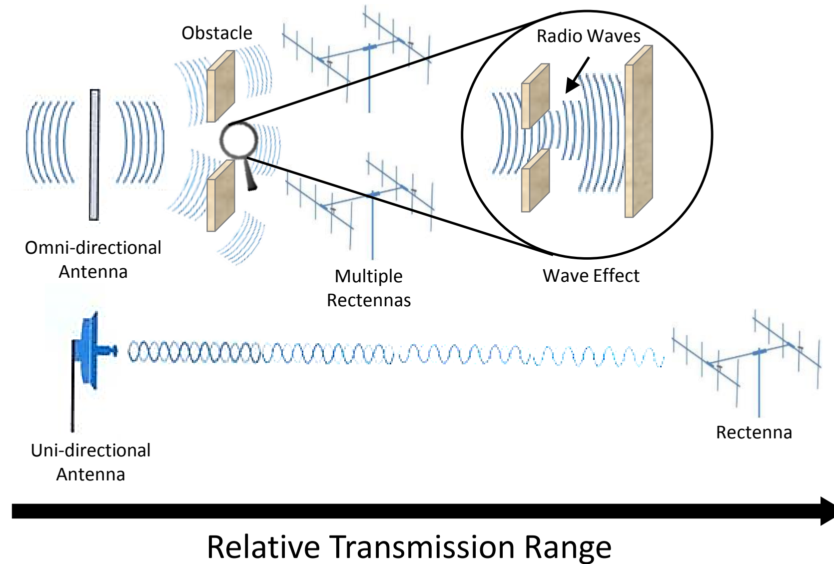


Figure 2.16: *WET using microwaves and RF transmissions. Large antennas are typically required, hampering their application to WSN-size devices. Omni-directional antennas pose less stringent line-of-sight requirements, but are less efficient because of spreading losses.*

ing mechanism to localize the receiver, due to directional energy transfer towards a moving target.

As shown in Figure 2.16, using an omni-directional antenna, such requirements become less stringent than for laser power beaming and the transfer mechanism is safer **WET5**, at the cost of higher spreading losses similar to unfocused sources of visible light. Ensuring safe operation of these systems is an actively researched topic. For example, algorithms exist to interrupt transmitters to avert dangerous continuous radiations [47], or to adjust their transmission power [48] to avoid safety hazards while retaining an efficient energy transfer.

Because of longer wavelengths, a large antenna is also normally required at both the source's and the receiver's end, in principle hampering the application to WSNs because of large form factors. Several efforts exist to miniaturize the rectenna design [146, 196, 197, 203]. The problem, however, is still open, in that no established solution is available. The size restriction on the receiver antenna, which should be on par with the size of the WSN device, still represents a limiting factor. For example, Bhatti et al. [26] show that a commercial 3W RF power transmission system [165] can generate mW of power only at a few cm.

Reported experiences Computational RFIDs [72], together with the recent push for highly miniaturized uW-level WSN platforms [108] point to a class of systems that may benefit from WET through microwaves and RF transmissions. Powercast chips [165] and the WISP node [195] represent concrete platforms available to evaluate the limits and applications of RF energy transfer in WSNs [211]. As an example, Peng et al. [160] build a prototype system with a Powercast transmitter on top of a mobile robot, and equip the WSN devices with a wireless energy receiver. An energy station monitors the energy level of the WSN nodes and routes the robot accordingly. Similarly, Tong et al. [211], Li et al. [118], and Dai et al. [46] also study deployment configurations, scheduling, and near-optimal routing of a mobile node to wirelessly recharge WSN nodes using a Powercast transmitter. Mishra et al. [143] show that multi-hopping is also possible when employing WET with RF transmissions, which may increase the operational range.

2.12 Wireless Energy Transfer: Discussion

To provide a summarizing view of how WET is employed in WSN applications, Table 2.4 shows representative efforts in this area, along with their key features. Based on these, we draw the following observations:

- WET using acoustic waves, due to its high permeability **WET4**, is often employed to power sensors in sealed or inaccessible environments, such as bio-medical implants and areas behind metal walls. The reported performance is in the mW scale with efficiency **WET1** from 21% up to 54%. Compared to the general state of the art, these figures represent a best-case performance in real-world deployments. In contrast, the practical feasibility of WET using acoustic waves in WSN applications is limited by its operational range **WET3**, which rarely exceeds a few cm.
- Magnetic fields as a means to implement WET also enjoy high permeability **WET4**, yet through different materials than acoustic waves, enabling WSN deployments underground or inside concrete structures. Compared to WET using acoustic waves, however, the attainable efficiency **WET1** is one to several orders of magnitude lower, while the operational range **WET3** is distinctively larger only when transferring through air. In the few cases where performance figures are indicated, mW scale energy transfers are reported across tens of cm.

Table 2.4: *Representative WSN deployments employing WET and their key characteristics.*

Technologies		Setting	Performance	Efficiency	Range	Literature reference
Acoustic waves		Biomedical	100 W	54%	3 cm (Skin)	[39]
		Biomedical	62.5 mW	21-35%	105 mm (Skin)	[18]
		Through metal wall	30 mW	–	7 mm (Aluminum)	[103]
		Through metal wall	0.25 W	–	5.7 cm (Steel)	[191]
Magnetic fields	Resonant inductive coupling	Through concrete	–	5.3%	40 cm (Concrete)	[96]
		Underground sensors	–	0.8%	5 m (Air)	[74]
Electro- -magnetic radiations	Microwave /RF transmissions	Through air	45 mW	1.5%	10 cm (Air)	[160]
	Visible light	Through air	7 mW	–	100 m (Air)	[26]
		Through air	85 uW	–	1.5 m (Air)	[221]

Table 2.5: *WET against the desirable properties of Section 2.2.*

	WET technique				
Efficiency	**Acoustic**	**Inductive coupling**	**Resonant inductive coupling**	**Microwaves/ RF transmissions**	**Light**
Performance	Medium	Low	High	Medium	High
Form factor	Medium	Medium	Medium	Big	Small
Range	Medium	Medium	Low	High	High
Permeability	High	Low	High	Low	Low
Safety	Medium	High	High	Medium	High (diffused)/ Low (lasers)
Routability	No	No	Yes	Yes	Yes

- WET through electromagnetic radiations shows different performance when using visible light or microwaves/RF transmissions. Because of low permeability **WET4**, WSN applications leveraging this technique are mainly limited to transfers through air. Efficiency **WET1** is comparable to that of magnetic fields, whereas the operational range **WET3** may reach hundreds of m while keeping mW scale performance at the receiver. In WSN applications, this performance is often achieved by means of a directional beam; for example, using laser light, provided line of sight and an accurate transmitter-receiver alignment is attainable.

Table 2.5 concludes the discussion by qualitatively illustrating how different WET techniques cater for the desirable properties of Section 2.8. Again, we do this independent of their reported use in WSN applications to foster new directions regardless of already existing efforts. Key observations are:

- Most efficient **WET1** techniques appear to be magnetic fields using resonant inductive coupling and visible light. The former enjoys better permeability through solids **WET4**, whereas the latter has greater operational ranges **WET3**.
- WET using visible light may also provide small form factors **WET2** compared to all other techniques, but it may pose safety **WET5** hazards when using lasers, due to the high energy density.
- WET using electromagnetic radiations is, in principle, most suited to scenarios needing long operational ranges **WET3**, also because

of the routability **WET6** properties that both visible light and microwaves/RF transmissions enjoy.

- High permeability **WET4**, in contrast, is provided by acoustic waves and resonant inductive coupling, especially through solids. Only the latter is routable **WET6**.

The following section opens the last part of the article, providing a set of overarching considerations that begin with mapping energy harvesting and wireless transfer techniques back to the characteristics of target WSN deployments.

2.13 Mapping WSN Environments to Harvesting and Transfer Techniques

The discussion thus far points out the applicability of energy harvesting and wireless transfer techniques as a function of the target environment. In light of this discussion, we give the reader a set of guidelines to gauge the most appropriate solution based on the characteristics of the deployment. To this end, we distill a set of paradigmatic WSN deployment environments and map energy harvesting and wireless transfer back to them. Figure 2.17 pictorially illustrates the mapping.

2.13.1 Outdoor Environments

Early literature on battery-operated WSNs included numerous reports on outdoor deployments with little infrastructure support [61,62,194,210], including remote [134], harsh [126], underwater [78], and subterranean [135, 206, 207] locations. Later, outdoor deployments in urban areas also received considerable attention [51, 66, 220]. Each of these environments features distinctive characteristics that may alter the choice of the most suitable energy harvesting or wireless transfer solution.

Outdoor environments → remote Locations that lack human infrastructure in the vicinity are, for example, forests, deserts, glaciers, and volcanoes. In such locations, solar light remains perhaps the single most likely natural source of energy, which can be extracted as explained in Section 2.4.1. Nonetheless, other forms of energy harvesting may be feasible in these locations, such as harvesting energy from air or water flows, as illustrated in Section 2.3.2, and from thermal gradients, as discussed in Section 2.5.

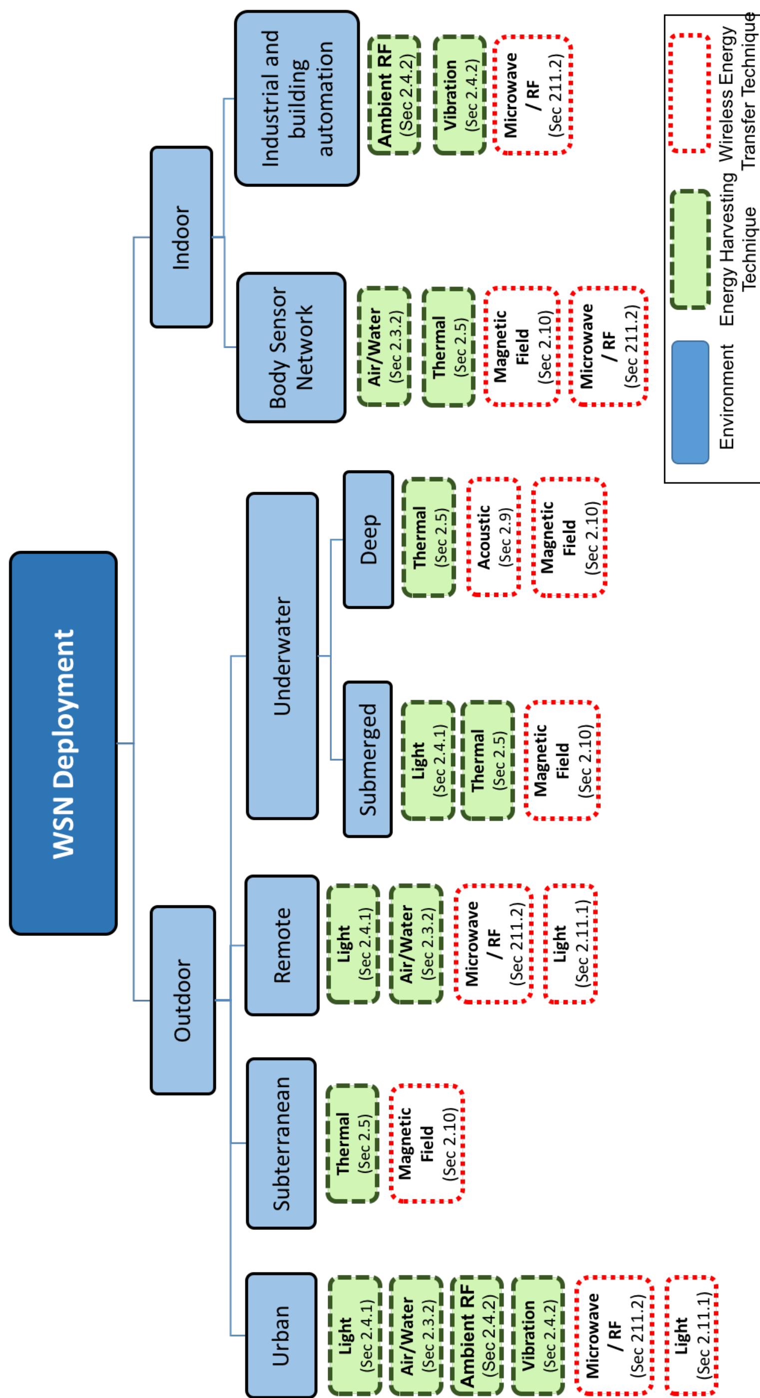


Figure 2.17: *Mapping from WSN deployment environments to energy harvesting and wireless transfer.*

WSNs deployed in remote environments may also benefit from WET. The availability of the aforementioned energy sources, indeed, can vary significantly, both temporally and spatially. Larger harvesting units can be deployed at locations with abundant harvestable energy, later re-distributed through WET. While other techniques might also be suitable in specific scenarios, the potentially large distances favor WET using lasers or microwaves, illustrated in Section 2.11.1 and 2.11.2, respectively. In the long term, it is expected that systems will be deployed catering for energy from very far; for example, if satellite stations eventually provided energy services like GPS today [190].

Outdoor environments → underwater Underwater sensor networking differs from traditional WSNs on many accounts; for example, because of the use of acoustic waves for data transfer, which results in significantly lower bandwidth and higher latencies. The need for energy harvesting and wireless transfer is natural in these environments, since battery replacement is highly undesirable or even impossible.

Submerged WSN deployments, that is, located near the water surface, include applications such as monitoring water quality in rivers or lakes, and checking for occlusions in home water pipes [37, 40, 141]. Since these deployments are close to the surface, it is natural to harvest kinetic energy from water waves, as described in Section 2.3.2. Moreover, solar energy can be harvested at the surface and transferred to the nodes below using water-permeable WET technique such as acoustic waves, illustrated in Section 2.9, or inductive resonant coupling, as explained in Section 2.10. WET through electromagnetic mechanisms, described in Section 2.11, is ruled out due to the high attenuation through water.

Deep underwater deployments observe phenomena such as sub-sea oilfields [78] and enjoy little or no direct access to ambient energy sources. In these scenarios, the only realistic—yet not entirely practical to date—solution is to deploy a large energy source, like a radioisotope thermoelectric generator, on the ocean floor that can transfer energy using one of the water-penetrating WET techniques.

Outdoor environments → subterranean WSN deployments are also reported in underground colonies of animals [135,206,207], inside mines [117], and in agricultural fields [97]. Harvesting opportunities in these locations are minimal, with the exception of vibrational and thermal sources. Vibrational sources may emerge due to movements above ground that propagate through the soil, harvested using approaches presented in Section 2.3.1. As an example, in agricultural fields, shallow subterranean deployments may benefit from the vibrations generated by regular movement of heavy ma-

chinery including seeders, harvesters, and mobile irrigation systems [97]. Thermal sources appear due to differences between soil and ambient temperatures. The extraction techniques described in Section 2.5 are usable there.

WET based on magnetic fields is arguably the most suitable mechanism to supply energy in subterranean deployments, due to permeability in soil. For example, a source above ground may employ inductive resonant coupling to transfer energy through soil at 10-40 cm of depth [140], employing the techniques illustrated in Section 2.10. The natural mobility of subterranean fauna under observation [121]; for example, whenever animals come out and go back into burrows, can be exploited to transfer bulks of energy, later redistributed to nodes inside burrows using any of the short range WET technique, such as inductive resonant coupling. Similarly, for deep underground operations, such as in mines, an energy source that we may deploy in-situ may employ the routability features of WET using magnetic fields, as described in Section 2.10.

Outdoor environments → urban These environments are characterized by an abundance of infrastructure for energy, transport, and living. Thus, they offer a host of sources for energy harvesting, being energy generated not just by nature, as in the case of solar light and wind, but also by humans and their activities, as in the case of RF transmissions, discussed in Section 2.4.2, water flows, reported in Section 2.3.2, or vehicle-induced vibrations, as in Section 2.3.1.

WET for WSNs in urban environments is, however, arguably nontrivial. Current trends indicate the possibility of eventually deploying a low-energy wireless power-grid. Cities can deploy power beaming systems; for example, using microwaves, described in Section 2.11.2, at locations that have best solar irradiation to gather energy and re-distribute it within a mesh of energy-distributing nodes. One such example is the "array of things" deployment in Chicago [208], where energy-distributing nodes deployed on top of light poles use microwaves to hand out harvested solar energy. Urban environments, nonetheless, impose strict safety **WET5** requirements, ruling out unsafe WET technologies, such as laser, in densely populated areas.

2.13.2 Indoor Environments

Market needs progressively drove the deployment of WSNs in structured indoor environments. These installations serve a multitude of purposes; for example, providing means to extend the scope and granularity of building

automation systems [233, 242], and replacing existing industrial wired sensors [76, 91]. WSN-powered medical and smart-health applications also emerged; for example, in the form of body sensor networks (BSNs) providing one's vital signs for real-time analysis [45, 236]. Similar to the outdoor case, every such environment bears distinctive characteristics.

Indoor environments → industrial and building automation Most of these environments enjoy an existing wired energy distribution infrastructure. Still, energy harvesting and wireless transfer remain desirable for a number of reasons; for example, to exploit renewable energy sources or to enable rapid prototyping.

In factory environments, vibrations due to moving machines provide an abundant source of kinetic energy. The corresponding extraction techniques, as discussed in Section 2.3.1, are both sufficiently mature and provide reasonable performance. In buildings, numerous other harvestable energy sources exist, such as air flows induced by the operation of ventilation systems and RF transmissions due to WiFi and cellular networks. Extraction techniques apt to the former are sufficiently developed, as discussed in Section 2.3.2. Differently, the state of the art in rectenna design, still battling with the need to miniaturize the devices as discussed in Section 2.4.2, is arguably not quite at a level of widespread adoption.

WET in such energy-rich environments might facilitate seamless deployment of WSN nodes. With the advent of commercial, high intensity WET solutions [60, 224], using resonant inductive coupling, discussed in Section 2.10, or RF transmissions, described in Section 2.11, one may foresee the availability of a full energy infrastructure within buildings. WSNs installed therein can thus primarily focus on the application requirements and not on energy-conservation issues, as these technologies are provisioned for multi-watt consumer appliances, such as TVs and mobile phones, and thus easily support the mW requirement of WSNs.

Indoor environments → body sensor networks (BSNs) Applications employing BSNs are often required to operate unattended for months or years, and yet they often exhibit varying degrees of mobility and severe form factor constraints. These characteristics make applying traditional battery technology very difficult. Because of this, BSNs are often coupled with energy harvesting and wireless transfer techniques.

For example, kinetic energy sources including muscle movements and blood flows, as well as thermal energy sources such as temperature differentials, are well suited to energy harvesting in such networks. The corresponding extraction techniques, described in Section 2.3 and 2.5, are able to match the form factor constraints without impacting the node mobility.

WET is also a practical solution when monitoring patients inside hospitals or at home, mainly because they are expected to be restricted in their movements, and thus remain within the range of WET mechanisms such as magnetic resonance, as described in Section 2.10, or light, as explained in Section 2.11.1. It is even conceivable that purely inductive coupling is used to transfer energy for sub-cutaneous devices, using a robotic infrastructure in a patient's vicinity [189].

2.14 Research Agenda

Battery-powered WSNs manifested new challenges out of the necessity to manage *finite* energy budgets against sensing, computation, and communication needs. Energy harvesting and wireless transfer fundamentally redefine these challenges, as the assumption of a finite energy budget is replaced with that of potentially *infinite*, yet *intermittent* energy supply. This profoundly impacts every aspect of wireless sensor networking, ultimately including the pattern of operation. This will eventually transform from the traditional sense-compute-transmit to a *harvest*-sense-compute-transmit-*share*, that is, beginning and ending with energy- and not data-related tasks.

Based on this observation, we identify four major areas worth additional research efforts. We discuss next the issues that we maintain are to be researched in every such area, providing specific directions for future work.

1. Hardware design In energy-harvesting WSNs, the input power is a function of energy availability in the environment, and thereby heavily fluctuates. Electronics employed in battery-powered WSNs, on the other hand, feature a narrow *operational power spectrum*, that is, the range of different power inputs the electronics can withstand. This makes traditional electronics ill-unsuited to energy-harvesting WSNs [229]. To make things worse, the same kind of electronics often presents high surge current requirements, possibly preventing WSN nodes from (re-)booting even if energy is available.

These considerations influence every aspect of a node's hardware design, from sensors to memory, to processors, to radios. New designs are thus required that: *i)* can cope with highly variable supplies of energy, and *ii)* reduce the gap between a node's energy requirements and the generation capacity of the harvesting unit. For example, multiple operational states of the hardware, with varying performance levels and power requirements, can be defined to stretch the operational power spectrum. Efforts are currently undergoing to achieve these objectives [108], yet the challenge is arguably

far from being fully addressed.

2. Networking energy The integration of energy harvesting in traditional WSNs does not pose significant challenges, as it requires extensions—in the form of a harvesting unit—that only impact individual nodes.

The case is different when applying WET. Energy, previously considered as a node's local commodity, now becomes a deployment-wide shareable resource. How to concretely take advantage of this conceptual leap is still quite unclear. For example, one needs to understand how to schedule energy transfers; what amount of energy to transfer; and whom to transfer energy to, which may further require accurate localization when using directional techniques. Some recent work started exploring these questions [118, 240, 241], providing early evidence of the opportunities enabled by embedding a notion of a energy distribution as a first class concept that impacts both the sensing functionality and communication stack of WSNs. Data- and energy-networking will need to be co-designed, and an *energy management stack* be integrated to serve energy distribution requests. The possible design choices are also several. For example, scheduling energy transfers is achievable with static policies like *earliest-dead first*, or based on application-specific requirements.

Because of the characteristics of current energy harvesting and wireless transfer technologies, we may also need to redefine the roles of different nodes. A homogeneous network model, which considers every node to be equally capable of energy harvesting and wireless transfer, is hardly desirable. As discussed in Section 2.8, for example, deployment constraints may create significant imbalances in the ability to harvest energy. As a result, a tiered network model is probably more appropriate, similar to RFID systems. More capable nodes will be responsible for generating energy, perhaps from a renewable ambient source, and for transferring it to less capable in-situ devices that harvest and consume it. A similar model was repeatedly advocated for data networking [61, 187]; the integration of an energy management stack is only going to make these efforts more relevant.

3. System software The ability to harvest energy from the ambient is also changing how the system software operates, including operating systems and data networking. In addition to being energy-aware, systems must be *supply-aware*, that is, able both to accommodate intermittent supplies of energy and to withstand power outages.

Operating systems must be capable of stretching an application's processing across periods of energy unavailability, letting the system *resume*, and not restart, the previously-running tasks. Efficient solutions to this

problem, however, are far from straightforward. For example, periodically checkpointing the entire system state for later resumption is likely inefficient. Strategies must be conceived to decide when to checkpoint based on the current system state and remaining running time, doing so with minimal disruption of the application's processing. Some works in the area of computational RFIDs [172] resonate with some of these considerations, yet the specific solutions are difficult to port to WSNs because of the significant degree of decentralized processing that characterizes the latter. Efficient state retention techniques for modern WSN platforms exist [28], yet it is also likely that only parts of the system state are to be checkpointed, or that state information bear time-dependent validity constraints; for example, when using sensor data to enact decisions on the environment. Developers need to be given ways to express these aspects in their programs; for example, through proper language constructs.

Challenges also exist for data networking. A better understanding of how existing protocols are influenced by periods of energy unavailability is required in the first place. It may be argued, for example, that nodes running out of energy and later resuming are analogous to nodes crashing and eventually being replaced. This argument, however, plainly depends on the time scales at hand. A more robust approach would dissect the relevant networking mechanisms, such as neighborhood management and packet forwarding in routing protocols, and possibly design ways to adapt these mechanisms to an intermittent computing pattern. *Anytime algorithms* [243]—returning a valid solution at any point in their execution—may provide a means to this end.

The OS-level and networking challenges meet when thinking of the issues possibly arising whenever the state of a node resuming after a period of energy unavailability shows *inconsistencies* against the state of nodes enjoying different energy supplies. This may create ripple effects that ultimately worsen performance, as it was already recognized in the literature in the context of software failures [41]. In principle, addressing these issues would require coordinating checkpoints on a system- or neighborhood-wide scale, making sure that nodes resume from a state that remains meaningful compared to that of other nodes. These considerations may bring back to life a whole body of work on distributed checkpointing [105], now cast in a domain of resource-constrained devices operating across multi-hop topologies.

4. Environment models and tools The need for accurate environment models is clear already in battery-operated WSNs. For example, models of wireless propagation in a given deployment may serve simulators

or pre-deployment tools that allow users to establish performance vs. cost trade-offs. Obtaining this kind of models is a challenge, as the relevant environment features are difficult to identify.

In a similar vein, obtaining models of energy propagation or availability in a given environment is also complex. As for energy propagation, considerations akin to wireless transmissions largely apply. In the case of energy harvesting, the general characteristics of an area may be known; for example, Southern California is sunny, the Pacific Northwest less so. However, available energy in a specific location varies greatly over short distances and time. A paradigmatic example is that of sun-flecks: rapidly moving solar spotlights on the forest under story [112]. Their presence or absence can greatly change solar irradiation on ground, yet their patterns are surprisingly complicated.

Confronted with these challenges, we need to conceive suitable environment models and incorporate them in concrete tools to help users understand the energy nature of the environment, enable pre-deployment simulations to understand performance vs. cost trade-offs, assist debugging at deployment time, and enable remote tuning and adaptive management of network-wide energy. Designing such tools, in turn, raises several questions; for example, how to trade accuracy vs. processing times in the simulation of energy-harvesting WET-enabled WSNs? How long do we need to observe a site to gain sufficient information for feeding the models? What level of spatial granularity is required to obtain trustful estimates for a given site?

2.15 Summary

Energy harvesting and wireless transfer are gradually finding their way in WSNs. The former can mitigate the energy constraints of traditional battery-powered WSNs, and possibly achieve the longstanding vision of perpetual deployments. However, the feasibility of a particular energy harvesting technique is deployment-specific. WET can overcome these limitations by provisioning an energy-deficient environment with abundant harvestable energy. Because of these crucial features, a plethora of research work appeared on these subjects.

In this chapter, we defined desirable properties that energy harvesting and wireless transfer techniques must present to enable their use in WSN applications, we surveyed and classified existing solutions, and argued about their applicability in different deployment environments. Although the initial upsurge in these fields is clearly visible, a lot remains to be researched

to reap maximum benefits. For example, the gap between the efficiency of exiting techniques and the energy demands of WSN nodes is to be reduced further. Moreover, WET makes energy become a network wide shareable resource, fundamentally impacting the pattern of system operation. As a result, a number of further research directions open up involving hardware design, networking energy, system software, environment models, and tools.

CHAPTER 3

Transiently-Powered Embedded Systems

In the previous chapter, we covered the challenges that arise because of energy harvesting and wireless transfer in WSNs and some of them need further investigation and improvements. One particular research problem discussed at a very high level in Section 2.14, is about the class of embedded systems that operate solely on ambient harvested energy, i.e., transiently-powered embedded systems. This chapter further explores the challenges these transiently-powered embedded systems have to overcome in actual real-world deployments and describe the range of existing solutions that address these challenges.

3.1 Introduction

Over the past decade, researchers invested considerable effort towards designing increasingly small and low-power computing devices. While there has been considerable progress in every domain, the holy grail of compact, low-cost, long-lasting batteries remains elusive. The pullulating demand in manufacturing smaller, cheaper and self-sustainable computing devices triggers a new trend to design battery-less systems with miniaturized energy storage capacity, equipped with ambient energy harvesting systems. With

the increase in the number of Internet of Things (IoT) devices, the tasks performed by these devices has also gone way beyond simple sense-compute-send kind of applications [171]. These devices nowadays are required to perform complex computations.

Transient-powered embedded systems is a field of autonomous systems in which tiny embedded devices receive power directly from the ambient energy harvester output. It not only expunges the requirement of tethered power or continuous battery maintenance, but also reduces the size, weight, cost and ecological footprint of the device.

The trouble is, energy provisioning from ambient harvesting or wireless transfer is generally erratic [27] and exhibits high spatial and temporal variation. This makes the transiently-powered embedded devices shutdown and reboots continuously. In the meantime, applications lose state, wasting most of the computation already performed, and need to restart from scratch. This represents a waste of computing resources and therefore of energy, as applications will need to re-initialize, re-acquire state, and perform re-synchronization with other nearby devices.

This chapter is a survey of various technical papers that propose solutions using different technologies for the challenges transiently-powered embedded systems face in real-world deployment. The chapter is broadly structured in two parts:

- Section 3.2 describes and discusses the challenges of designing an efficient solution for transiently-powered embedded systems.
- Section 3.3 defines the taxonomy of existing solutions and briefly summarizes each solution to guide the reader.

3.2 Challenges

Embedded systems running on harvested energy, which can either be solar, wind or vibrations [120, 139, 166] etc, normally have unpredictable power supply [27] and because of it, they face frequent shutdowns. Whenever the voltage of energy buffer drops below the operating voltage (V_{off}), MCU with volatile registers and main memory loses their content. The system has to wait till the harvested energy is sufficient for the device to work. When the device starts harvesting energy, the voltage starts to increase and when this voltage reaches a certain threshold, i.e.,V_{on}, the device starts working again.

However, the work done previously on the device needs to be performed again. With a very small energy buffer, the system will never be able to

complete a computation required by most of the modern-day applications in one power cycle. Thus these devices need to have some mechanism of saving the system state (i.e. checkpointing mechanism) across power cycles to complete the computation without losing useful work done in the previous power cycle.

Most of the solutions designed for transiently-powered embedded systems have another voltage level namely the threshold voltage, i.e., $V_{threshold}$, such that $V_{off} < V_{threshold} < V_{on}$. When the voltage reaches the threshold value, the device stops performing computations and starts saving the system state, so it can resume from that exact state in the next power cycle. The choice of voltage threshold depends on the amount of state that needs to be saved. Volatile state which needs to be saved before power failure includes general and special purpose registers (GPRs and SPRs), peripherals state, stack, heap, .data, .bss segments etc. The larger the state, the higher the voltage is required to have enough energy in energy buffer for copying the state from volatile memory to NVM. In this way, "*what*" to checkpoint (i.e size of state) becomes a prerequisite question for transiently-powered computing solutions to answer the next big question, i.e., "*when*" to checkpoint?

The amount of energy required to save the system state is directly proportional to the number of bytes of volatile state that needs to be saved to NVM. This determines the least amount of energy, which is required by the transiently powered embedded systems to retain state across power cycles. Energy is directly proportional to the voltage from the following equation.

$$E = \frac{1}{2}CV^2 \tag{3.1}$$

To insure the system state is retained across power cycles, transiently-powered embedded systems have to stop doing computations and start saving the state when it reaches the *threshold voltage*. The useful work done after the last checkpoint and power failure is always lost. Ideally, this computation/energy loss should be equal to zero and device should save state only when the remaining energy is only sufficient for checkpointing operation, not greater than that.

There is another class of transiently-powered embedded system solution, which is free from all of the above challenges, called non-volatile processors (NVPs). This kind of solution extends the non-volatility down to the level of flip-flops, making application immune to transient power failures. We will discuss more about these solutions in the next section (section 3.3).

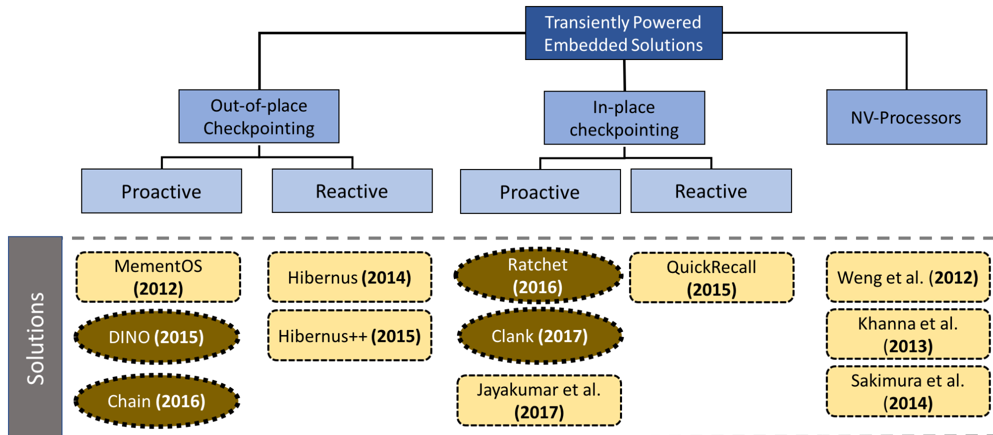


Figure 3.1: *Taxonomy of transiently-powered embedded system solutions. Solid rectangular blocks represent categories whereas dashed rectangular blocks represent solutions belonging to each category. Oval blocks with dotted lines represent solutions handling data inconsistency.*

3.3 Taxonomy of Transiently Powered Computing Solutions

Keeping in view the challenges of transiently powered embedded system solutions, researchers have proposed various solutions. Some of them require hardware modifications, and some do not require any hardware modification. Some of them explicitly use checkpointing mechanism to save state across power cycles while others define a new architecture/mechanism that does not require checkpointing and are robust against the multiple reboots. Based on this heterogeneity, we have divided the solutions into three categories as shown in Figure 3.1.

3.3.1 Out-of-place Checkpointing

We categorize all those solutions in "*out-of-place*" checkpointing that employ NVM as an external storage for saving system state. These systems target mainstream IoT architectures employing volatile main memory for efficient processing and NVM as an external storage, i.e., Flash or FRAM.

The volatile state that these systems need to save before power failure includes general/special purpose registers (GPRs and SPRs), stack, heap, .data, .bss segments etc. One can easily see that saving the entire volatile state into NVM will solve the problem but it is not going to be an optimal solution because of the following reasons:

- Saving the entire volatile state will consume energy in saving those

memory locations into NVM that contain a non-valid/empty memory locations; thus, wasting precious time/energy on useless work.

- Saving the entire new checkpoint is not always required. Computing what has changed from the previous checkpoint can create a new up-to-date checkpoint without having to copy memory locations that never changed from the previous checkpoint; thus, saving energy.

These are some of the reasons why simple approaches are inefficient for "*out-of-place*" checkpointing systems. In a nutshell, following are some goals for the "*out-of-place*" checkpointing system solutions to make them more reliable and efficient.

a. **Checkpoint size reduction:** Solutions should intelligently copy the volatile state to NVM and reduce the amount of data that need to be saved as much as possible. Energy should not be wasted in copying the data that has not been changed from the previous checkpoint or is never modified by the program. This will save the energy that can be used in actual computations.

b. **No exclusion of data**: Solutions should not exclude any important data segments that limits the functionality/flexibility of the program (i.e. heap etc), besides empty spaces and redundant data, while saving the state onto NVM. Excluding data segments from checkpointed state limits the programmer and scope of the applications.

c. **Least amount of wasted energy:** Solutions should minimize the amount of work lost from the last checkpoint, ideally making it to zero, by saving system state only when the remaining energy is strictly sufficient for checkpointing only. The amount of work done by the device from the last checkpoint to the power failure does not become part of any checkpoint. When energy buffer is depleted, this work done is lost and is re-performed by the device.

d. **Ensure data consistency**: Solutions should ensure the consistency of data after the restoration of the system state and re-execution of the code. Inconsistent data will corrupt the program execution and may produce erroneous results.

e. **Minimum user intervention**: Solutions should require only minimalistic input from the user and should work in an automated manner, i.e, configuring threshold voltage, reducing checkpoint size, placement of `trigger` calls (briefly discussed in Section 1.3) etc. Their working

should be hidden from the user (programmer, system developer, user or application program).

The transiently-powered embedded system solutions can be divided into two sub categories: *proactive* and *reactive* checkpointing systems.

Proactive Systems:

These systems proactively probe the energy buffer and decide whether to checkpoint system state, based on the remaining energy. To proactively probe the energy buffer, these systems insert **`trigger`** calls at specific locations in the code based on different strategies.

MementOS [171]

Ransford et al. [171] developed MementOS on top of LLVM compiler [111] and used external NVM, i.e., Flash, for storing system state (i.e stack, global variables, MCU registers etc). It works by inserting static trigger calls within the code based on three different strategies:

- ***loop-latch***: In this mode MementOS places trigger calls at the end of loop iterations.
- ***function-return***: In this mode MementOS places trigger calls at function return points.
- ***time-aided***: In this mode MementOS periodically executes the trigger calls after a fixed interval.

Loop-latch and *function-return* are the locations where one may expect the stack to store less data, which would then reduce the size and energy cost of saving system state to Flash memory. It meets **goal a: checkpoint size reduction**, as trigger calls are placed at selective locations minimizing computational overhead. MementOS also exposes an API to the programmer to insert trigger calls manually within the code. Regardless, whether the trigger calls are placed automatically or manually, they all check if the static threshold voltage has reached or not. If it has reached, trigger call will initiate checkpointing mechanism to save system state onto NVM.

The threshold voltage is obtained through repeated emulation experiments. These experiments eventually determine a single program-wide threshold, based on average run-time behavior and user-supplied energy traces (which is compulsory); thus, it does not completely satisfy **goal e: minimum user intervention**. If the voltage of the energy buffer is below

a static threshold voltage, MementOS saves the volatile state i.e. .bss, .data and stack segments along with general-purpose registers (GPR's) and peripheral state, into NVM, i.e., Flash.

MementOS does not checkpoint heap by arguing the class of applications that normally run on these devices do not use dynamic data structures. Thus, MementOS does not meet **goal b: no exclusion of data**. However, modern IoT platform developers, i.e., ARM-based platforms [20] which support MPU (memory protection unit) and MMU (memory managment unit), encourage developers to use dynamic memory structures for higher programming flexibility with more complex IoT applications [114]. This approach limits the applicability of MementOS to a wide set of IoT applications. As MementOS only handles contiguous areas of memory, the processing is quite simple.

Due to the placement strategies of MementOS, there is no upper bound on the space between two consecutive trigger calls that can be greatly increased depending upon the structure of the code. Also, because of its trivial threshold voltage based decision logic, whether to checkpoint system state, the time between last checkpoint and actual power failure can increase. Subsequently, because of these two factors, work done from the last checkpoint till actual power failure can greatly increase and not become part of any checkpoint, hence not satisfying **goal c: least amount of wasted energy**.

Data consistency is not the focus of this work, so it does not satisfy **goal d: ensure data consistency**.

DINO [173]

One major issue with proactive systems is data inconsistency. In proactive systems, there always exists a non-negligible amount of work that does not become part of any checkpoint. This work has to be performed again when the system resumes from the checkpoint. Performing the same work again can cause data inconsistency when it is performed on non-volatile variables as shown in Fig 3.2 [173].

DINO [131] is one of the first solutions that tackle this problem and ensures consistency in transiently powered systems through a task-based programming model. This approach requires the programmer to split programs statically into smaller tasks. The programmer is responsible for identifying the optimal length of the task so that the program can finish its execution.

DINO generates control and data-flow graphs of the program, where nodes in these graphs represent instructions of the program. According to DINO, If there exists a path between two nodes in a CFG (control-flow

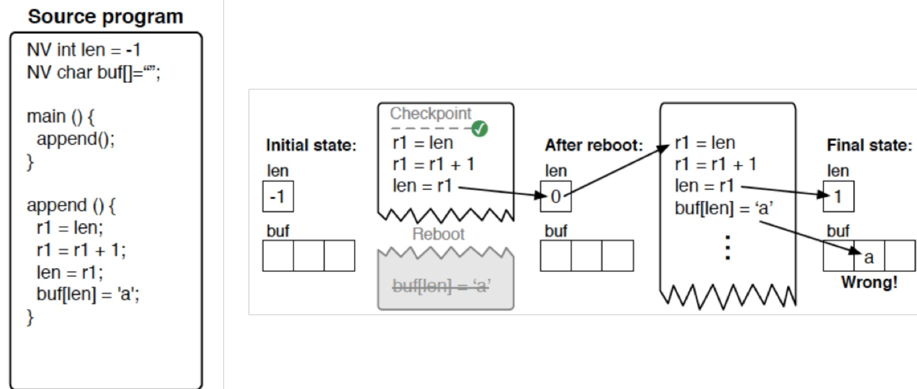


Figure 3.2: *The operations before the reboot update* ***len*** *but not* ***buf****. When execution resumes–e.g., from a checkpoint–r1 gets the incremented value of* ***len*** *and the code writes* ***a*** *character into buf's second entry, not its first. Updating* ***buf*** *leaves the data inconsistent: only* ***buf's*** *second entry contains* ***a****, which is impossible in a continuous execution [173]*

graph) and its DFG (data-flow graph), then it means both instructions come in the same path of execution and access same memory with anyone of them writing it. DINO uses CFG and DFG to identify potential non-volatile variables that can become inconsistent after reboot.

DINO's runtime environment provides support for checkpointing and data versioning between task boundaries to ensure state retention across reboots. The updates within the task are not visible to other tasks until they are completely executed. At each task boundary, DINO's checkpointing strategy copies all potential non-volatile variables that can become inconsistent after reboot onto the stack to make a volatile copy of them. At task boundary, DINO executes checkpointing mechanism just like MementOS [171], which means it does not satisfy **goal b: no exclusion of data**. Later during the restoration of the checkpointed state, DINO restore the values of non-volatile variables from stack making it consistent with the volatile state. This allows non-volatile variables to change their state only at task boundaries satisfying **goal d: ensure data consistency**.

As DINO places the burden of identifying task boundaries on the programmer, it does not satisfy **goal e: minimum user intervention**. However, during the compile time, DINO's compiler emits a warning for each boundary that can have slightly lower checkpoint size by moving task boundary by few instructions, satisfying **goal a: checkpoint size reduction**. Also, during the compile time, DINO's compiler emits a warning if a task is

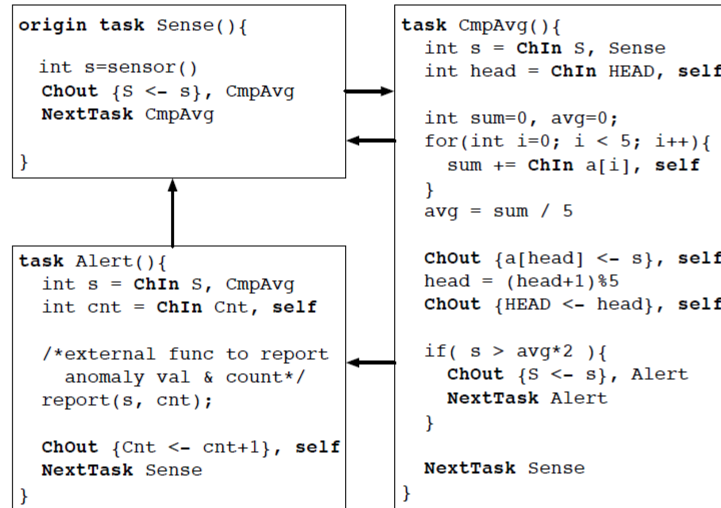


Figure 3.3: *There are three tasks in this example Chain program namely Sense, CmpAvg and Alert. Each task at its end specifies the next task to be executed, forming a chain. Each task takes input variable using channel ChIn and send the output variables using ChOut channel. These channels are named regions in NVM controlled by Chain [44].*

too long and contains I/O operations (which are costly in terms of energy). It suggests the programmer to split the task into subtasks minimizing the risk of losing work done since the last checkpoint, hence satisfying **goal c: least amount of wasted energy**

Chain [44]

Chain [44] proposes very similar approach used by DINO in which task boundary is enforced through the programming language by the programmer. Chain proposes a programming language that provides task-based flow control that ensures progress and channel-based data flow between tasks.

Each task defined in Chain's programming language has a successor and predecessor task. Each task has to define the next task to be executed after the current task is finished. Chain keeps track of the currently executing task. There is only one entry and exit point of a task and one task is marked as the origin. This ensures that control cannot jump abruptly anywhere in the program which can be seen in Fig 3.3.

The input to a task and output from that task occurs only by means of channels. A channel is a region in NVM defined between tasks for sharing data, in simple words checkpointing is happening in the channel. All

variables of a task, in Chain, are local and volatile. This allows each task to be restarted on failure with almost zero restoration cost (MCU registers and peripherals still need to be checkpointed and restored). Since input and output of tasks lie on NVM, data consistency issue is resolved, satisfying **goal d: ensure data consistency**.

The only data that Chain needs to checkpoint is; the data which tasks share among them (which can be any type of data: heap or stack), so it meets both **goal a: checkpoint size reduction** and **goal b: no exclusion of data**. However, Chain places the burden on the programmer to identify task boundaries and enforce its programming constructs, so it does not meet **goal e: minimum user intervention**. Also, as Chain only checkpoints between tasks, the processing done within the task can be lost when a power failure happens, not satisfying **goal c: least amount of wasted energy**.

Reactive Systems:

In these systems, an external interrupt may preempt the execution at any time and prompt the system to take a checkpoint. Since, in these systems, the threshold voltage is detected using in-built, or external, voltage comparator, no continuous probing of energy buffer is required.

Hibernus [21]

Hibernus [21] belongs to a class of solutions that does not need to place the trigger calls within the code. It generates an interrupt when the device voltage has dropped below the threshold voltage. If the voltage drops below the threshold, interrupt service routine saves the system state (complete RAM along with MCU registers) to the FRAM (NVM). This brings Hibernus close to the checkpoint on last practical point when the power supply fails, reducing the wasted energy and providing more energy for actual computation satisfying **goal c: least amount of wasted energy**.

Once the voltage rises above the minimum voltage requirement of the device, Hibernus recovers the saved checkpoint state from NVM and restores it to continue the computation. Furthermore, the use of FRAM for storing checkpointed state allows Hibernus to have a low threshold voltage value which further increases the active period of the main program.

The threshold voltage depends upon the size of the RAM and energy required to copy the complete RAM (and MCU registers) onto FRAM. Hibernus uses MSP430FR5739 evaluation board with built-in FRAM, voltage comparator and decoupling capacitance of size 16μF. It does not focus on reducing the size of the checkpoint that can reduce the threshold voltage;

thus, Hibernus does not satisfy **goal a: checkpoint size reduction**.

Hibernus can work with platforms with the Flash memory but this will significantly increase the energy overhead, potentially requiring additional capacitance on the supply and may also require external voltage comparator if the platform does not have the built-in comparator.

Hibernus meets **goal d: ensure data consistency**, as data inconsistency issue arises only when a non-idempotent piece of code re-executes and some of the variables used in that code are in NVM. In Hibernus, the system always resumes from where it stopped working. However, Hibernus does require user configuration of threshold voltage, not satisfying **goal e: minimum user intervention**.

Hibernus++ [22]

One main limitation of Hibernus is an off-line characterization of the device to find the threshold voltages for checkpointing and restoring the system state. This limitation was addressed in Hibernus++ [22] by the authors by proposing an adaptive approach to self-calibrate Hibernus checkpointing threashold (V_H), depending upon the system power consumptions and dynamics of energy source.

Hibernus++ self-calibration routine inserts another voltage threshold: calibration start voltage V_{cal}. Hibernus++ self-calibration routine waits for the energy buffer voltage to reach the V_{cal}. Once this voltage is reached, the harvesting source is disconnected by closing the switch and a complete snapshot of main memory (including MCU registers and peripherals' state) is saved onto NVM. The drop in supply voltage due to checkpointing is given by V_{cal} - (voltage measured at the end of the checkpointing process). Based on this information, V_H is set to minimum operating voltage (V_{min}) + the drop in supply voltage. If the calibration routine fails, V_{cal} has to be increased. V_{cal} is first set as low as possible to determine the lowest value of V_H.

This calibration strategy makes the overall system transparent and portable across multiple systems by adapting voltage threshold at run-time, considering system power consumption, decoupling capacitance and energy source behavior. This approach satisfies all the goals in the same manner as Hibernus except for **goal e: minimum user intervention**, as user intervention is reduced by run-time characterization of the device.

3.3.2 In-place Checkpointing

This class of solutions targets device architectures that employ non-volatile main memory (replacing traditional SRAM [90]). This relieves these systems from checkpointing the main memory which in return reduces the size of the checkpoint. They only need to checkpoint MCU registers and peripherals' state, increasing the energy efficiency of the system as compared to external NVM transiently-powered embedded system solutions. Also, due to in-place checkpointing, solutions under this category do not focus on reducing the checkpoint state size.

However, "in-place" checkpointing solutions are only beneficial under those scenarios where power interruption happens very frequently. As FRAM accesses are 3x slower than SRAM, applications using FRAM consume more time during execution.

Following are some goals which "in-place" checkpointing systems need to achieve to make them more reliable and efficient.

a. **Insure data consistency**: In "out-of-place" checkpointing system, data consistency issue arises only when some of its variables are declared in NVM. In "in-place" checkpointing, data consistency is still a major goal regardless all the local variables are now saved in NVM. In "in-place" checkpointing, we still need to checkpoint registers. Depending upon the mechanism used to trigger checkpointing, i.e, reactive or proactive, portion of the code is re-executed causing data inconsistencies.

b. **Least amount of wasted energy:** As discussed above, these solutions still need to save MCU special/general purpose registers and peripheral state to completely retain MCU's state. Just like in "out-of-place" checkpointing, in "in-place" checkpointing, the useful work done between the last checkpoint and power failure is always considered lost (even though data of main memory is preserved). These solutions should also try to make this loss to zero by saving MCU registers only when the remaining energy is only sufficient for checkpointing.

c. **Minimum user intervention**: Just like in "out-of-place" checkpointing, "in-place" checkpointing solutions should require only minimalistic input from the user and should work in an automated manner (configuring threshold voltage and placement of trigger calls etc.). Their working should be hidden from the user (programmer, system developer, user or application program).

Proactive Systems:

These systems do not need to checkpoint main memory, yet they still need to insert **`trigger`**s to checkpoint MCU special and general purpose registers.

Ratchet [214]

Van et al. [214] proposed Ratchet, which uses NVM as main memory. As the application data already resides in NVM, Ratchet only needs to save general and special purpose registers, along with other peripherals' state when checkpointing. The main focus of Ratchet is to solve the problem of data inconsistency in "in-place" checkpointing solutions by finding all possible write-after-read (WAR) dependencies and inserting checkpoint calls (for saving registers/peripheral states) between the instruction with WAR dependencies in the code. This is based on the intuition that if two instructions have a WAR dependency, a code must not re-execute the both instructions in the same cycle.

Van et al. implement their approach (Ratchet) on top of LLVM [111] compiler. According to [52], between any two instructions having a WAR dependency, there is an idempotent section. After finding all possible WAR dependencies, the proposed approach inserts a checkpoint call between the two idempotent section using an optimization which finds the minimum number of checkpoint call insertions (minimum trigger calls) in the code. Ratchet resolves the issue of memory inconsistency, satisfying **goal a: ensure data consistency**

Since this approach is based on statically placed checkpoint calls, there will exist some computations that will not become part of any checkpoint when energy buffer becomes empty. This will lead the approach to waste energy, not satisfying **goal b: least amount of wasted energy**.

Ratchet requires no user intervention to work, as it finds all possible write-after-read(WAR) dependencies and inserts checkpoint calls automatically; thus, satisfying **goal c: minimum user intervention**.

Clank [82]

Matthew Hicks [82] designed a system (Clank) that splits the program into idempotent sections, just like Ratchet [214]. However, instead of detecting idempotency violation with the help of compiler support like Ratchet, Clank relies on hardware support.

Matthew Hicks uses three additional hardware buffers (volatile memory) which keeps the track of each memory address accessed: read-first, write-

first, and write-back buffers. The read-first buffer holds read dominated addresses, while the write-first buffer holds write dominated addresses. During the execution of the program, Clank checks if the memory access is a "write" and if the address of the access is already in the read-first buffer, it signals an idempotency violation, satisfying **goal a: ensure data consistency**

Unlike Ratchet, Clank does not take checkpoint when idempotency violation happens. It simply adds the violating address and its value in the write-back buffer to delay the checkpointing. This allows Clank to stretch the execution of an idempotent section past its natural limits. However, when any of these three buffers overflow, Clank initiates its checkpointing mechanism (which involves saving MCU registers and peripherals' state) and then empty all the buffers.

Clank also uses watchdog timers to minimize the gap between two checkpoints because of two reasons: 1) to ensure that no single idempotent section is big enough that it exceeds average power cycle time, and 2) to reduce the overhead due to re-execution of the idempotent section. This makes Clank satisfy **goal b: least amount of wasted energy**.

Overall, Clank enables existing programs to execute in intermittent power environment automatically, without programmer intervention, satisfying **goal c: minimum user intervention**.

Jayakumar et al. [94]

However, replacing traditional SRAM with FRAM does not completely solve the problem of energy/time overhead required to save the content of main memory. As mentioned earlier, FRAM consumes more time and energy as compared to SRAM. On the other hand, SRAM, while being efficient in terms of access time and energy consumption, is volatile and vulnerable when frequent power losses occur.

This raises a question if there is a way to extract the best of both worlds. To answer this question, Jayakumar et al. [94] proposed a hybrid approach where they map different sections of the program, i.e., **`text`**, **`stack`** and **`data`**, to either FRAM or SRAM depending upon the optimal configuration at the granularity of functions. They use function as the basic unit because it can be considered as an independent entity having its own **`text`**, **`stack`** and **`data`** sections that can be mapped onto memory at runtime. They show that mapping different sections of functions in either FRAM or SRAM leads to different energy and time consumption values.

It can easily be seen that all embedded sensing devices follow deterministic behavior. The execution flow of these devices does not change and

the clock cycle/energy consumption of the application remains constant. One can analyze energy consumption on all possible memory configurations for a function and find the best maping for each function that has the least energy consumption. This one-time characterization of the device is named as *eMmap*. A checkpoint occurs only at the function boundary in an effort to reduce trigger calls. Whenever a function is executed, its sections are loaded into memory sections for which it has optimal energy measurements and when the function has finished, the volatile state (if any) is checkpointed. A function is executed only if it has enough energy to:

- migrate the code into the relevant memory section.
- execute the function.
- and checkpoint the state when the function ends.

If remaining energy is less than the sum of energy required for all these operations, the system shuts down so that it should not waste energy and recharge quickly satisfying **goal b: least amount of wasted energy**.

However, interrupts can cause non-determinism and can lead to an invalid state. Interrupts have higher priority in the system than other tasks. So if an interrupt occurs during function execution, depending upon the length of ISR(interrupt service routine), the function may or may not be able to complete even though initially, it had sufficient energy to execute.

There are two types of interrupts in the system, deterministic and non-deterministic interrupts. Deterministic interrupts are the ones that occur after a regular interval of time. This includes timer interrupt or any other periodic task. For this type of interrupts, profiling can be done in the same way as any other task. Non-deterministic interrupts (NDI) are the ones that occur when some event is detected. Example is the pressure sensor. When pressure falls below a threshold, an interrupt is generated to indicate possible leak.

Jayakumar et al. [94] handle NDIs by adding additional energy per function, in addition to that energy of the function (as measured by eMmap), to correctly execute the function. It increases the optimal energy value for each function, by a user-defined factor α, to take care of NDI. If the program has more NDIs, then programmer should set it high, otherwise, set it low.

Jayakumar et al. only ask the programmer to define α to take care of non-deterministic interrupts (NDI) satisfying **goal c: minimum user intervention**. Jayakumar et al. do not handle data inconsistency issues, not

satisfying **goal a: ensure data consistency**.

Reactive Systems:

These systems use an external interrupt that may preempt the execution at any time and prompt them to checkpoint MCU special and general purpose registers.

QuickRecall [93]

QuickRecall [93] belongs to the reactive class of solutions. It proposes a hardware/software based solution that integrates FRAM as the main memory instead of traditional SRAM. In comparison with Ratchet, QuickRecall does not need to place trigger calls within the code as it uses external comparators to check the voltage levels continuously. A trigger call is generated only when the voltage level of the device goes below the pre-defined threshold. The external comparator sends the signal to the MCU when the voltage level drops below the pre-defined threshold voltage.

The threshold voltage does not need to be high, as QuickRecall only saves registers, peripheral state and a checkpoint flag. Checkpoint flag is used to determine the existence of a valid checkpoint in memory. Whenever low voltage interrupt arrives, QuickRecall will save registers and set the checkpoint flag. QuickRecall's boot sequence then stalls the execution till the supply voltage surpasses the threshold voltage. When sufficient energy is available, QuickRecall first checks the checkpoint flag. If the flag is set, QuickRecall's ISR revives the MCU state and clears the checkpoint flag. If the flag is not set, the system will boot normally.

QuickRecall estimates triggering voltage, using the time it requires to save the registers, peripheral state, checkpoint flag, and the rate of discharge of capacitor after complete energy cut-off. The required time varies depending upon the embedded devices. QuickRecall requires the user to input this information; thus, not satisfying **goal c: minimum user intervention**.

Since a checkpointing routine is triggered only at the threshold voltage, wasted energy is zero, satisfying **goal b: least amount of wasted energy**. Due to the same reason, data inconsistency issue never arises with QuickRecall as program resumes its execution from exactly the same point it was interrupted. In this way, no piece of code executes twice, satisfying **goal a: ensure data consistency**.

3.3.3 Non-volatile Processor

Up till now, we have discussed how NVM are used as a backup for volatile memory to save the state of the system in transiently-powered environments. There is the new class of embedded systems where the entire processor is designed using these nonvolatile technologies, known as Non-Volatile Processors (NVPs).

Wang et al. [222] designed the first NVP chip, THU-1010N, that uses ROHM ferroelectric based technology for the construction of nonvolatile flip-flops (NVFF). This makes the register file, ALU, Timer and other MCU components non-volatile in nature which reduces the sleep time to 7μs and wake-up time to 3μs with zero standby power for the ambient energy harvesting devices. This architecture includes a configurable voltage detection system (CVDS) which generates signals for power failure and regain. This helps the MCU know when to sleep and wake-up.

Sakimura et al. [180] designed an architecture which uses 4072 non-volatile magnetic flip-flops (MFFs) to capture the context of MCU. They managed to reduce the wake-up time to $120ns$. There are two special instruction in the instruction set namely "SAVE" and "LOAD" to flexibly save and restore state from MFFs.

Khanna et al. [102] propose ferroelectric capacitor-based non-volatile arrays, called NVL arrays, for saving the state before going into sleep mode. It uses traditional FRAM for saving program memory. The system includes a power management state machine which receives input from power supply detector. It saves the state when the supply is low and restores it when power is regained. It claims 400ns wake-up time for ultra-low power applications.

Overall, the nonvolatile processor (in combination with FRAM) can realize complete in-place checkpointing and restore process (including registers). However, a NVP may consume more power than the volatile processor due to the inherently higher power required for a non-volatile read and write operation.

3.4 Summary

In this chapter we discussed some of the prominent solutions in the field of transiently-powered embedded systems. Table 3.1 and Table 3.2 summarize the goals achieved by *"out-of-place"* and *"in-place"* checkpointing solutions.

Overall, *"out-of-place"* checkpointing solutions target mainstream IoT

Table 3.1: *Goals achieved by each of the "**Out-of-place**" checkpointing solutions*

	Checkpoint size reduction	**No exclusion of data**	**Least amount of wastage energy**	**Ensure data consistency**	**Minimum user intervention**
MementOS	Yes	No	No	No	Energy traces
DINO	Yes	No	Yes	Yes	Task boundaries
Chain	Yes	Yes	No	Yes	Task boundaries
Hibernus	No	Yes	Yes	Yes	Threshold voltage for complete RAM
Hibernus++	No	Yes	Yes	Yes	Yes

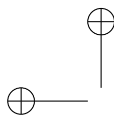
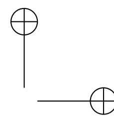
Table 3.2: *Goals achieved by each of the "**In-place**" checkpointing solutions*

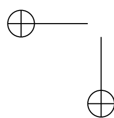
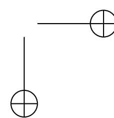
	Least amount of wastage energy	**Ensure data consistency**	**Minimum user intervention**
Ratchet	No	Yes	Yes
Clank	Yes	Yes	Yes
Jayakumar et al.	Yes	No	Yes
QuickRecall	Yes	Yes	Threshold voltage for saving MCU registers

architectures employing a volatile main memory for efficient processing, and NVM as external storage. "*In-place*" checkpointing solutions relieve the system from checkpointing main memory but it increases the energy consumption during normal computations, due to the use of FRAM as main memory. Although they are commercially available, they are not widespread and still far from massive production. On the other hand, "*non-volatile processors*" solutions relieve the system from checkpoints altogether, yet require dedicated processor designs and do not exist commercially. Both these solutions, *"in-place"* and *"non-volatile processors"*, are beneficial only under those scenarios where power interruptions happen very frequently.

Part II

System Support for Transiently Powered Embedded Sensing Systems

CHAPTER 4

”*How?*” :Designing Checkpointing Mechanism

The solutions for transiently powered embedded sensing system, as discussed in Section 3.2, is confronted with set of requirements (goals). One of the requirement is to save the system state on non-volatile memory (NVM) in an energy-efficient manner without leaving any data segment behind.

In this chapter, we present an efficient checkpointing mechanism to support embedded sensing applications on 32-bit microcontrollers whose energy provisioning is assisted through ambient harvesting or wireless energy transfer. We investigate the fundamental building block necessary to this end, and conceive three mechanisms to checkpoint and restore a device's state on non-volatile memory (NVM) quickly and in an energy-efficient manner. The problem is unique in many regards; for example, because of the distinctive performance vs. energy trade-offs of modern 32-bit microcontrollers and the peculiar characteristics of current flash chips. Our results, obtained from real experiments using two different platforms, crucially indicate that there is no “one-size-fits-all” solution. The performance depends on factors such as the amount of data to handle, how in memory the data is laid out, as well as an application's read/write patterns. This

chapter is published as *"Efficient State Retention for Transiently-powered Embedded Sensing"* in [28].

4.1 Introduction

As we have discussed in chapters 2 and 3, progresses in micro electro-mechanical systems are redefining the scope and extent of the energy constraints in networked embedded sensing. Technologies to harvest energy from the ambient can integrate with embedded devices to refill their energy buffers. A variety of these technologies appeared that apply to, for example, light and vibrations, while matching the physical constraints of the devices [56, 75, 97]. Wireless energy transfer complements these techniques by enabling opportunistic recharges. Several techniques recently appeared that enable practical wireless energy transfer at scales suitable for embedded sensing [26, 116, 122].

These technologies, however, can rarely ensure a predictable supply of energy. Computing under such transient energy conditions becomes a challenge. Devices experience frequent shutdowns, to later reboot as soon as energy is newly available. In the mean time, applications lose their state. This translates into a lack of dependable behavior and a waste of resources, including energy, as applications need to re-initialize, re-acquire state, and perform re-synchronization with other nearby devices. As a result, even if an application ultimately manages to make some progress, the overall system performance inevitably suffers.

Meanwhile, embedded sensing systems are increasingly built around modern 32-bit microcontrollers (MCUs), such as those of the ARM Cortex-M series [3]. These provide increased computing power and larger amounts of memory compared to earlier 16-bit MCUs, at a modest increase in energy consumption. These features enable employing more sophisticated algorithms and programming techniques, facilitating more demanding embedded sensing applications in several that require dependable behaviors, including wireless control [1, 11] and Internet-connected sensing [2].

In this context, we aim at allowing an application's processing to cross the boundaries of periods of energy unavailability. We wish to do so without resorting to hardware modifications that may greatly impact costs, especially at scale. Solutions to similar issues exist, for example, in the domain of computational RFIDs [172, 235], whose applications and hardware characteristics are, however, sharply different from the platforms above. As further elaborated in Section 6.2, the net result is that existing solutions are hardly applicable.

In this chapter, we study the fundamental building block to reach the goal, and investigate efficient system support to checkpoint an application's state on stable storage, where it can be later retrieved to re-start the application from where it left. Two requirements are key for these functionality:

1. they must be *energy-efficient* not to affect the duration of the next computing cycle; indeed, the energy spent in checkpointing and restoring is subtracted to the energy budget for computing and communicating.

2. they need to execute *quickly* to minimally perturb the system; as the time taken to complete the routines grows, applications may be increasingly affected as they are often not designed to be preempted.

As described in Section 4.3, the checkpoint and restore routines we design are made available to programmers through a single pair of **`checkpoint()`** and **`restore()`** functions. Key to their efficiency is the way the state information is organized on stable storage. Embedded devices are indeed typically equipped with flash chips as stable storage, which are energy-hungry and offer peculiar modes to perform read and write operations. Section 4.4 describes three dedicated storage modes that exploit different facets of how data is laid out on modern 32-bit MCUs and of the energy consumption characteristics of current flash chips.

We study the trade-offs among the three schemes and two baselines taken from the literature through real experiments using two different platforms. Our results, reported in Section 6.5, provide evidence of several trade-offs that depend, for example, on the amount of data to handle and an application's read/write patterns. Section 4.6 discusses these trade-offs based on our results, and provides insights on what kind of application may benefit most from what storage mode. Section 4.7 ends the chapter with brief concluding remarks.

4.2 Background

Our work targets modern embedded platforms, whose characteristics depart from traditional mote-class devices. Differently, existing software techniques for state retention on transiently-powered devices mostly target computational RFIDs, whose programming techniques and resource constraints do not match those of the aforementioned platforms.

4.2.1 Target Platforms

We consider 32-bit MCUs of the ARM Cortex-M series as representatives of modern embedded sensing platforms. This specific breed of MCU is gaining momentum [2], due to excellent performance vs. energy consumption trade-offs.

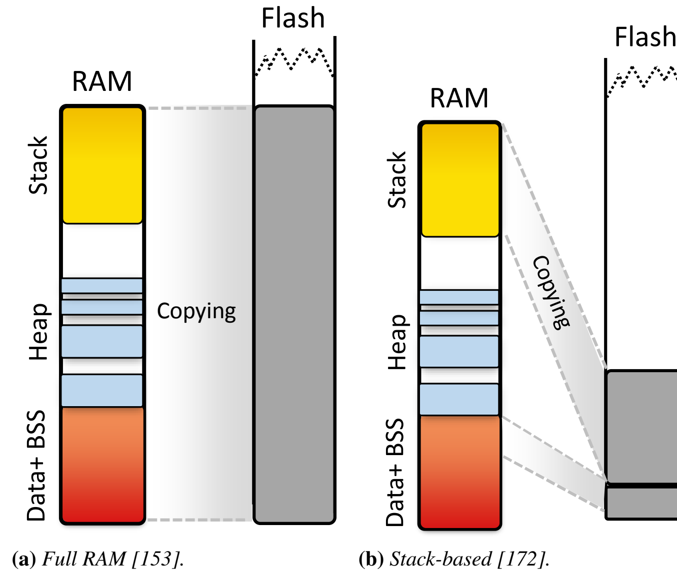
ARM Cortex-M We use two STM32 Cortex-M prototyping boards, one ST Nucleo L152RE board equipped with a Cortex-M3 MCU, and one ST Nucleo F091RC board equipped with a Cortex-M0 MCU. The two boards represent, in a sense, opposite extremes within the Cortex-M family. The Cortex-M3 board offers higher processing power, 80 KBytes of RAM space, and maximum energy consumption of 0.365 mA/MHz. Differently, the Cortex-M0 board has more limited processing capabilities, 32 KBytes of RAM space, and maximum energy consumption of 0.31 mA/MHz.

The Cortex-M design provides sixteen core registers. The first thirteen registers (R0-R12) are 32-bit General-Purpose Registers (GPRs) for data processing. The *Stack Pointer* register (SP, R13) tracks the address of the last stack allocation in RAM. The *Link Register* (LR, R14) holds the address of the return instruction when a function call completes, whereas the *Program Counter* (PC, R15) holds the address of the currently executing instruction.

The characteristics of Cortex-M MCUs as well as the availability of dedicated development environments [2,4] and efficient compilers [88] are impacting existing embedded programming techniques. A paradigmatic example is the use of heap memory. Traditionally discouraged because of overhead and lack of predictable behavior, it is increasingly gaining adoption [24, 89]. Besides providing better programming flexibility, heap memory allows developers to employ sophisticated programming languages and techniques, such as object orientation with polymorphic data types and exception handling [113]. Moreover, it facilitates porting existing libraries, such as STL containers, to embedded systems [89].

Flash memory Representative of existing platforms is also the kind of stable storage aboard both boards we use. The MCU is connected to a NAND-type flash memory chip through a dedicated instruction bus, optimized for smaller chip size and low energy cost per bit.

This kind of flash memories are divided into *sector*s, which are then sub-divided into *page*s. The two units determine the read/write modes. The flash chip on the Cortex M3 board requires to write half of the *page* size at a time, whereas the flash chip on the Cortex M0 board permits writes of a 32-bit word in a single turn. This complicates saving arbitrary amounts of



(a) *Full RAM [153].* **(b)** *Stack-based [172].*

Figure 4.1: *Existing checkpointing techniques.*

data on the flash. Moreover, the written data cannot be modified in-place as in RAM; data needs to be erased before re-writing. The unit size of an erase operation is, however, different than the unit size for writes, which further complicates matters. For example, the Cortex M0 board requires the erase of an entire 2 KByte sector at a time, possibly to modify a single bit in a sector.

These aspects combine with the peculiar energy consumption of flash chips: write and erase operations are slow and extremely energy-hungry, whereas read operations takes significantly shorter time and consumes less energy. To put things in perspective, the flash chip of the Cortex-M3 board draws 11.1 uA/MHz, that is, orders of magnitude more than any other peripheral on the board.

4.2.2 Prior Art

Checkpointing and restoring the system’s state is not a new concept. In database systems, for example, these mechanisms are used for ensuring the consistency of concurrent transactions on replicated databases [25]. In distributed debugging, checkpointing aids identifying root causes by providing the input to re-play concurrently executing processes [105]. Checkpoints are also used for ensuring fault-tolerance in redundant real-time embedded systems, such as those interconnected via wired buses [13].

Checkpoint and restore techniques are often reported for testing and experimentation using mote-class devices. For example, Osterlind et al. [153] present a checkpointing scheme similar to the one in Figure 4.1a, where the entire RAM space is transferred to stable storage. The objective is to facilitate transferring network state between testbeds and simulations, thus achieving increased repeatability. Their technique targets TMote Sky nodes. However, dumping the entire RAM space onto stable storage is likely inefficient, as the procedure also includes empty areas of memory that do not need to be saved. Moreover, the work of Osterlind et al. [153] does not necessarily support resuming the execution from the point in the code where the last checkpoint is taken, which is however required in our setting.

Chen et al. [41] augment the TinyOS operating system with mechanisms to checkpoint and restore selected components upon recognizing state inconsistencies. The mechanisms to trigger the checkpoints are generally application-specific, and meant to describe the conditions that indicate data faults. In our setting, the motivation for checkpoint and restore is different; it originates from a lack of the energy necessary to continue the computation, rather than data faults. As a result, we do not aim at checkpointing selected components, but the entire application state so that a device can survive periods when it completely shuts down.

Existing works closest to ours target computational RFIDs equipped with 16-bit MCUs and small amounts of memory, such as the WISP mote [35]. For these platforms, MementOS [171] allows programmers to inserts "trigger points" to save programmer-selected parts of the BSS or DATA sections and the stack onto stable storage, as shown in Figure 4.1b. As it only handles contiguous areas of memory, the processing is quite simple. It is, however, inapplicable to our case. For example, we are to include also heap memory as part of checkpoint and restore. This creates issues such as how to cope with fragmentation in the heap, which are specific to the setting we consider in this work.

To ameliorate the energy overhead of flash memory, Quickrecall [93] resorts to hardware modifications by replacing traditional SRAM with non-volatile ferroelectric RAM chips. However, ferroelectric RAM is currently significantly less dense and more expensive compared to normal SRAM, which makes it less desirable for high-performance embedded devices, such as those built with Cortex-M MCUs. Furthermore, memory-mapped FRAM will create data inconsistencies among non-volatile data variables. Lucia et al. [130] ensure volatile and non-volatile data consistency by asking the programmer to manually place the calls to the checkpoint routines within

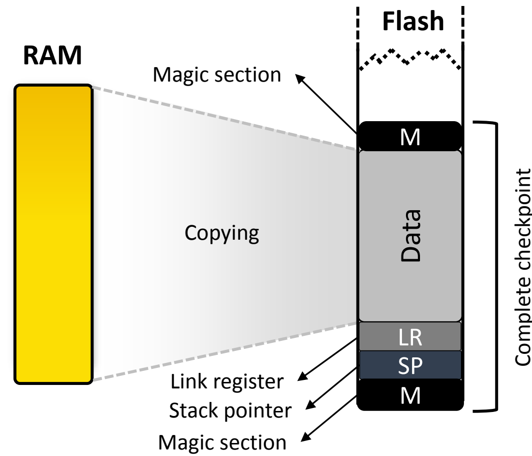


Figure 4.2: *Fundamental operation during checkpoint.*

the code. In this work, we intend to use non-volatile memory as a support, not a replacement of RAM. We only use non-volatile memory for storing and retrieving checkpoints, rather than supporting general computations.

4.3 Fundamental Operation

We describe the choice of the minimum state information required for correctly enabling a subsequent restore, as well as the fundamental operation of the checkpoint and restore routines. The former are independent of how the state information is mapped to stable storage. We deal with this aspect in Section 4.4, by presenting three different storage modes.

Our target platforms employ a plain memory map. The program data is divided into five segments: DATA, BSS, heap, stack, and TEXT. The DATA segment includes initialized global and static variables, and is typically located at the starting address of the RAM. The BSS segment is located adjacent to the DATA segment and includes uninitialized global variables. At the end of the BSS segment starts the heap segment, dedicated to dynamic data. The stack segments starts from the bottom address of the main memory and grows towards the heap. The TEXT segment resides in a flash-type memory and holds the program instructions.

In principle, the minimum state information for later restoring the device's state includes: *i)* the values of all GPRs, *ii)* the content of the RAM, including stack, heap, and the BSS and DATA segments, and *iii)* the values of stack pointer (SP), program counter (PC), and link register (LR). To checkpoint the device's state, we initially push the values of all GPRs onto

the stack through an assembly function—this is the only device-specific step in the whole procedure. Next, we proceed backward from the last address of the stable storage, as in Figure 4.2. In doing so, we:

1. save a "magic section" [172] on the last address of the stable storage, which includes a randomly generated number and the size of the RAM data we need to store—this information is used to ensure a checkpoint is complete when restoring, as explained next;

2. save the current values of stack pointer and link register: as described later, these two are sufficient to resume the computation using the checkpoint information;

3. copy to stable storage the RAM data, including stack, heap, the BSS and DATA segments, as well as the values of the GPRs we copied to RAM earlier;

4. save the same "magic section" again and pop the GPR values from the stack back into their respective registers, so the program can resume its normal execution.

These operations are made available through a single C function **`checkpoint()`** that takes the value of the stack pointer (SP) as an argument[1]. The reason why the function requires this value is because the call to the function itself affects the stack pointer. However, a checkpoint must resume the computation right after the call to **`checkpoint()`**, that is, in a situation where the stack pointer holds the same value as before the call. Because of this, it is also not necessary to save the value of the program counter (PC). The link register (LR), which holds the return address of the call to **`checkpoint()`**, carries precisely the point in the program where we wish to resume the computation after restoring.

To restore the device's state, we provide a symmetric C function **`restore()`**, which is to be called immediately after the device starts running the **`main()`** function. The key functionality is to ensure that only complete checkpoints are restored. It may indeed happen that **`checkpoint()`** is called when the energy left on the device is insufficient to complete the operation, and the device turns off before the function finishes. In these circumstances, the data on the stable storage cannot be used to resume the computation: a partial restore may ultimately bring the device to an inconsistent state that prevents any other progress.

[1]For programming convenience, this information is provided through a C macro.

To address this issue, the **`restore()`** function proceeds in the opposite way compared to **`checkpoint()`**. It first reads the magic section. Based on this, it calculates the size of the whole checkpoint and retrieves the other copy of the magic section at the end of the checkpoint. If the two copies of the random number in the magic section are equal, it means the checkpoint data is complete, that is, the **`checkpoint()`** function correctly reached the end of its processing. Only in this case, **`restore()`** proceeds by reading the data from the checkpoint to re-populate the RAM space and to update the stack pointer (SP) as well as link register (LR). Setting the latter to the instruction immediately following the call to **`checkpoint()`** makes the program resume as if the computation was never interrupted.

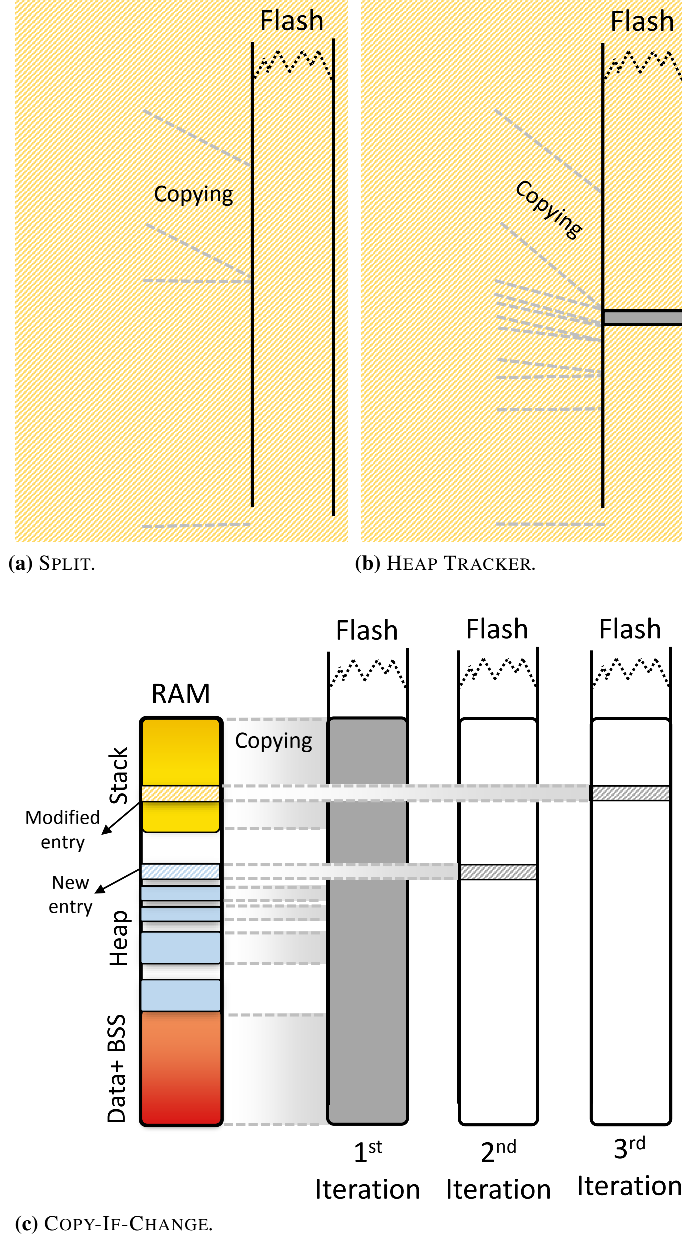
4.4 Storage Modes

The crucial aspect determining the performance of the checkpoint and restore routines is the organization of the state information on stable storage. We design three storage modes, described later and illustrated in Figure 4.3. In Section 6.5 we report on extensive experimental results revealing several performance trade-offs.

4.4.1 Split

To include the heap segment in the checkpoint, the most natural optimization over copying the whole RAM space [153] is to split the operation between the stack, heap, and the BSS and DATA segments. This allows one not to write to stable storage the unused memory space between the end of the heap and the top of the stack, avoiding unnecessary energy-hungry write operations.

Based on this reasoning, as shown in Figure 4.3a, the SPLIT mode processes the stack information and the rest of the memory segments separately. First, it copies the whole stack segment to stable storage. This is possible because, as explained in Section 4.3, the checkpoint routine receives the current value of the stack pointer as input. Next, we copy the DATA, BSS, and heap segments as a whole to stable storage. To this end, the checkpoint routine needs to know the highest allocated address in the heap segment. We gain this information by wrapping **`malloc()`** and **`free()`** with the functionality to keep track of this address as memory is allocated and deallocated during the application's lifetime. The checkpoint routine then simply copies everything below this address up to the RAM start address.



(a) SPLIT.

(b) HEAP TRACKER.

(c) COPY-IF-CHANGE.

Figure 4.3: *Storage modes.*

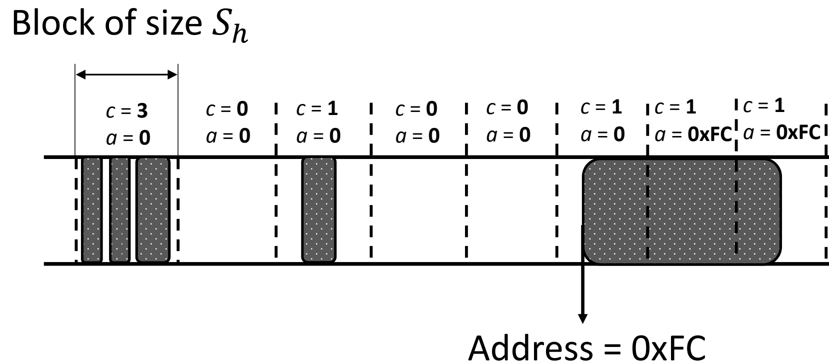


Figure 4.4: *Example configuration in* HEAP TRACKER *when allocated memory crosses multiple blocks.*

Trade-offs The processing required by SPLIT is extremely simple. Moreover, the additional state information to be kept is minimal: it solely amounts to keeping track of the highest memory address allocated in the heap. On the other hand, the custom **`malloc()`** and **`free()`** system functions introduce some slight processing overhead. In addition, space is still wasted on stable storage if the heap segment is fragmented. Again, unnecessary writes may be detrimental to the system's lifetime and, particularly, to the ability of the checkpoint routine to correctly complete.

4.4.2 Heap Tracker

To overcome the potential energy waste due to writes of fragmented areas of the heap, we must achieve higher granularity in keeping track of allocated and deallocated memory. This is, however, challenging in the general case. It is indeed quite complex to predict allocation and deallocation operations in the heap, or to forecast the size of the allocated or deallocated chunks of memory.

To address these issues, we conceive a simple, yet effective scheme called HEAP TRACKER, intuitively illustrated in Figure 4.3b. We split the heap segment in blocks of size S_h, and create a supporting data structure m with M/S_h entries, M being the maximum size of the heap. Each entry $m[i]$ carries two pieces of information: a 1 byte integer counter c and a memory address a.

The counter c records the number of memory chunks allocated in the i-th heap block. The checkpoint routine checks this information before copying the block to stable storage, and performs the operation only if c is greater than zero. A counter is necessary, rather than simply a flag, because

in the general case S_h may be larger than the size of the allocated chunks of memory, so a single block may accommodate multiple allocations. The counter for every block is incremented or decremented upon allocating or deallocating memory within the block's boundaries, again through proper wrappers to **`malloc()`** and **`free()`**.

The address a serves the cases where allocations and deallocations cross multiple blocks. For example, the right part of Figure 4.4 shows a case where a chunk of memory is allocated across three blocks. In this situation, the counters of all affected blocks are to be incremented upon allocating memory, and vice-versa when deallocating. The operation is simple in the former case, but special care needs to be taken when deallocating. Indeed, unless one modifies the internal implementation of **`malloc()`** and **`free()`**, which we would rather avoid for better portability, it is difficult to know the size of the deallocated memory when **`free()`**-ing. To address this, the memory address a is set to a value corresponding to the one returned by the original **`malloc()`** when allocating the crossing chunk. In a sense, it indicates where the crossing chunk starts out of the current block. This way, the wrapper for **`free()`** can recognize the situation based on the function's input argument, and proceeds decrementing the counter for all blocks where a matches.

Note that, as the data structure m is in the DATA segment, it implicitly becomes part of the checkpoint. The restore routine uses this information to reconstruct the heap, including the fragmented areas, before resuming the computation.

Trade-offs The choice of the value for S_h greatly impacts the performance of HEAP TRACKER. Larger values for S_h decrease the size of the supporting data structure, thus alleviating the memory overhead due to additional state information. However, the achieved granularity may still cause some un-allocated space to be written to stable storage if memory is allocated in chunks smaller than S_h. Conversely, smaller values for S_h ameliorate this issue, but increase the size of the additional state information required by HEAP TRACKER. This makes the checkpoints larger, and thus increases the energy required when writing to stable storage.

Orthogonal to this trade-off is the fact that if the block size S_h does not align with the smallest writeable unit on stable storage, the same heap block may require multiple writes on stable storage unit, as discussed in Section 4.2.1, causing unnecessary energy overhead. Among these conflicting requirements, we choose to optimize the energy spent in writing the memory blocks to stable storage, and set S_h equal to the size of the smallest writeable unit of stable storage. Based on the maximum heap size allowed

by the compiler we use, this creates an overhead of 3200 (1280) Bytes for the Cortex M3 (M0) board, which is at most 4% of the available RAM.

4.4.3 Copy-If-Change

A different take at the problem is to try and understand whether a write to stable storage is needed at all. It may indeed be the case that the previous checkpoint already includes the same information, thus re-writing to stable storage is unnecessary. This reasoning finds justification in some of the characteristics of modern flash chips, as discussed in Section 4.2.1, where read operations are often more quick and energy-efficient than writes. Thus, trading the energy necessary to read from the previous checkpoint to possibly avoid a write may be beneficial overall.

To leverage this aspect, COPY-IF-CHANGE splits the *entire* RAM space in blocks of size S_c again equal to the size of the smallest writable unit on stable storage. As illustrated in Figure 4.3c, for each such block, COPY-IF-CHANGE first reads the corresponding memory block from the previous checkpoint if available, and compares that with the current content of the RAM. If the two differ, the block is updated on stable storage; otherwise, we proceed to the next block. In the first iteration, COPY-IF-CHANGE considers the previous checkpoint as empty, thus all blocks are updated.

Trade-offs COPY-IF-CHANGE evidently incurs high overhead for the first checkpoint, as all the blocks appear as modified and need to be copied to stable storage. Conversely, the fewer modifications to the RAM, the more efficient the mode becomes, as more energy-hungry write operations are avoided. As experimentally verified in Section 6.5 and unlike the previous two modes, the energy performance of COPY-IF-CHANGE is also simple to predict, as it shows a basic relation with the number of modified memory blocks.

4.5 Evaluation

We discuss the experimental results we obtain by comparing the performance of the storage modes in Section 4.4 against each other, as well as with *i)* a mode equivalent to that of Österlind et al. [153] called FULL, whereby the entire RAM space is copied to stable storage regardless of how memory is occupied, and *ii)* a mode akin to MementOS [172] called STACK, whereby only the BSS, DATA, and stack segments are copied to stable storage.

The results we present next indicate that no single solution outperforms all others in all settings. Thus, the choice of what storage mode to employ

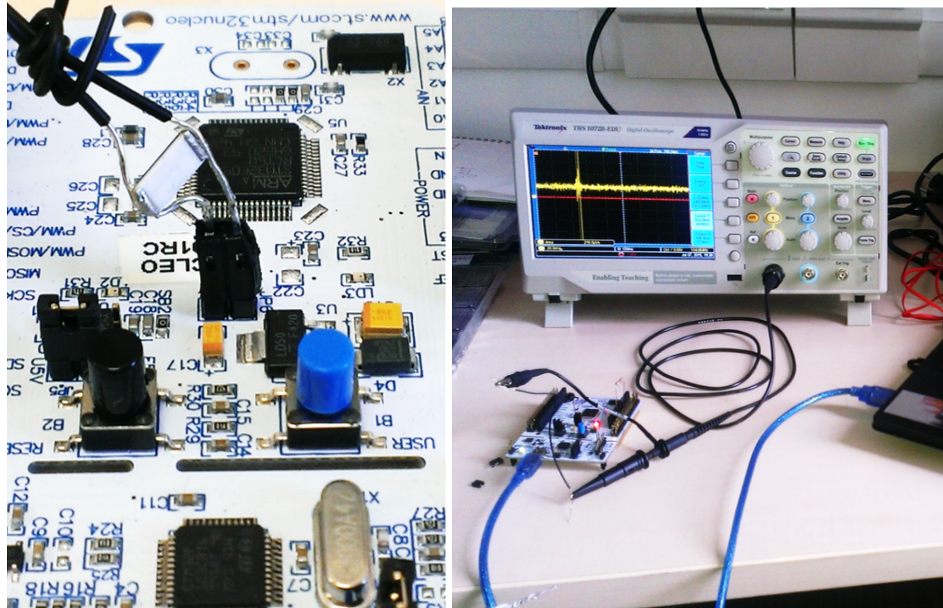


Figure 4.5: *The* 1Ω *resistor in series with the IDD connector aboard the ST Nucleo boards and the measurement setup used in the evaluation.*

depends on the application's characteristics. We provide an overarching discussion of these aspects, including examples of target applications for each storage mode, in Section 4.6.

Metrics and setup

We consider two metrics based on the requirements we elicit in the Introduction: *i)* the *energy consumption* for the checkpoint and restore routines, and *ii)* the *time* to perform the routines since the time of the call to the corresponding C function. Note that the impact of restore operations on the overall system performance is generally much smaller than checkpoint ones. This is essentially because: *i)* as already mentioned, reading from flash memories is both faster and consumes less energy than writes, *ii)* restore operations generally happen with the node charged, as opposed to checkpoints.

The metrics are a function of both the energy spent to operate on the flash chip and by the MCU for processing. To compute them, we place a 1Ω resistor in series with a dedicated connector provided by the ST Nucleo boards, shown in Figure 4.5. A Tektronix TBS 1072B oscilloscope tracks the current flowing through the resistor. This allows us to accurately record both the energy absorbed and the time taken during checkpoint or restore.

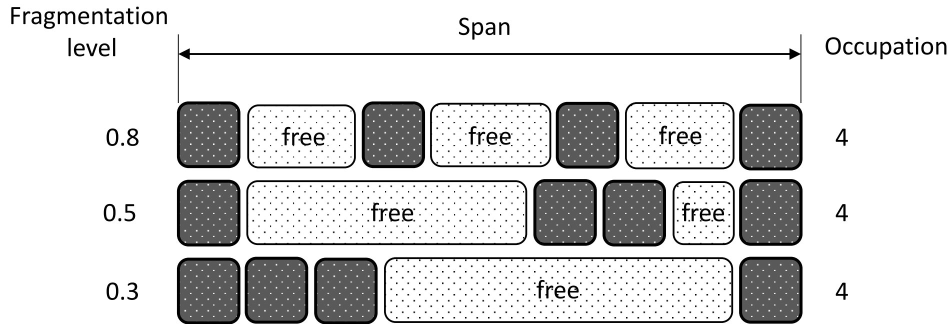


Figure 4.6: *Memory configurations representing the same* span *and* occupation*, but different* fragmentation.

We set $S_h = 128$ Bytes for HEAP TRACKER, which corresponds to the smallest writeable unit on the flash chip of the Cortex M3 board. All values we present next are averages and error bars obtained over at least ten repetitions.

Memory configuration

Conceiving a thorough set of inputs for measuring the performance of the checkpoint and restore routines is only deceptively simple. Their functioning is indeed determined by the content of the memory when running the checkpoint, which is arbitrary. Quantitatively characterizing the relevant aspects for the DATA and BSS segments might not be difficult, as the data therein is necessarily contiguous and their size is known at compile time. This is not so for the stack, and especially for the heap.

We thus consider the model represented in Figure 4.6 to synthetically characterize the inputs to our experiments, and accordingly define three metrics that apply to either the stack or the heap segment:

1. the *span* indicates the memory interval from the first allocated chunk to the last one, that is, what portion of RAM space is covered by either segment.

2. the *occupation* measures the net amount of data found in memory within a given *span*. This corresponds to the *span* only for the stack, as the memory allocation is contiguous; the same does not hold for the heap as chunks of unallocated memory may be present.

3. for the heap, the *fragmentation* measures how allocated and unallocated memory chunks are distributed within the *span*; we quantita-

tively characterize this as

$$fragmentation(x) = 1 - \frac{x \times (\text{\#free chunks of size } x)}{(\text{total free bytes})} \quad (4.1)$$

where x is the size of the largest allocated memory chunk at the time of taking the measure.

Equation (4.1) evaluates to 0.0 in configurations where it is possible to allocate the maximum possible number of objects of size x, that is, the memory is not fragmented. Differently, it evaluates to 1.0 when it is impossible to allocate any chunk of size x, that is, memory is extremely fragmented.

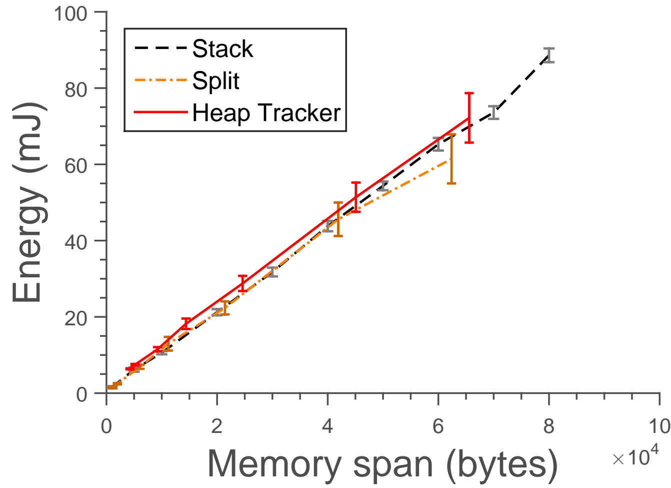
Note that, for the heap, these metrics are orthogonal. For example, the same span may correspond to different occupations. Given a value of span and occupation, different configurations may yield different levels of fragmentation depending on the distribution of the allocated memory chunks, as illustrated in Figure 4.6.

4.5.1 Contiguous Data

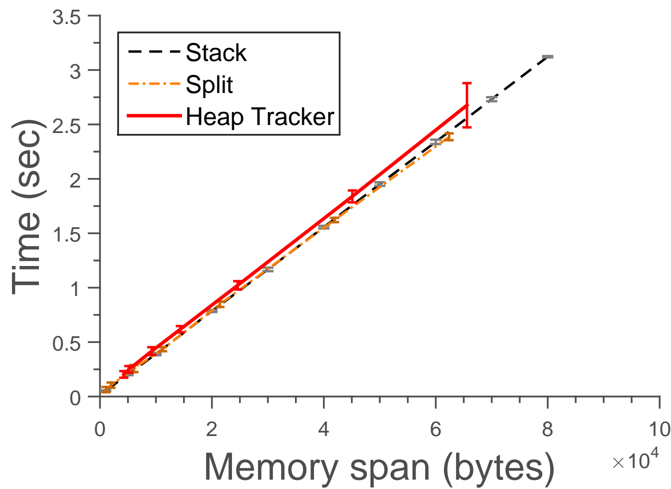
We investigate the performance of the different storage modes when RAM data is allocated in a contiguous manner. This is the case of applications whose memory demands are mostly known beforehand; in these cases, programmers tend to pre-allocate the necessary data structures. Differently, the case of non-contiguous data and general fragmentation are investigated in the following.

Setting We vary together the *span* and *occupation* of different RAM segments to understand how these affect the performance. We test values within the limits allowed by either physical memory or the compiler we use, and operate differently depending on the storage mode.

For the STACK mode, we artificially increase the *span* of the stack by growing the size of local variables in a dummy function. As STACK does not consider the heap, its manipulation is indeed immaterial. For both SPLIT and HEAP TRACKER, we artificially increase the *span* and *occupation* of the heap by growing dynamically-allocated dummy structures, and keep the stack segment to the minimum. This is to investigate the performance as the varies; if it is empty, both SPLIT and HEAP TRACKER behave equivalent to STACK. For COPY-IF-CHANGE, we initially consider the first iteration, whereby all blocks are found to be different from the previous (empty) checkpoint, and later study the case of a varying number of



(a) *Average energy consumption.*



(b) *Average time taken.*

Figure 4.7: *Cortex M3: performance of the* checkpoint *routine with increasing* span *of contiguous RAM data. Heap fragmentation is 0.* STACK and SPLIT show similar performance in this setting, whereas HEAP TRACKER suffers from the overhead of additional support data without being able to take advantage of it.

blocks requiring an update. The FULL mode covers the entire RAM space anyways.

Results Figure 4.7 summarizes the results obtained with the Cortex M3 board in energy and time, using STACK, SPLIT, and HEAP TRACKER. Overall, the values are quite limited. A checkpoint that covers the *entire*

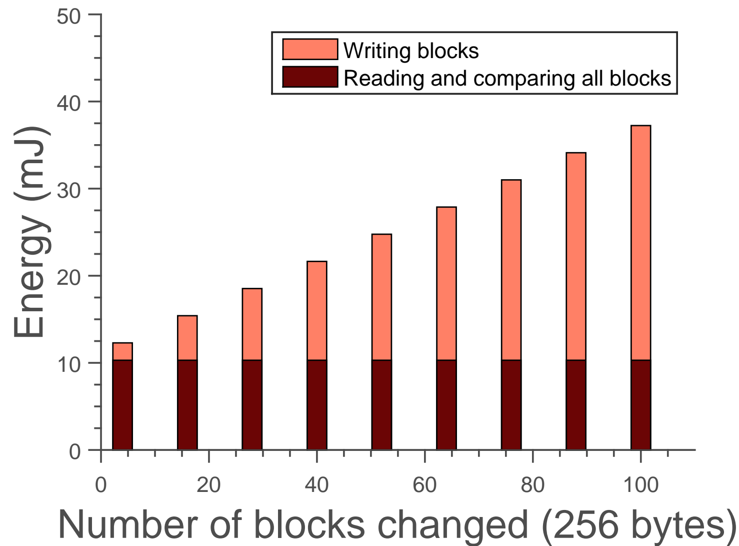
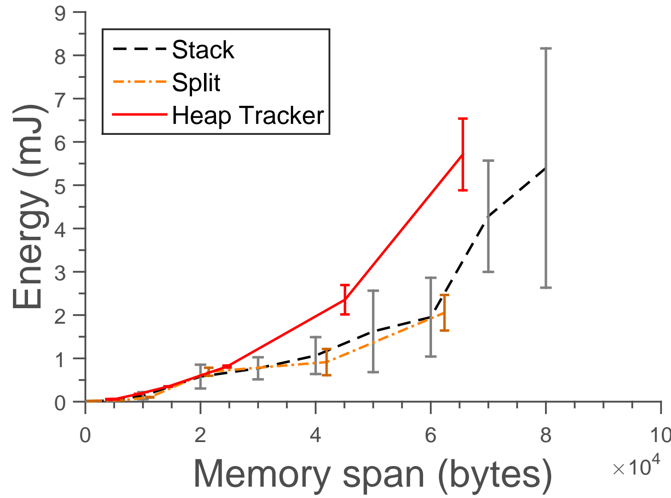


Figure 4.8: *Cortex M3: energy consumption in* COPY-IF-CHANGE *against the number of blocks that need to be updated on flash.* The performance is a function of a fixed overhead for reading and comparing all blocks, plus the variable cost of re-writing those that are found modified.

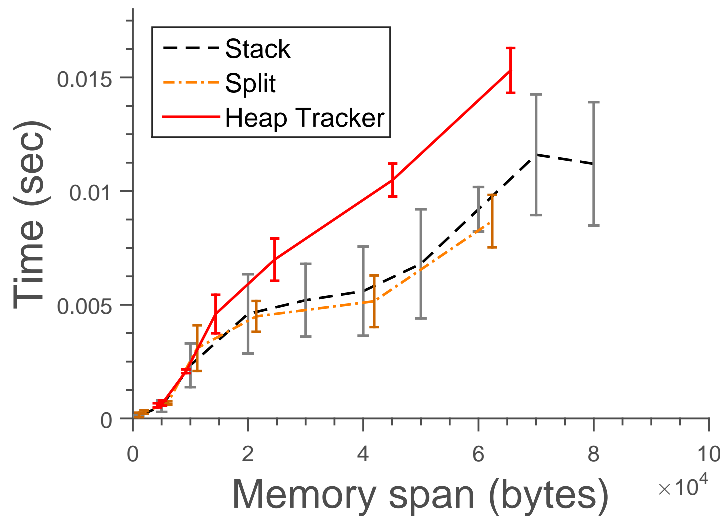
RAM space is completed in slightly more than 3 secs, arguably resulting in a moderate disruption of the application processing. In all modes, writes to the flash chip dominate both energy and time; thus the two figures are highly correlated, as seen by comparing Figure 4.7a and 4.7b.

All modes in Figure 4.7 show a linear increase in both metrics as the memory *span* grows. STACK and SPLIT follow each other closely, as their working principles are the same, that is, they copy memory segments as a whole. Differently, the performance of HEAP TRACKER is slightly, but constantly worse than STACK and SPLIT. This quantifies the trade-offs discussed in Section 4.4.2, as it represents the price for: *i)* storing the support data structure that indicates what memory blocks are occupied in addition to the application data, and *ii)* performing additional processing on such data structure during checkpoint. In absolute terms, the overhead is limited in a worst-case situation for HEAP TRACKER: the contiguous memory allocation prevents it from leveraging the ability to avoid copying some blocks if they do not cover chunks of allocated memory.

The energy performance of COPY-IF-CHANGE for the initial iteration and of FULL—not shown in the charts as they are independent of the memory *span*—is 94.5±1.34 mJ and 85.2±4.0 mJ, respectively. We are indeed considering a worst case also for COPY-IF-CHANGE, as all memory blocks



(a) *Average energy consumption.*



(b) *Average time taken.*

Figure 4.9: *Cortex M3: performance of the* restore *routine with increasing* span *of contiguous RAM data. The heap fragmentation is 0.* Because of the small absolute values, the overhead of restoring the support data and reconstructing the heap becomes more visible for HEAP TRACKER as compared to STACK and SPLIT.

are detected to be different from the previous (empty) checkpoint. In this case, COPY-IF-CHANGE also copies the entire RAM space. Unlike FULL, however, COPY-IF-CHANGE also needs to read all blocks from the previous checkpoint before deciding whether to update.

The performance of COPY-IF-CHANGE in the general case is rather il-

lustrated in Figure 4.8, where we artificially create a situation with a varying number of modified blocks in RAM, which thus require an update on flash when checkpointing. The performance corresponding to every value on the X-axis is a combination of fixed overhead caused by reading and comparing *all* the blocks, plus erasing and re-writing those that are found modified.

Figure 4.9 plots the performance of the restore routine in energy and time for the Cortex M3 board. Because of the small values at hand, the overhead of HEAP TRACKER due to additional data structures and further processing becomes more visible. During the restore routine, the latter processing might be non-trivial, as we need to carefully reconstruct the layout of the heap using the information in the support data structure. For the same reason, the measures come close to the granularity of our equipment, hence higher variability is observed. COPY-IF-CHANGE restore performance, not shown in Figure 4.9, is constant and worse than any other mode. This is because COPY-IF-CHANGE does not leverage any information about what blocks need to be restored, thus it always re-writes the whole RAM.

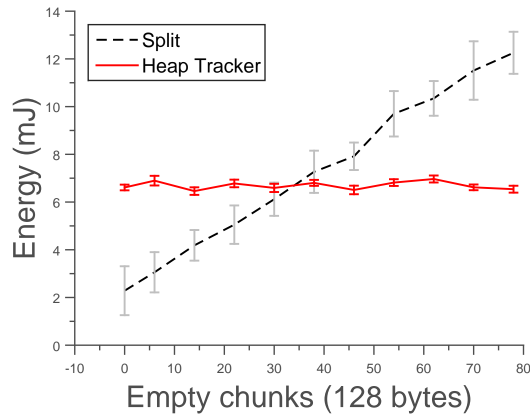
We draw similar conclusions also for the Cortex M0 board. The experiments leading to Figure 4.7 and 4.9, however, do not exercise the ability of HEAP TRACKER to avoid writing some memory blocks in case of heap fragmentation. We investigate this setting next.

4.5.2 Non-contiguous Data

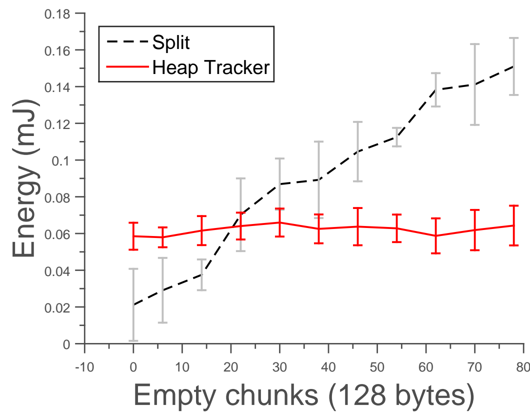
To investigate the other extreme compared to the case above, we concentrate on how SPLIT and HEAP TRACKER handle the case of non-contiguous data in the heap.

Setting We employ a configuration with a fixed memory *occupation* and a varying *span*. This models the case where data is continuously allocated and deallocated in ways that prevent the program to use previous areas of the memory. To this end, we artificially create a situation with two 128-byte chunks of dynamically allocated memory separated by a variable number of unallocated chunks of the same size. All other memory segments, including the stack, are kept to the minimum. We only consider SPLIT and HEAP TRACKER because this setting bears no influence on FULL and STACK. COPY-IF-CHANGE, on the other hand, would show almost constant performance, as it would update at most two blocks.

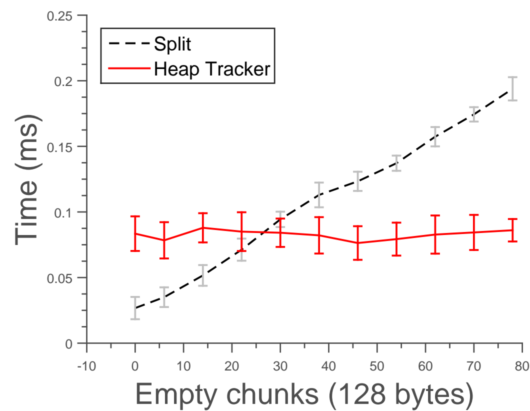
Results Figure 4.10 shows the energy and time performance for the Cortex M3 board as the *span* increases with a growing number of unallocated chunks.



(a) *Average energy consumption during checkpoint.*

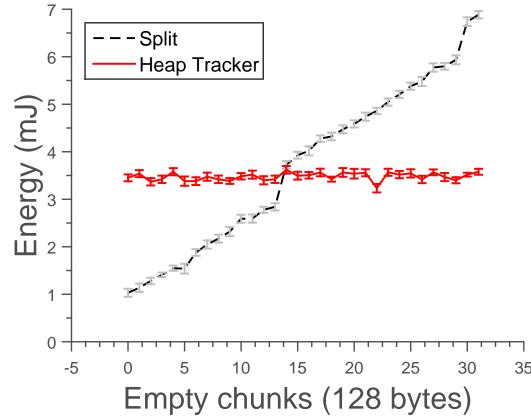


(b) *Average energy consumption during restore.*

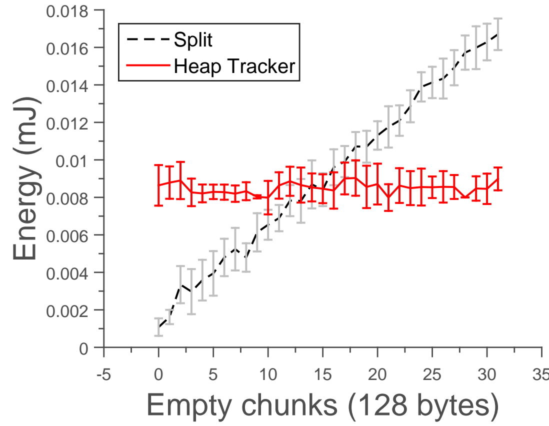


(c) *Average time taken during restore.*

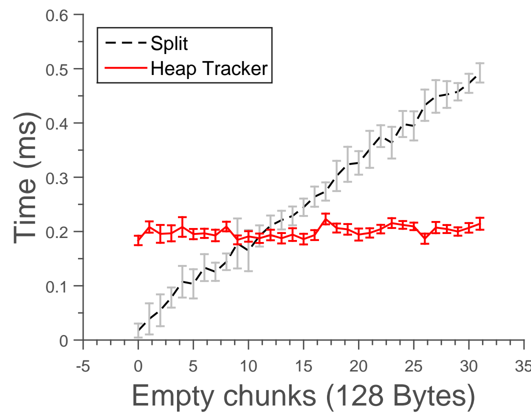
Figure 4.10: *Cortex M3: performance in the case of non-contiguous data in the heap.* HEAP TRACKER starts paying off in energy and time as soon as the number of memory blocks it can avoid writing to flash equals the size of the support data structure used to track the heap.



(a) *Average energy consumption during checkpoint.*



(b) *Average energy consumption during restore.*



(c) *Average time taken during restore.*

Figure 4.11: *Cortex M0: performance in the case of non-contiguous data in the heap.* Even though processing using a Cortex M0 is cheaper energy-wise, but also slower compared to a Cortex M3, the performance is mainly determined by the flash chip.

As for energy consumption during checkpoint, shown in Figure 4.10a, the performance of HEAP TRACKER is about constant: by tracking what blocks cover chunks of allocated memory, the net amount of bytes written to stable storage remains the same regardless of the unallocated chunks. SPLIT, however, is unable to recognize the situation. It copies an increasingly higher amount of data to the flash chip as the *span* increases, because the highest memory address allocated on the heap continues to grow.

Nevertheless, the constant performance of HEAP TRACKER is worse than SPLIT as long as the size of the unallocated memory chunks collectively equals the size of the support data structure used by HEAP TRACKER to map out the heap. This occurs around 35 unallocated chunks; as soon as this grows larger, HEAP TRACKER shows overall better performance than SPLIT. In other words, HEAP TRACKER starts to pay off whenever the number of blocks it can avoid writing to flash counter-balances the overhead due to the support data structure. Similar considerations also apply to time, whose plot we omit for brevity.

Even though the trends remain similar, the break-even point occurs earlier for the restore routine: around 20 unallocated chunks, as shown in Figure 4.10b. This is an effect of the cheaper cost, in terms of energy consumption, of read operations from flash compared to writes. In this respect, the Cortex M3 imposes a further cost: as shown in Figure 4.10c, the break-even point for time no longer corresponds to the one for energy during restore. The added overhead is imputable to the processing required to reconstruct the heap based on information in the support data structure, which in the case of the Cortex M3 becomes appreciable.

Different, and sometimes opposite consideration apply to the results obtained from the Cortex M0 board, because of the different combination of MCU and flash chip, as visible in Figure 4.11. The flash chip on the Cortex M0 board requires erasing an entire 2 KByte segment before a new write can occur on the same segment. Figure 4.11a indeed indicates two steep increases in energy consumption between 14-15 and 30-31 chunks, corresponding to when a whole flash segment needs to be erased before performing a write.

Moreover, the flash chip largely determines the restore performance. Processing on a Cortex M0 is cheaper energy-wise than on a Cortex M3, but also slower. Despite this, comparing Figure 4.11b with Figure 4.11c indicates that the break-even point between SPLIT and HEAP TRACKER occurs earlier when considering time as opposed to energy, unlike what we observe in Figure 4.10b and 4.10c for the Cortex M3. Thus, reconstructing the heap using a Cortex M0 does not impose a significant time overhead

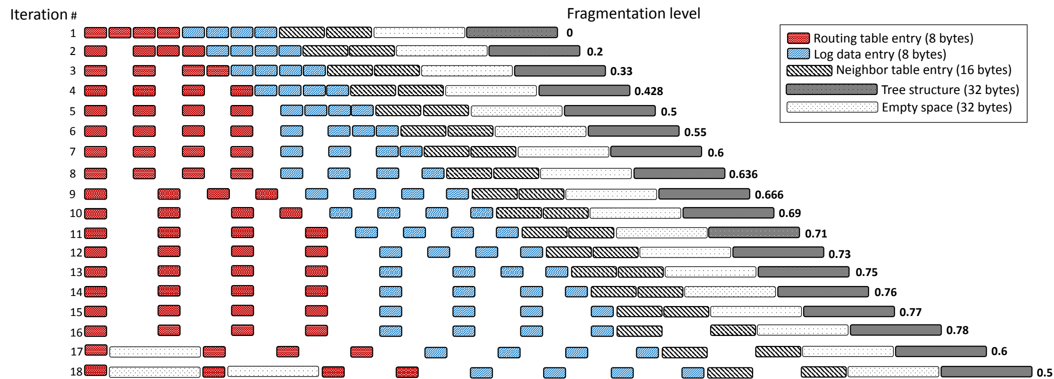


Figure 4.12: *Evolution of the memory configurations as both* span *and* fragmentation *continue to change.*

due to processing, even though the MCU is slower than a Cortex M3. The overall performance is determined by the flash chip.

The results above all consider cases of 0% fragmentation. We study next the case of varying degrees of fragmentation.

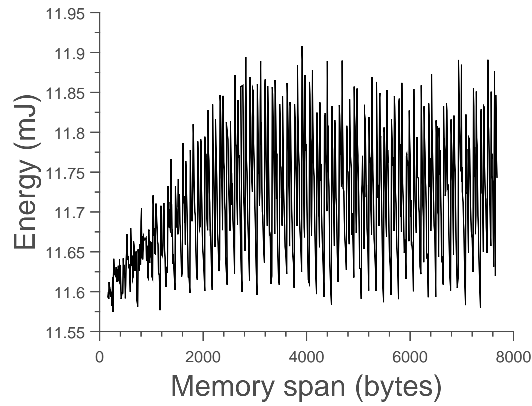
4.5.3 Fragmented Data

We aim at realistically creating different levels of fragmentation in the heap to study how SPLIT, HEAP TRACKER, and COPY-IF-CHANGE handle the situation.

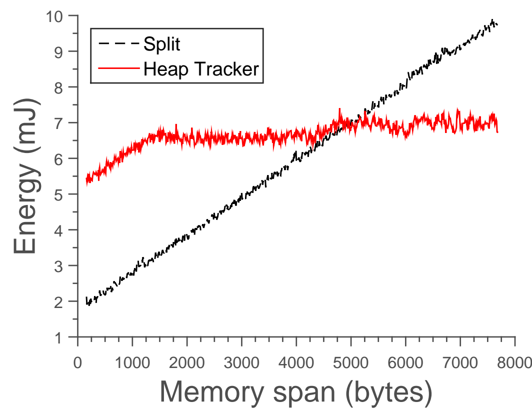
Setting We borrow the definition of data structures from the CTP [5] protocol, from the custom design logging, and from a link estimator table [6] to emulate a scenario where differently-sized data items are continuously allocated and deallocated in the heap. Figure 4.12 exemplifies the first few iterations in these experiments. Memory *occupation* remains constant, as the sum of allocated memory chunks is always the same. Both the memory *span* and *fragmentation* change at every iteration. The former grows monotonically, whereas the latter yields seemingly casual values.

Such a setting replicates—on a smaller scale—the evolution of heap memory when using general purpose libraries of commonly used data structures. The only difference compared to reality is that we force the *span* to continue grow to sweep this parameter as well, whereas normally the heap manager would eventually start re-using previously deallocated memory chunks. All other memory segments, including the stack, are kept to the minimum possible.

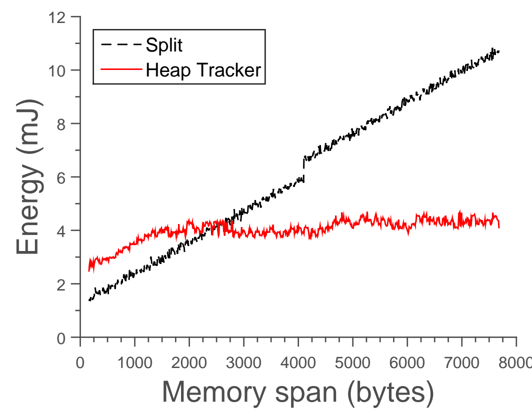
Results Figure 4.13 plots the results of energy consumption for the checkpoint routine, against a varying memory *span*. Similar overall trends are



(a) *Cortex M3:* COPY-IF-CHANGE.

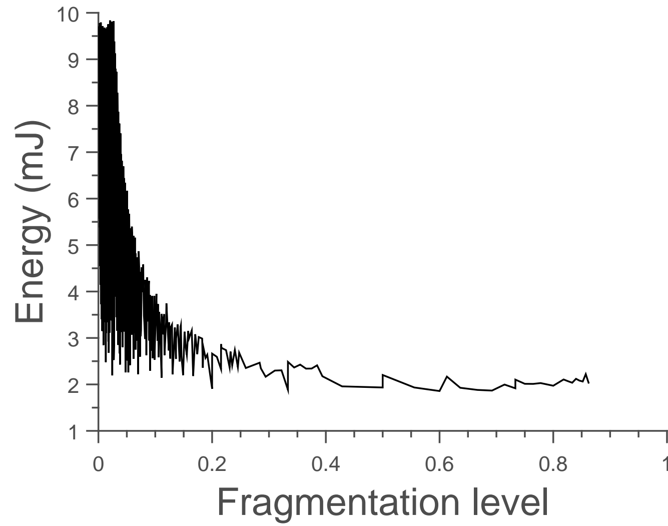


(b) *Cortex M3:* SPLIT *and* HEAP TRACKER.

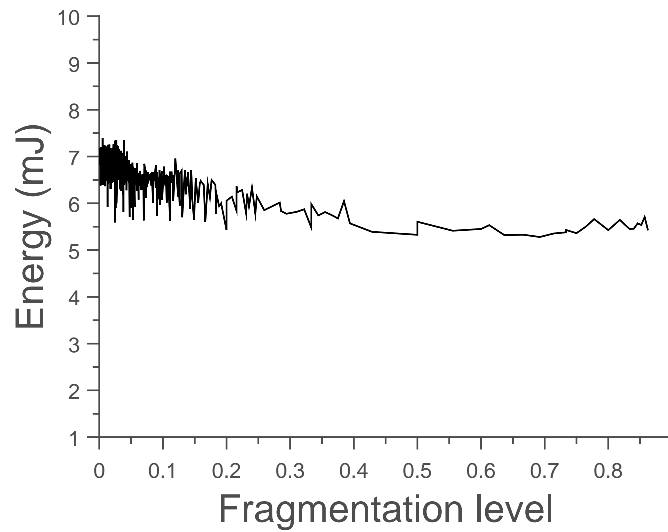


(c) *Cortex M0:* SPLIT *and* HEAP TRACKER.

Figure 4.13: *Energy consumption performance with varying* span *in case of fragmented memory, as shown in Figure 4.12.* For COPY-IF-CHANGE, the performance oscillates depending on how the changes in RAM align with the blocks on the flash chip. SPLIT and HEAP TRACKER provide different trade-offs depending on the memory *span*.



(a) SPLIT.



(b) HEAP TRACKER.

Figure 4.14: *Cortex M3: energy performance against different* fragmentation *levels.* HEAP TRACKER outperforms SPLIT for low levels of *fragmentation*, whereas the opposite holds for high levels of *fragmentation*.

observed also for time.

The results for COPY-IF-CHANGE on the Cortex M3 board are shown separately in Figure 4.13a for better clarity. The performance is highly oscillating as it depends on how the changes in RAM align with the blocks on the flash chip. Initially, the trend is increasing because the allocated chunks are still close to each other in the first few iterations, and so they "move" within the same block that constantly needs to be dumped on flash. As the allocated blocks spread out, situations where an entire block is not impacted by changes increasingly occur, eventually determining oscillations within a specific interval. Similar trends are also seen for the Cortex M0 board, yet the 2 KByte granularity of page erases changes the scale of the oscillations.

Figure 4.13b plots the energy consumption for SPLIT and HEAP TRACKER on the Cortex M3 board. The absolute numbers indicate that both outperform COPY-IF-CHANGE within the 8 KByte maximum *span* we test, which we maintain to be a reasonable limit considering the intended use of the heap on this class of MCUs [24, 89]. Comparing SPLIT with HEAP TRACKER, as discussed in Section 4.5.1, the latter provides benefits as soon as the number of blocks it can skip writing on flash counterbalances the added overhead due to storing information to track the heap. Due to fragmentation in these experiments, HEAP TRACKER skips some blocks as soon as the size of unallocated memory chunks grows larger than S_h. Because of this, the break-even point with SPLIT occurs around a *span* of 5 KBytes. Different than Section 4.5.1, however, the performance of HEAP TRACKER is not always constant: the initial increase in energy consumption is due to the same reason as in Figure 4.13a.

The results for the Cortex M0 board, shown in Figure 4.13c, show similar trends as the corresponding results for the Cortex M3 board in Figure 4.13b. Again, the different combination of MCU and flash chip makes the break-even point between SPLIT and HEAP TRACKER occur earlier, whereas the steep increase in memory consumption around 4 KBytes of memory *span* is again due to the page erase mode on the specific flash chip.

Figure 4.14 shows the results of these same experiments from the perspective of the *fragmentation* level rather than the memory *span*. We only consider SPLIT and HEAP TRACKER here, as COPY-IF-CHANGE is not directly affected by this dimension. The key observation based on comparing Figure 4.14a with 4.14b is that for low levels of fragmentation, HEAP TRACKER often outperforms SPLIT, whereas the opposite always holds for high levels of fragmentation. The explanation is that, also based on Figure 4.12, with constant *occupation* low levels of fragmentation are more likely to manifest as the *span* increases. Whenever this happens, however,

we already observed that SPLIT incurs in high costs as it is unable to optimize the writes to flash based on unallocated memory chunks. The same applies to the Cortex M0 board, despite the different hardware.

4.6 Discussion

The results we collect crucially indicate that no single storage mode is efficient in all situations. We discuss next these insights and attempt at identifying the application's characteristics that determine the recommended mode. Our conclusions are summarized in Table 4.1.

4.6.1 The role of memory *span*

Section 4.5.1 indicates that, independent of the *span*, using COPY-IF-CHANGE the overhead for reading a block from the previous checkpoint to understand whether an update is needed is quite limited. This is due to the characteristics of flash chips, where read operations are more energy-efficient than writes. Differently, the performance of all other storage modes drastically grows as the *span* increases, regardless of the need to update the previous checkpoint. As an example, comparing Figure 4.7 with 4.8 indicates that updating 25.6 KBytes of memory, that is, about one third of the entire RAM space on our Cortex M3 board, using COPY-IF-CHANGE roughly costs the same energy as using SPLIT with an *overall span* of 32 KBytes.

This observation makes COPY-IF-CHANGE attractive for applications characterized by a large memory *span* and a limited number of updates between consecutive checkpoints. This is the case, for example, of applications possibly required to run in a disconnected fashion. Under these circumstances, a node accumulates data until some form of opportunistic connection is established and data is offloaded. The data is often appended at the end of buffers while performing few changes on other data structures [148]: a pattern particularly suited to the way COPY-IF-CHANGE operates. Based on the same considerations, running SPLIT or HEAP TRACKER where COPY-IF-CHANGE is preferred would likely be inefficient. The former save all data regardless of changes and their performance worsen as the span grows, as already shown in Figure 4.7.

Opposite considerations apply to applications characterized by small memory *span* and frequent updates to data structures. This is the case, for example, of applications mostly concerned with routing packets on behalf of other nodes [68], where a small set of data structures is continuously updated as the wireless topology changes and protocols need to adapt. Most

of our results indicate that, if the *span* is limited and regardless of *occupation* and *fragmentation*, SPLIT outperforms all other modes. This is, in essence, a result of its simple operation. Compared to COPY-IF-CHANGE, SPLIT does not pay the cost of initially reading all blocks from the previous checkpoint; compared to HEAP TRACKER, it does not suffer from the overhead of support data structures. As an example, Figure 4.13 indicates that checkpointing around 3 KBytes of heap data using SPLIT costs 40% less energy than using HEAP TRACKER, and one third of that with COPY-IF-CHANGE.

4.6.2 The role of *fragmentation*

Section 4.5.2 and 4.5.3 point to a fundamental trade-off between SPLIT and HEAP TRACKER that concerns cases where the memory *span* grows without the *occupation* necessarily following. In these circumstances, HEAP TRACKER outperforms SPLIT provided the overhead of the support data structure equals the savings due to avoiding the write of some blocks on flash. The latter occurs when unallocated memory chunks are large enough to cover multiples of S_h, which we set to the smallest writable unit on the flash chip for better energy saving.

These memory configurations correspond to low levels of *fragmentation*. As an example, Figure 4.14 often demonstrates better performance for HEAP TRACKER up to 0.1 fragmentation. As a result, whenever the memory *span* is not too limited, and yet it is still not comparable to the entire RAM space, the choice of SPLIT or HEAP TRACKER ultimately depends on the expected levels of fragmentation. If the size of data structures in an application's implementation is close to S_h, low levels of *fragmentation* are likely and thus HEAP TRACKER is recommended. This may be the case of control applications, where the representation of the process state is typically rendered with complex data structures [244]. Differently, if high

Table 4.1: *Summary of insights from the experimental results and mapping to example target applications.*

Span	Fragmentation	Recommended mode	Example target
Large	-	COPY-IF-CHANGE	Disconnected operation
Small	-	SPLIT	Networking support
Intermediate	Low	HEAP TRACKER	Process control
	High	SPLIT	Remote sensing

levels of *fragmentation* are expected, SPLIT is to be favored. This would be the case of Internet-connected sensing [2], where small data items are acquired for the time necessary to perform some simple processing before sending the data out towards a long-distance destination.

4.7 Outlook and Summary

Our work here is foundational: we aim at developing the basic building blocks for state retention. In doing so, the contribution we present certainly has limitations. For example, we use synthetic settings rather than deploying real applications for evaluating the performance. Our methodology creates a controlled environment that offers repeatability and allows us to uniformly sweep the parameter space, at the cost of reduced realism. In contrast, concrete applications would introduce several sources of randomness, such as the unpredictable evolution of application state and the unequal harvesting performance across devices.

Necessary to enable an assessment in real applications is also to decide on the location of **`checkpoint()`** calls in the code. In principle, they should be placed at a point where the application makes a progress worth to be saved and the remaining energy is sufficient to complete the checkpoint. How to generalize such a notion is both challenging and orthogonal to increasing the efficiency of the individual checkpoint and restore operations, which is the goal we set forth in this work. We are currently investigating this problem with the goal of automatically deciding on the placement of **`checkpoint()`** calls, for example, based on control flow graph information, as opposed to manual placement of checkpoints by programmer [130].

In this chapter, we presented techniques to checkpoint and restore a device's state on stable storage, catering for scenarios where devices opportunistically harvest energy from the ambient or are provided with wireless energy transfer mechanisms. Our work aims at reducing the time for these operations and at minimizing their energy cost. We target modern 32-bit MCUs and currently available flash chips, making the checkpoint and restore routines available to programmers through a pair of simple C functions. The three storage modes we designed in support expose different trade-offs that depend on the memory *span*, its *occupation*, the possible *fragmentation*, and the read/write patterns in memory. The experimental results we gathered allowed us to quantity these trade-offs and discern the application's characteristics that would make one storage mode preferable over another.

CHAPTER 5

"*What?*" : Differential Checkpointing

The *Copy-if-change* technique, as discussed in Section 4.4.3, greatly reduces the data written to NVM. Performing such comparison, however, requires to sweep the entire checkpoint data, resulting in a high number of read operations on NVM.

In this chapter, we present DICE, a set of differential techniques to tracks modifications in the application state and isolates these from the slice of the previous checkpoint data that remains unaltered, without reading previous state from NVM. At the following checkpoint, DICE may thus only update the parts it detects as modified, improving the time and energy overhead of this operation. We design DICE as a complement to existing system support, and show it may be integrated with systems based on reactive (Hibernus) or proactive (MementOS, HarvOS) checkpoint mechanisms. As a result, DICE allows an existing system support to shift part of the energy budget from checkpoints to useful computations, yielding better overall energy efficiency and reduced execution latency. For example, using DICE, HarvOS can complete the execution of RSA algorithm with 86% fewer checkpoints and a 34% reduction in execution latency.

5.1 Introduction

Modern energy harvesting and wireless energy transfer techniques [27] allow embedded devices to mitigate, if not to eliminate, their dependency on traditional batteries. However, this form of energy provisioning is generally highly variable and unpredictable across space and time.

This trait clashes with the increasing push to realize tiny devices enabling unobtrusive and pervasive deployments. Energy storage facilities possibly used to ameliorate fluctuations in energy supplies need to be miniaturized as well, as they often represent a dominating factor in size. System shutdowns due to energy depletion in such settings are thus difficult to avoid. The computing pattern then becomes *intermittent* [170, 229]: periods of normal computation and periods of energy harvesting come to be unpredictably interleaved [131].

Problem: As we discuss in Section 5.2, dedicated system support exists to enable this kind of intermittent computing. Many of such systems employ a form of *checkpoint* to let the system cross periods of energy unavailability. This essentially consists in replicating the application state over non-volatile memory (NVM), where it is retrieved back once the system resumes with sufficient energy to compute.

Due to the characteristics of available NVM technology, the checkpoint operation is extremely costly both in energy and time. When using flash memories as NVM, for example, the energy cost is orders of magnitude larger than most common system operations [28,63]. Availability of FRAM technology improves these figures; still, checkpoints may often represent the dominating operation in an application's energy and time profile [21, 29]. As the cost of checkpoint is ultimately subtracted from the resources invested in useful computations, taming such an overhead is crucial.

DICE: To reduce the energy and time overhead of checkpoints, we design and implement DICE (DIfferential ChEckpointing), a set of techniques to evaluate differentials between the previous checkpoint data in NVM and the current application state. DICE uses this information to refrain from writing to NVM the slices of the previous checkpoint that remained unaltered. This way, we reduce both the energy spent during, and the time taken for checkpoints.

These differentials originate from the use of programming constructs that mutate the application state, such as variable assignments and memory references. DICE automatically instruments existing code to track such changes. Section 5.3 describes the design rationale for DICE, as well as the different techniques we employ to track changes in the application state

depending on whether they reside in globally-accessible memory, dynamic memory, or the call stack.

We conceive DICE as a complement to existing system support. This adds a further challenge. Systems such as Hibernus [21, 22] operate in a *reactive* manner: an interrupt is fired that may preempt the application execution at *any* point in time. Differently, systems such as MementOS [172] and HarvOS [29] place explicit function calls within the code to *proactively* probe the energy buffer and decide whether to checkpoint. Knowledge of when and where in the code a checkpoint may possibly take place influences what differentials need to be considered, and how to track them. Section 5.4 details the code instrumentation of DICE, together with the different techniques we employ to support both *reactive* and *proactive* systems.

Benefits: Following implementation details in Section5.5, we quantitatively report on the performance of DICE in Section6.5. Our results are based on a combination of three benchmarks across three different existing system support and two different hardware platforms. Based on more than 107,000 data points, we demonstrate both the effectiveness of our design choices and their performance impact.

For example, we show that DICE reduces the amount of data to be written on NVM by orders of magnitude when used with Hibernus, and by a fraction of the original size when used with MementOS or HarvOS. This bears beneficial cascading effects on a number of other key performance metrics. It abates the energy cost of checkpoints, reducing the peak energy demand due to checkpoints and allowing the system to use energy more for computation than for checkpoints. The former enables a reduction of up to 88% in the size of the energy buffer necessary for completing a given workload, cutting the time required to reach the operating voltage when charging and enabling smaller device footprints. The latter yields up to 97% fewer checkpoints to complete a workload, compared to those originally required. In turn, sparing checkpoints lets the system progress farther on a single charge, cutting down the execution latency up to one order of magnitude.

We conclude the chapter by surveying relevant work in the area in Section 5.7 and with brief concluding remarks in Section 5.8.

5.2 Background

Specialized custom hardware support [80, 93] exists to make the system retain the application state in the absence of energy, as discussed further in

Section 5.7. Using mainstream device architectures, however, checkpointing the application state on NVM is the only means for an application to make progress across periods of energy unavailability [81].

When to take a checkpoint, however, involves striking a trade-off between postponing the checkpoint as long as possible; for example, in the hope the environment provisions new energy, and anticipating the checkpoint to ensure sufficient energy is available to successfully complete it.

Existing solutions: Hibernus [21] and Hibernus++ [22] employ dedicated hardware support to constantly monitor the amount of energy left. Whenever the remaining energy falls below a threshold, both systems *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 in time. Both systems spare any kind of compile-time instrumentation by copying the entire memory area—including unused or empty portions—onto NVM. Using FRAM as NVM, this is still a reasonable choice energy-wise. We call this strategy *copy-all*.

Differently, MementOS [172] operates at compile-time by inserting dedicated system calls to check the voltage of the energy buffer: if the voltage is below a given threshold, a checkpoint is taken. Checkpoints then happen *proactively* and *only* whenever the execution reaches one of these calls. The threshold is heuristically determined with repeated emulation experiments of a given application code based on developer-provided energy traces, against progressively decreasing thresholds. The minimum value that guarantees completion of the workload is taken. During a checkpoint, every *used* segment in main memory is copied to NVM regardless of changes since the last checkpoint. We call such a strategy *copy-used*, and note it is mainly motivated by the need to reduce overhead when NVM operations are particularly expensive; for example, when using flash memories.

HarvOS [29] also instruments the code with dedicated system calls. Unlike MementOS that employs fixed strategies to place the calls, HarvOS does so based on the control flow graph (CFG) of the program and a worst-case estimate of the energy consumption of every edge in the CFG. Depending on the programming construct, HarvOS decides on call placement and voltage threshold to ensure that either the next system call is reached with sufficient energy to checkpoint, or a checkpoint can succeed at the current system call. HarvOS adopts the same *copy-used* strategy as MementOS.

Similar in spirit to DICE is the *copy-if-change* strategy of Bhatti et al. [28]. To understand the differentials since the last checkpoint, they compare the content of main memory with the existing checkpoint data on NVM on a word-by-word basis. Every segment found to be different is updated.

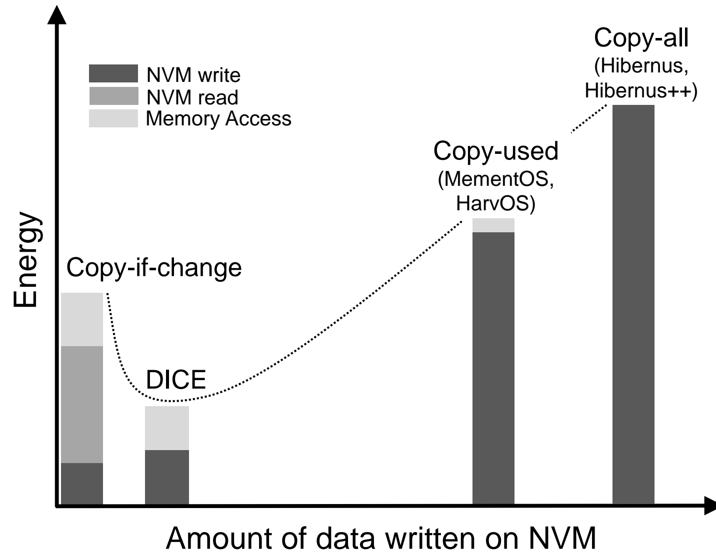


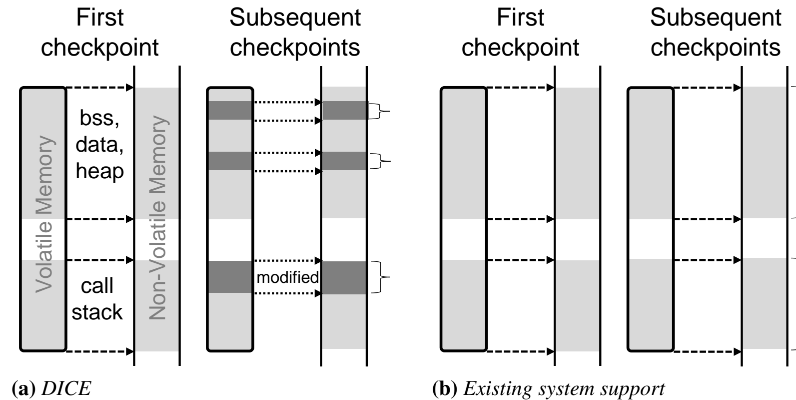
Figure 5.1: *Comparison of the different checkpoint techniques. Copy-all* in Hibernus has highest energy costs due to maximum write operation on NVM. *Copy-used* in Mementos and HarvOS avoids copying unused memory areas, reducing the energy cost of write operations. *Copy-if-change* further reduces the latter, at the cost of read operations from NVM to compute a word-by-word differential. DICE trades additional write operations to NVM, due to recording changes in the call stack at frame granularity, with the ability to only operate in main memory to record differentials.

In all the solutions hitherto described, MCU registers are always treated the same and copied to NVM during a checkpoint.

Landscape: By factoring out any performance difference due to NVM characteristics, Figure 5.1 qualitatively compares the performance of the solutions above. The plot relates the amount of data written on NVM against energy overhead. We split the latter between read/write operations on NVM, and operations in main memory.

Hibernus [21] and Hibernus++ [22] lie at the top right with their *copy-all* strategy. The amount of data written to NVM is maximum, as it corresponds to the entire memory space regardless of occupation. Both perform no read operations from NVM during checkpoint, and essentially no operation in main memory. MementOS [172] and HarvOS [29] write fewer data on NVM during checkpoint, as their *copy-used* strategy only copies the occupied portions. To that end, they need to keep track of a handful of information, such as stack pointers, adding minimal processing in main memory.

Bhatti et al.'s [28] *copy-if-change* technique is at the other extreme. Be-



(a) *DICE* **(b)** *Existing system support*

Figure 5.2: DICE *fundamental operation.* DICE allows to update the checkpoint with differentials recorded at variable level in the global context, or with modified stack frames.

cause of word-by-word comparison between the current memory state and the last checkpoint data, the amount of data written to NVM is greatly reduced. Performing such comparison, however, requires to sweep the entire checkpoint data, resulting in a high number of read operations on NVM. Because write operations on NVM tend to be more energy-hungry than reads [142], Bhatti et al. [28] demonstrate that the energy overhead is still reduced compared to a *copy-used* strategy. We return to Figure 5.1 in Section 5.3 to argue about the placement of DICE in the picture.

5.3 Overview

Figure 5.2 describes the fundamental operation of DICE. Once an initial checkpoint is available, DICE tracks changes in main memory to only update the affected slices of the existing checkpoint data, as shown in Figure 5.2a. We detail such a process, which we call *recording differentials*, in Section 5.4. DICE therefore contrasts existing system support, shown in Figure 5.2b, which works by blindly dumping on NVM the entire system's memory [21], or at most the used part [29, 172]. This happens regardless of what changes in the application state occur since the previous checkpoint.

We apply different criteria to determine the granularity for recording differentials. The patterns of data reads and writes, in fact, are typically distinct depending on the memory segment [106]. We individually record modifications in the global context, including the BSS, DATA, and HEAP

segments. Such a choice minimizes the size of the update for these segments on checkpoint data. Differently, we record modifications in the call stack at frame granularity. Local variables of a function are likely frequently updated during a function's execution. Their lifetime is also the same: they are allocated when allocating the frame, and collectively lost once the function returns. Because of this, recording differentials at frame-level abates overhead for variables whose differentials would likely be recorded together.

A dedicated *precompiler* instruments the code so that both kinds of differentials are efficiently recorded. For global context, the precompiler inserts DICE code to populate an in-memory data structure with information about modified memory areas. Identifying where such changes in global context may occur means identifying a predetermined set of statements including assignments and unary operations. The precompiler also instruments the code to record differentials in the call stack by tracking the changes to the base pointer since the last checkpoint, as explained in Section 5.4. The latter only requires to identify function calls.

In Figure 5.1, our techniques place DICE between the *copy-if-change* and *copy-used* strategies. DICE tends to write more data to NVM compared to *copy-if-change*, because modifications in the call stack are recorded at frame granularity. However, only recording differentials reduces the data to be written on NVM compared to *copy-used*, even if this happens only at frame granularity. Crucially, recording differentials only requires operations in main memory. As these are significantly more energy-efficient than read operations from NVM as used in *copy-if-change*, we argue the energy performance is ultimately improved, as shown along the Y-axis. Section 6.5 provides quantitative evidence.

5.4 Recording Differentials

We first describe the technique we adopt for recording differentials in the global context, depending on the system support. Next, we illustrate the mechanism to identify modified stack frames.

5.4.1 Global Context

DICE maintains a data structure in main memory, called *modification record*, to record differentials in global context. It is updated as a result of the execution of a **`record()`** primitive that the DICE precompiler inserts when detecting a potential change to global context. The modification records are not part of checkpoint data, as described in Section 5.5.

```
...
record(&var,sizeof(var));
var++;
...
```

Figure 5.3: *Example instrumented code to record differentials in global context.*

Figure 5.3 shows an example of instrumented code. The **record()** primitive simply takes as input a memory address and the number of bytes allocated to the corresponding data type. These information are sufficient for the modification record to keep track that the corresponding slice of the checkpoint data is to be updated: the memory values have possibly changed. How to inline the call to **record()** depends on the underlying system support, which may operate in a reactive or proactive manner, as explained in Section 5.2

Reactive checkpoints: In Hibernus [21] and Hibernus++ [22], an external interrupt may preempt the execution at any time and prompt the system to take a checkpoint. This creates a potential issue with the placement of **record()** compared to where a change in global context takes place.

If the call to **record()** is placed right after the statement modifying global context and the system triggers a checkpoint right after the latter, the modification record includes no information on the latest change during checkpoint. The remedy would be to require atomic execution of both **record()** and the statement changing data in global context; for example, by temporarily disabling interrupts. With systems such as Hibernus [21] and Hibernus++ [22], however, this may delay the execution of critical interrupts, affecting the dependability of application execution.

Because of these issues, we choose to place calls to **record()** right before the relevant program statements, as shown in Figure 5.4(b), removing the need for atomic executions. However, we must still address the case when the interrupt triggering the checkpoint occurs immediately after the call to **record()**, but before the modification occurs in the following statement. In this case, when resuming from checkpointed state, the following statement is executed first, but the corresponding changes are not tracked in the next checkpoint, as **record()** already executed for that statement before the previous checkpoint. We handle this by marking the memory region reported in the most recent call to **record()** to be included in the next *two* consecutive checkpoints. We prefer this minor additional overhead for these corner cases, rather than atomic executions.

The operation of the precompiler is straightforward when the statements modifying global context are part of compound statements, such as the

```
1   ...
2   int v1,v2,v3,arr[size];
3   ...
4   void foo(){
5     int local_v;
6     ...
7     if(local_v < MAX){
8       local_v++;
9
10      v1 =local_v;
11
12      v2 =v2 + local_v;
13
14
15    }
16    else{
17
18      v2 = MAX;
19
20    }
21    for(int i=0;i<SIZE;i++){
22
23      arr[i] =zoo(i);
24    }
25
26    trigger();
27    ...
```

(a) Before instrumentation

```
1   ...
2   int v1,v2,v3,arr[size];
3   ...
4   void foo(){
5    int local_v;
6    ...
7    if(local_v < MAX){
8      local_v++;
9      record(v1,sizeof(v1));
10     v1 = local_v;
11     record(v2,sizeof(v2));
12     v2 = v2 + local_v;
13
14
15   }
16   else{
17     record(v2,sizeof(v2));
18     var2 = MAX;
19
20   }
21   for(int i=0;i<SIZE ;i++){
22     record(arr[i],sizeof(int));
23     arr[i] =zoo(i);
24   }
25
26
27    ...
```

(b) After instrumentation (reactive)

Figure 5.4: *Example instrumentation for reactive checkpoints.* With reactive checkpoints, each statement possibly changing global context data is preceded by a call to **`record()`**.

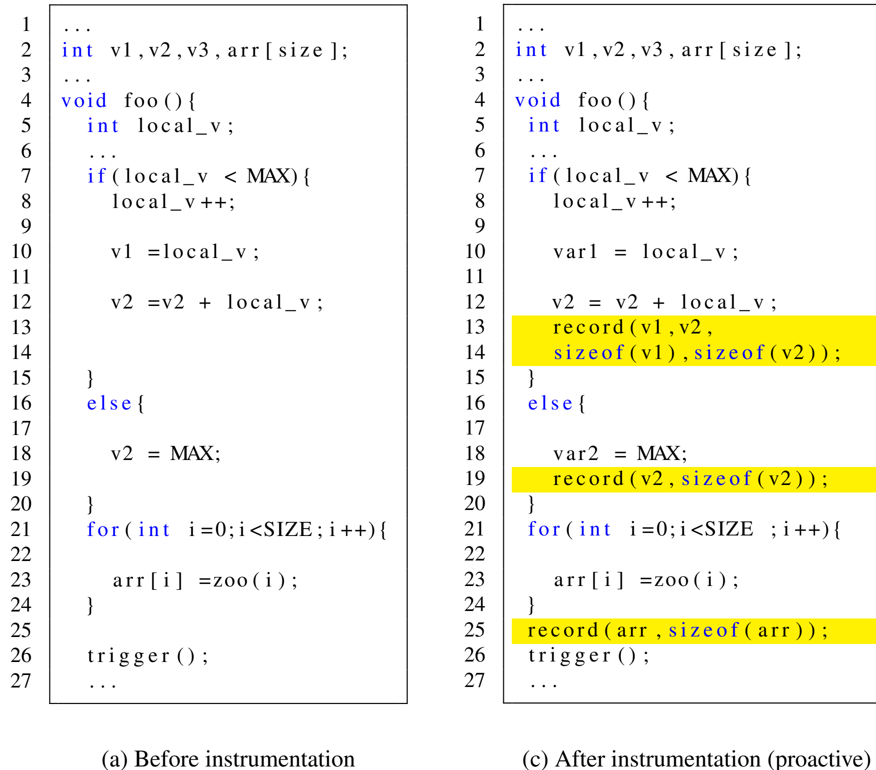
cases in Figure 5.4 and Figure 5.5. However, additional care is necessary at certain unusual locations where the call to **`record()`** is not trivial to add. For example, the ternary operator (**`?:`**) may also modify global variables, yet in the C language the operator cannot be used with multiple statements in either of the sub-expressions. Here, we may either convert the ternary operator into a traditional **`if-else`**, or pessimistically record an address as potentially modified even if the corresponding statement may not be executed. DICE currently employs the latter strategy, due to its general applicability for all similar situations.

Proactive checkpoints: Systems such as MementOS [172] and HarvOS [29] insert systems calls called **`trigger`**s at specific locations in the code, as described in Section 5.2. These triggers, based on the current state of the energy buffer, decide whether to checkpoint before continuing. This approach, besides eliminating the need to insert **`record()`** calls for every statement possibly modifying global context, exposes the code to further optimizations.

As example, Figure 5.5(b) shows the same code as Figure 5.4(b), now instrumented for a proactive system support. For code segments that do not involve loops, we may now aggregate updates to the modification record at the *basic block* level or just before the call to **`trigger()`**, whichever comes first. The former case is shown, for example, in line 8 to 14. In these cases, we cannot postpone the update to the modification record any further, as branching statements can determine only at run-time what basic block is executed.

In the case of loops operating on contiguous memory areas, further optimizations are available. Say a loop is employed to process the elements of a sequential data structure, as in Figure 5.4 lines 21 to 24. A call to **`record()`** inside the loop body, necessary in Figure 5.4(b) for each and every iteration of the loop, may be now replaced with a single call right before the call to **`trigger()`** to record modifications in the whole data structure at once, as shown in Figure 5.5(b) line 25.

Certain peculiarities of such a technique warrant careful consideration. For instance, loops may, in turn, contain branching statements that may execute before updating one of the sequential elements. This may lead to false positives in the modification record, which would result in an overestimation of differentials. Similarly, a loop with a complex expression in its *condition* part may not necessarily inspect the whole sequence. A number of fine-grained optimizations may be possible in these cases, which however would require to drastically increase the complexity of instrumentation and/or to ask for developer intervention. As of now, we opt for a conser-



(a) Before instrumentation

(c) After instrumentation (proactive)

Figure 5.5: *Example instrumentation for proactive checkpoints.* With proactive checkpoints, code locations where a checkpoint may take place are known, so calls to **`record()`** can be aggregated to reduce overhead.

vative approach: we record modifications on the entire memory area that is *possibly*, but not definitely modified inside the loop, favoring simplicity over a slightly increased overhead.

5.4.2 Call Stack

Unlike data in global context, we record differentials of local variables in a function at frame level. As described in Section 5.3, this is essentially because local variables in a function are often modified together and their lifetime is the same. To this end, DICE monitors the growth and shrinking of the call stack, respectively due to function calls and returns. At the time of checkpoint, we update the stack frames possibly modified since the last checkpoint.

Operations on the stack are normally handled through a pair of memory pointers. The *base pointer* (BP) points to the base of the frame of the currently executing function. It normally serves to access parameters and local variables, which happen to be located at a fixed offset from BP. The *stack pointer* (SP) points to the top of the stack, and is normally used to determine where to start allocating a new frame.

Recording differentials at frame level in DICE only requires one additional memory pointer, called the *stack tracker* (ST). We use ST to keep track of changes in the value of BP between successive checkpoints. To identify the region of the stack possibly modified, we only need to know how BP moved since the previous checkpoint. We proceed according to the following four rules:

R1: ST is initialized to BP every time the system resumes from the last checkpoint, or at startup;

R2: ST is unchanged as long as the current or additional functions are executed, thus more frames are pushed onto the stack, that is, ST does *not* follow BP as the stack grows;

R3: whenever a function returns that possibly causes BP to point deeper in the stack compared to ST, we set ST equal to BP, that is, ST follows BP as the stack shrinks;

R4: at the time of checkpoint, we save the memory region between ST and SP, as this corresponds to the set of frames possibly changed since the last checkpoint.

Figure 5.6 depicts an example. Pretend the system is starting with an empty stack. Therefore, ST, SP and BP all point to the base of the stack according to **R1**. As the application unfolds, a chain of three nested function

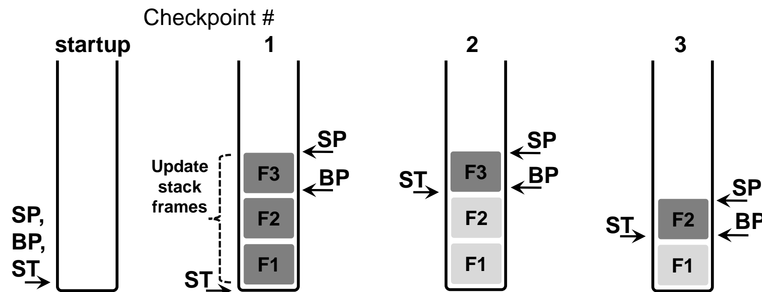


Figure 5.6: *Identifying possibly modified stack frames.* The stack tracker (ST) is reset to the base pointer (BP) when the system resumes or at startup. ST does not follow BP as the stack grows, but it does so as the stack shrinks. The dark grey region between ST and the stack pointer (SP) is possibly modified.

calls is executed. The stack grows accordingly. While executing **`F3`**, BP points to the base of the corresponding frame. Say a checkpoint happens at this time, as shown under checkpoint #1 in Figure 5.6: the memory region between ST and SP is correctly considered as a differential since the initial situation, due to **R4**. The checkpoint is accordingly updated.

When resuming from checkpoint #1, we set ST equal to BP because of **R1**. Function **`F3`** continues its execution; no new functions are called and no functions return. According to **R2**, ST and BP remain unaltered. The next checkpoint happens at this time. As shown for checkpoint #2 in Figure 5.6, **R4** indicates that the memory region to consider as a differential for updating the checkpoint corresponds to the frame of function **`F3`**. In fact, the execution of **`F3`** might still alter local variables, requiring an update of the checkpoint data.

When the system resumes from checkpoint #2, function **`F3`** returns. Because of **R3**, BP is updated to point to the base of the stack frame of **`F2`**. If a checkpoint happens at this time, as shown under checkpoint #3 in Figure 5.6, **R4** indicates the stack frame of function **`F2`** to be the differential to update on the checkpoint. This is necessary, as local variables in **`F2`** might have changed once **`F3`** returns control to **`F2`** and the execution proceeds within **`F2`**.

Note that the efficiency of recording differentials in the stack at frame level also depends on programming style and of specific NVM technology. If function calls are often nested, the benefits brought by this technique likely amplify compared to tracking individual local variables. Differently, on flash memories, updating an existing checkpoint with frames smaller than the size of a page may not lead to any benefit compared to track-

ing individual variables, as the corresponding page needs to be erased and rewritten anyways. To cater for specific deployment configurations, programmers may still opt to record differentials in the call stack at variable level, using a specific precompiler option.

5.5 Implementation

The implementation of DICE includes: *i)* the precompiler to instrument the code for estimating differentials, as in Section 5.4, *ii)* the implementation of the **`record()`** primitive to update the modification records, and *iii)* a modified implementation of the checkpoint routine of the chosen system support.

5.5.1 Precompiler

We implement the DICE precompiler targeting the C language using ANTLR [159]. The precompiler instruments the code for recording modifications in the global context as described in Section 5.4.1, and for identifying modified regions of the stack, as explained in Section 5.4.2. For the former, command-line parameters are accepted also to indicate the kind of underlying system support to target, being it proactive or reactive. We test the correctness of the precompiler implementation on a range of applications found in existing codebases, including those we use in the evaluation of Section 6.5.

As a result of the precompiler operation, DICE can capture modifications in main memory except for those originating from peripherals with direct memory access (DMA). These changes bypass the execution of the main code; thus DICE cannot track them. However, in embedded platforms, DMA buffers are typically allocated by the application or the OS, so we know where they are located in main memory. We may either always consider such memory areas as modified, or at run-time, flag them as modified as soon as the corresponding peripheral interrupts fire, independent of their processing.

5.5.2 `record()`

We implement **`record()`** as a variable argument function. This way, in the case of proactive checkpoint support, we can aggregate tracking multiple changes in main memory with a single call, as shown in Figure **??**(c).

How to store the modification records progressively accumulated bears an impact on the overhead of DICE on both the application processing

and main memory consumption. Out of the many different data structures available to this end, we choose to employ a simple bit-array, where each bit represents one byte in main memory as modified or not. Such a representation is particularly compact, causing little overhead in main memory. Crucially, it allows **`record()`** to run in constant time, as it supports direct access to arbitrary elements. This is key to prevent **`record()`** from changing the application execution timings, which may be critical on resource-constrained embedded platforms [230].

Such a data structure, however, causes no overhead on NVM, as it does not need to be part of the checkpoint. Every time the system resumes from the previous checkpoint, we start afresh with an empty set of modification records to track the differentials since the the time the system restarts.

5.5.3 Checkpoint

We also need to replace the checkpoint operation implemented in existing system support, either as an interrupt handler (in reactive systems) or as part of the **`trigger()`** (in proactive systems), with a DICE-specific one. We first estimate the checkpoint differentials by inspecting the modification records and the current values of ST and SP, as described in Section 5.4. These differentials, along with MCU registers, are used to update the last available checkpoint.

Similar to existing work [28,172], we ensure the validity of a checkpoint by adding a random byte at the beginning and at the end of the checkpoint. At the time of resuming, DICE checks if the two byte sequences are equal, which means the checkpoint operation completed successfully.

5.6 Evaluation

We experimentally assess the performance of DICE along multiple dimensions, using a combination of three benchmarks across three system support and two hardware platforms. Based on 107,000+ data points, we conclude that:

- DICE reduces the amount of data to be written on NVM during a checkpoint by orders of magnitude compared to the original Hibernus, and by a fraction of the original size with MementOS or HarvOS;
- such a reduction abates the peak energy demands and allows systems to use up to 88% smaller energy buffers to complete the same workload, cutting the time to reach the operating voltage and enabling smaller device footprints;

- the energy saved in checkpoints can be used for computation: DICE results in up to 97% fewer checkpoints required to complete the workload, compared with those originally required, resulting in better energy efficiency;
- fewer checkpoints also mean reduced execution latency: using DICE, the same workload is completed in a time up to one order of magnitude shorter than with the original system support implementations.

In the following, Sec. 5.6.1 describes the settings, whereas Sec. 5.6.2 to Sec. 5.6.5 discuss the results.

5.6.1 Settings

Benchmarks: We consider three benchmarks widely employed to evaluate system support for transiently-powered computing [21, 93, 172, 214]: *i)* a Fast Fourier Transform (FFT) implementation, *ii)* RSA cryptography, and *iii)* Dijkstra spanning tree algorithm. FFT is representative of signal processing functionality in embedded sensing devices. RSA is a paradigmatic example of the kind of security support deployed on modern embedded systems. Dijkstra's spanning tree algorithm represents a staple case of graph processing functionality, which may be found in some form also in embedded network stacks [92].

These benchmarks offer a variety of different programming structures, data types, memory access patterns, and processing load. For example, the FFT implementation operates mainly over variables local to functions and has moderate processing requirements, whereas RSA operates mainly on global data and demands great MCU resources. Dijkstra's algorithm is less computationally-intensive than the first two; it mainly handles integer data types as opposed to variable-precision ones, but exhibits much deeper levels of nesting due to loops and function calls, requiring a denser instrumentation compared to the first two. Such diversity allows us to generalize our conclusions. All implementations are taken from public code repositories [2].

Systems and platforms: We measure the performance of DICE when used with either Hibernus [21], MementOS [172], or HarvOS [29]. We thus test DICE with both reactive and proactive checkpoints, investigating the different instrumentation strategies in Section 5.4.1. We consider as baselines the original unmodified Hibernus, MementOS, and HarvOS using either the *copy-all* or *copy-used* strategies, as well as the *copy-if-change* [28] strategy where meaningful. To make our analysis of MementOS independent of the deployment-specific energy traces used to identify a suitable voltage

threshold, as described Section 5.2, in all experiments we manually sweep the possible settings of such parameter with steps of 0.2V, and always use the best performing one.

We run Hibernus on an MSP430-based TelosB interfaced over SPI with a byte-programmable 128 KByte FRAM chip as NVM, akin to the hardware originally used for Hibernus [21]. MementOS and HarvOS run on a Cortex M3-based ST Nucleo with a standard flash chip as NVM, already used to compare MementOS and HarvOS [29]. Neither of these boards is explicitly targeting transiently-powered computing; yet, as much as the original system support we plug DICE into [21, 29, 172], here we are only interested in data processing on the MCU and read/write operations on NVM. From this perspective, these boards offer a range of hooks to trace the execution, enabling fine-grained measurements. Further, our choice of platforms ensures direct comparison with existing literature.

Metrics: We compute four key metrics to quantify the benefits brought by DICE as well as the system overhead:

- The *update size* is the amount of data written to NVM during a checkpoint. Using DICE, it is determined by the modification records. For the baselines, it is determined by the specific strategy, as described in Section 5.2. This is the key metric that DICE seeks to reduce: assessing this figure is a stepping stone to understand the performance gains DICE provides in all other metrics.
- The size of the *smallest energy buffer* is the smallest amount of energy that allows the system to complete a fixed workload. If the energy buffer is too small, a system may be unable to complete checkpoints, ending up in a livelock situation where the execution always resumes from the same point and makes no further progress. On the other hand, transiently-powered devices typically employ a capacitor as energy buffer: a smaller capacitor reaches the operating voltage more quickly and allows one to reduce a device's form factor.
- The *number of checkpoints* is the number of times the system takes a checkpoint because energy is about to be exhausted, to later resume once the energy buffer is full. The more the checkpoints, the more the system subtracts energy from useful computations. Reducing the size of data written on NVM should allow DICE to shift part of the energy budget from checkpoints to computation, allowing an application to progress further on the same charge and reducing the number of required checkpoints. In essence, this metric represents the energy efficiency of a given system configuration.

Table 5.1: *Measured FRAM and flash characteristics*

Platform	Operation	Time			Current
		1 byte	256[1] bytes	512 bytes	
FRAM	Read	0.15 ms	6.47 ms	12.8 ms	360 μA
	Write	0.18 ms	7.52 ms	14.9 ms	360 μA
Flash	Read	n.a.	0.02 ms	n.a.	12.4 mA
	Write	n.a.	212 ms	n.a.	12.4 mA

[1] Flash is page-programmable with page size of 256 bytes on Nucleo boards. The write operation also includes energy required to erase a page, as necessary before rewriting.

- The *execution latency* is the total time to complete a fixed workload, including checkpoints and resume operations, but excluding the recharge times that are generally deployment-dependent. DICE introduces a runtime overhead due to recording differentials. On the other hand, smaller updates should reduce both the time required for a single checkpoint and, because of the reasoning above, their number too.

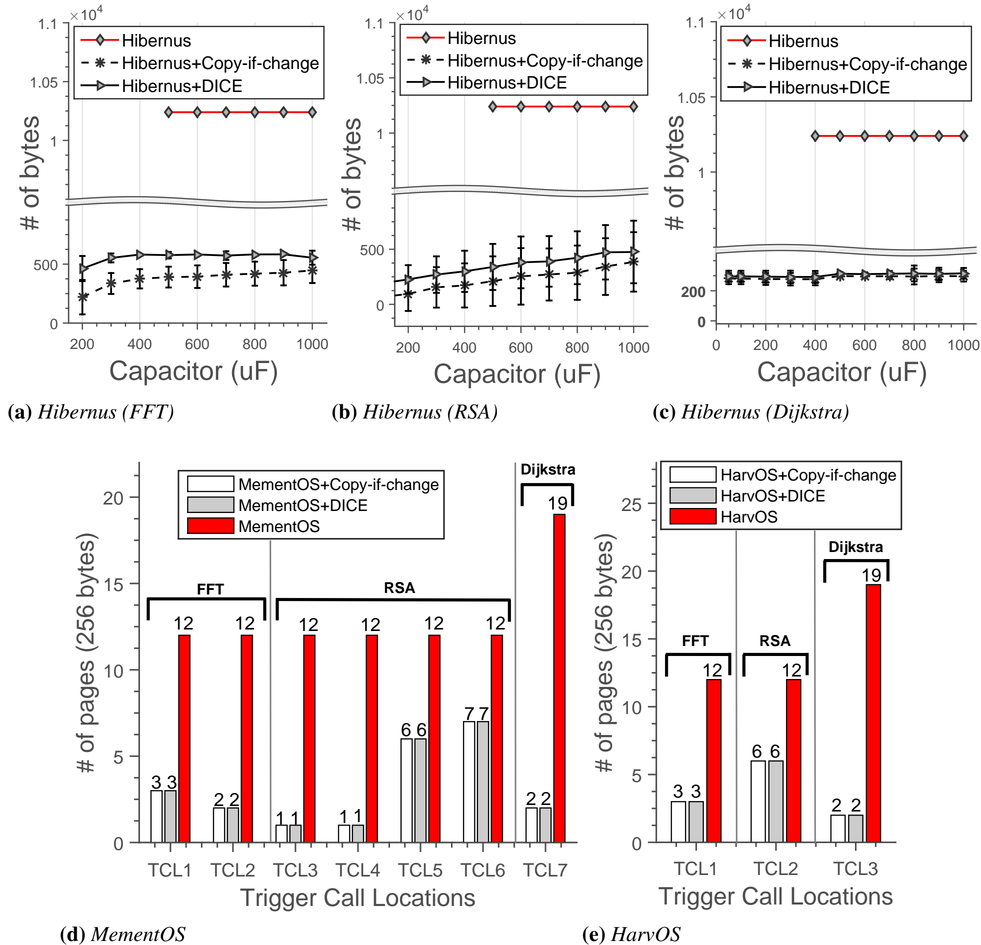
To accurately compute these metrics, we trace the execution on real hardware using an attached oscilloscope along with the ST-Link in-circuit debugger and Kiel μVision for the Nucleo board. This equipment allows us to ascertain the time taken and energy consumption of every operation involved during the execution, including checkpoints on FRAM or flash memory. Our findings on the performance of these two, summarized in Table 5.1, closely match existing measurements [142]. This is key to the accuracy of the results, enabling emulation of additional executions against arbitrary inputs and variable-size energy buffers.

The executions to gather the actual measurements are assumed to employ a capacitor as energy buffer, which we assume to fully recharge after being exhausted. The results below are obtained from 1,000 (10,000) iterations of the benchmarks on the MSP430 (Cortex M3) platform to avoid outliers overly impacting the results. We use dummy data to trigger the execution of the three benchmarks.

5.6.2 Results → Update Size

Figure 5.7 shows the update size we measure during our experiments. When considering Hibernus, the code location where a checkpoint takes place is unpredictable: depending on the capacitor size, an interrupt eventually fires prompting the system to checkpoint. Figure 5.7a, 5.7b, and 5.7c[1] thus report the average update size we measure during an experiment until completion of the workload, as a function of the capacitor size. Compared to the original *copy-all* strategy that mandates the update of the whole

[1] Some data points are missing in the charts for the original design of Hibernus, as it is unable to complete the workload in those conditions. We investigate this aspect further in Section 5.6.3.



(a) *Hibernus (FFT)* **(b)** *Hibernus (RSA)* **(c)** *Hibernus (Dijkstra)*

(d) *MementOS* **(e)** *HarvOS*

Figure 5.7: *Update size comparison.* The size of updates is significantly smaller when using DICE compared to the original system support. Compared to *copy-if-change*, it remains the same or marginally larger.

10 KBytes main memory on NVM, DICE provides orders of magnitude improvements. These gains are naturally a result of limiting updates to those determined by the modification records. On the other hand, *copy-if-change* provides marginal advantages over DICE, because modifications in the call stack are recorded at word-, rather than frame-granularity.

When considering the original design of MementOS or HarvOS as a baseline, the update size is independent of the capacitor size, but a function of the location of the trigger call. At different places in the code, in fact,

a checkpoint may find the stack with different sizes. Figure 5.7d and Figure 5.7e[2] show that DICE reduces the update size to a fraction of that in the original *copy-used* strategy, no matter the location of the trigger call. The same charts show that the performance of *copy-if-change* when combined with MementOS or HarvOS is the same as DICE. This is an effect of the page-level programmability of flash storage, requiring an entire page to be rewritten on NVM even if a small fraction of it is to be updated.

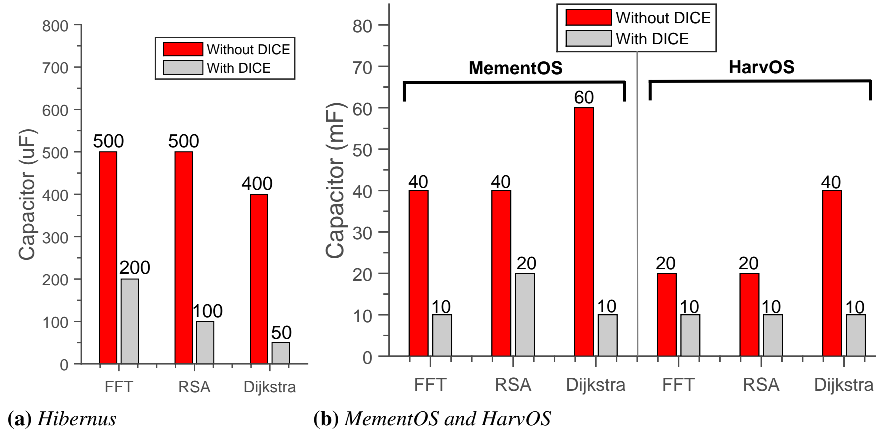
The cost for *copy-if-change* to match or slightly improve the performance of DICE in update size is, however, prohibitive in terms of energy consumption. *Copy-if-change* indeed requires a complete sweep of the checkpoint data before updating. Even for FRAM, arguably the most energy-efficient NVM available as of today, Table 5.1 shows that the cost of read operations is comparable to write operations. As an example, we compute the energy cost of a checkpoint with the update sizes shown in Figure 5.7b to be 93% higher with *copy-if-change* compared with DICE, on average. For Hibernus, *copy-if-change* would then result in an energy efficiency even worse than the original *copy-all* strategy. As energy efficiency is the figure developers are ultimately interested in, we justifiably narrow down our focus to comparing a DICE-equipped system support with the original designs.

5.6.3 Results → Smallest Energy Buffer

Figure 5.8 reports the minimum size of the capacitor required to complete the given workloads. A DICE-equipped system constantly succeeds with smaller capacitors. With Hibernus, DICE allows one to use a capacitor that is up to 88% smaller than the one required with the original *copy-all* strategy. Similarly, for MementOS and HarvOS, the smallest capacitor one may employ is about half the size of the one required in the original designs. Smaller capacitors mean reaching operating voltage faster and smaller device footprints.

Such a result is directly enabled by the reduction in the update size, discussed in Section 5.6.2. With fewer data to write on NVM at every checkpoint, their energy cost reduces proportionally. As a result, the smallest amount of energy the system needs to have available at once to successfully complete the checkpoint reduces as well. Provided the underlying system support correctly identifies when to start the checkpoint, the workload can be completed with a smaller capacitor.

[2]For MementOS, the trigger call locations refer to the "function call" placement strategy in MementOS [171]. We find the performance with other MementOS strategies to be essentially the same. We omit that for brevity.



(a) *Hibernus* **(b)** *MementOS and HarvOS*

Figure 5.8: *Smallest capacitor to complete the workloads.* A DICE-equipped system can complete a fixed workload with smaller energy buffers. This is due to a reduction in the energy cost of checkpoints, enabled by the reduction in update size. Therefore, the smallest amount of energy the system needs to have available at once reduces as well.

5.6.4 Results → Number of Checkpoints

Figure 5.9 reports the percentage reduction in the number of checkpoints at the smallest capacitor size where both the DICE-equipped system and the original design complete the workload, as shown in Figure 5.8. This is arguably the most likely hardware configuration; as we argued, it reduces the time to reach operating voltage and reduces the device footprint [80]. The improvements are significant and apply consistently with different benchmarks. They are relatively smaller in MementOS because of the fixed voltage threshold, which makes it start operating with a larger capacitor than HarvOS.

We detail the improvements against variable capacitor sizes in Figure 5.10[3]. With larger capacitors, the improvements are smaller, even if still appreciable[4]. This is expected: the larger the capacitor, the more the system behaves like a traditional battery-operated one: the application progresses farther on a single charge, checkpoints are sparser in time, while the modifications since the previous checkpoint accumulate as a result of increased processing. The state of main memory then becomes increasingly different than the checkpoint data, and eventually DICE ends up updating a significant part

[3]For MementOS, we tag every data point with the minimum voltage threshold that allows the system to complete the workload, if at all possible, as it would be returned by the repeated emulation runs [172].

[4]Note the log scale on the Y axis of Figure 5.10.

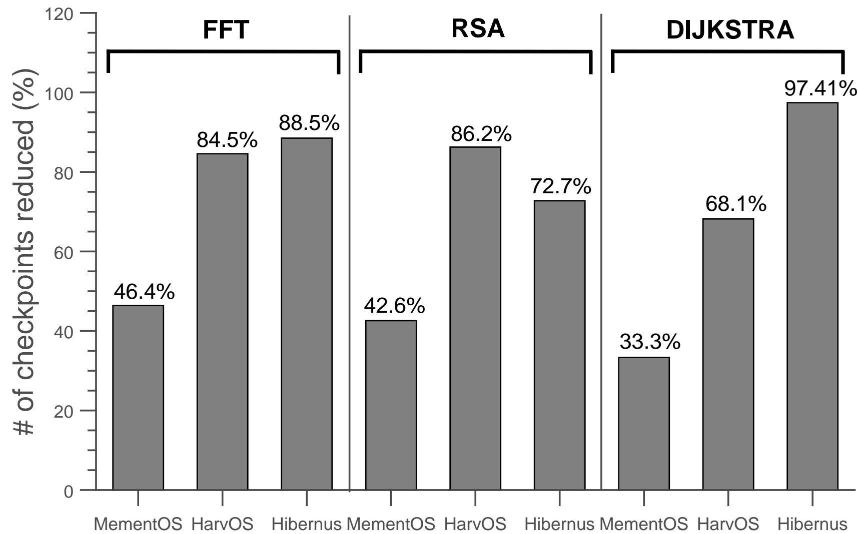
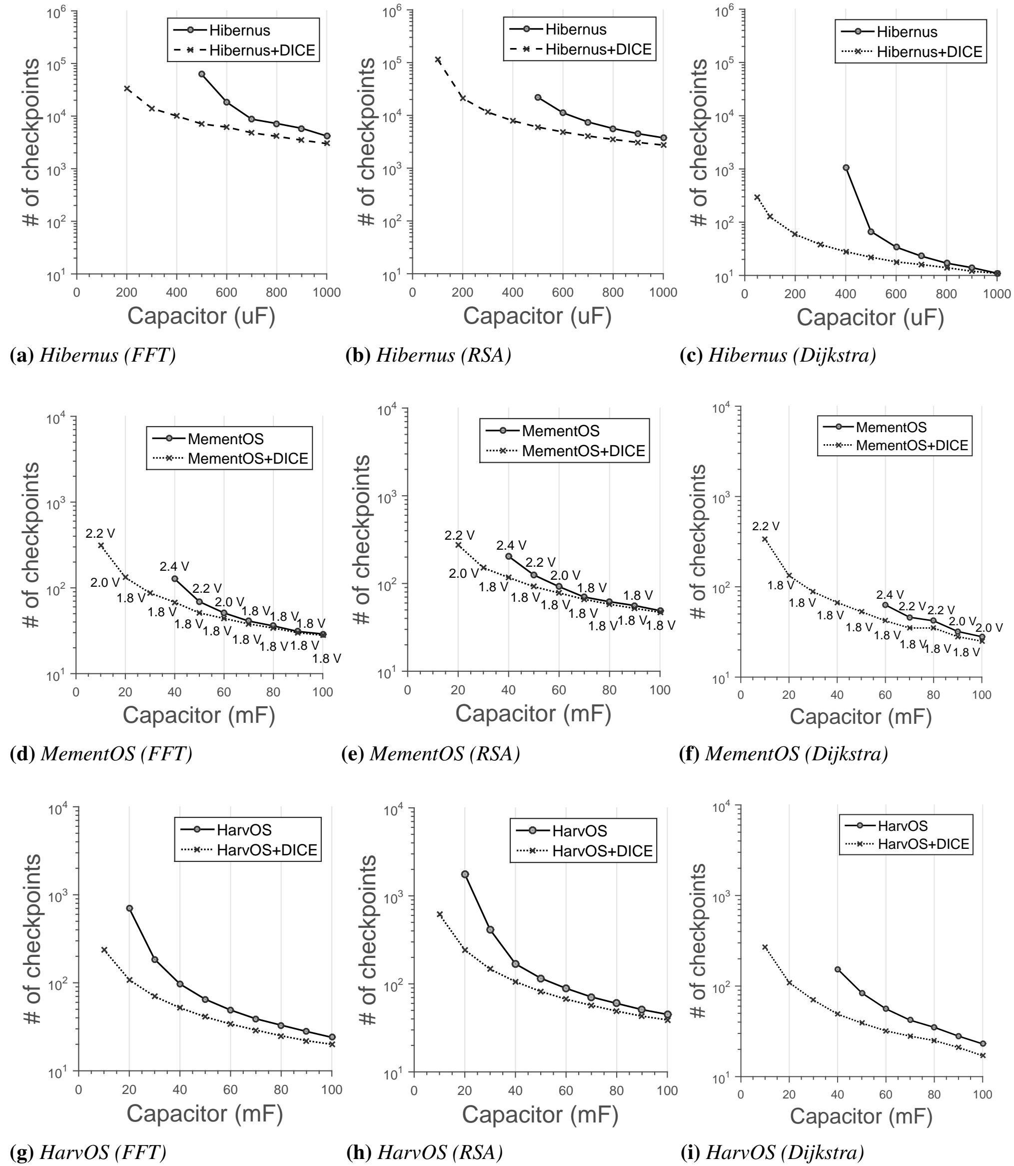


Figure 5.9: *Percentage reduction in the number of checkpoints enabled by* DICE *at the smallest capacitor.* Gains are enabled by DICE across systems and benchmarks, improving the overall energy efficiency as energy is used more for computation than for checkpoints.

of it. In contrast, with smaller capacitors, DICE enables larger improvements, that is, precisely for the kind of setting these systems are designed for. In fact, the more the computation becomes intermittent [81, 132], the more a DICE-equipped system support performs better than its original counterpart.

Figure 5.11 provides a summary view on the results for the number of required checkpoints, plotting the percentage reduction enabled by DICE against variable capacitor sizes and across systems. The curves tend to flatten with larger capacitors for the aforementioned reasons. A significant part of the chart is empty as no comparison is possible, because the workload cannot be completed in the original designs. When a comparison is possible, improvements are largest with smaller capacitors, topping above 80% in most cases.

Overall, reducing the number of checkpoints means energy is spent more in useful computations than in checkpoints. The application can make more progress between periods of energy unavailability, which in turn reduces the time to complete the workload, as we investigate next.

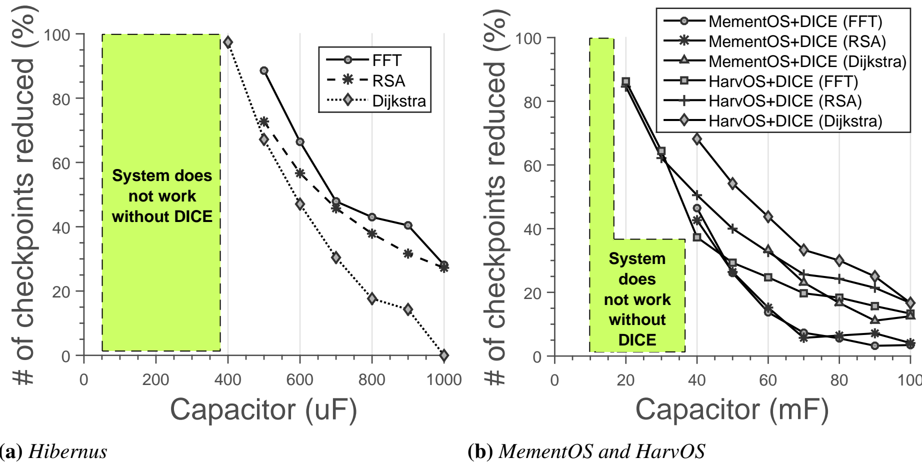


(a) *Hibernus (FFT)* **(b)** *Hibernus (RSA)* **(c)** *Hibernus (Dijkstra)*

(d) *MementOS (FFT)* **(e)** *MementOS (RSA)* **(f)** *MementOS (Dijkstra)*

(g) *HarvOS (FFT)* **(h)** *HarvOS (RSA)* **(i)** *HarvOS (Dijkstra)*

Figure 5.10: *Number of checkpoints necessary against varying capacitor sizes.* Improvements enabled by DICE are smaller with larger capacitors, as the system behavior starts approximating that of a traditional battery-operated one. With smaller capacitors, in contrast, the intermittent execution greatly benefits from DICE.



(a) *Hibernus* **(b)** *MementOS and HarvOS*

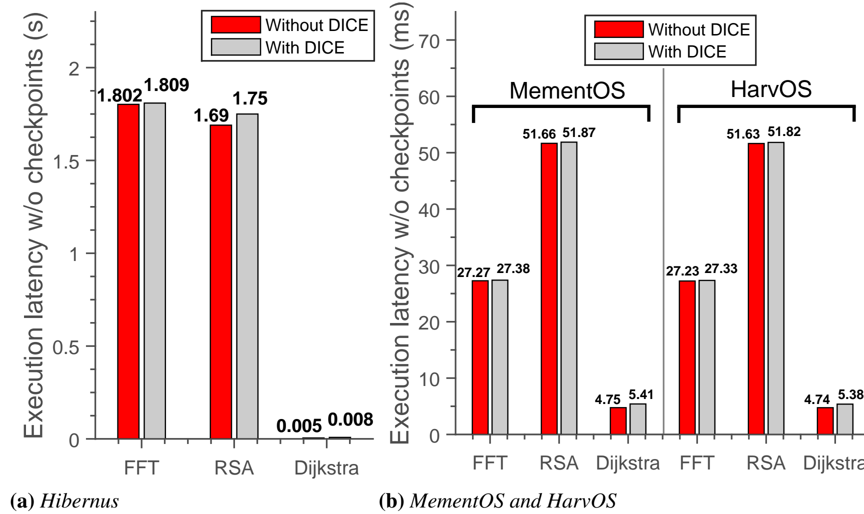
Figure 5.11: *Improvement at different capacitor sizes enabled by* DICE. The improvements are largest with smaller capacitors, which is precisely the kind of setting that system support for transiently-powered computing is designed for.

5.6.5 Results → Execution Latency

DICE naturally imposes a cost for the benefits reported thus far. Such a cost materializes as run-time overhead due to recording differentials, which may increase the time taken to complete a given workload. On the other hand, based on the above results, DICE enables more rapid checkpoints as it reduces the update size, which in turn allows one to reduce their number as energy is spent more on completing the workload than checkpoints. Both factors should rather reduce the execution latency.

Figure 5.12 investigates the additional execution latency in a single iteration of the benchmarks only due to recording differentials: the code executes normally, but we skip the actual checkpoint operations. This way, we can observe the net run-time overhead due to executing **`record()`**. The chart shows that this overhead is generally very limited. This is valid also for reactive checkpoints in Hibernus, despite the conservative approach at placing **`record()`** calls due to the lack of knowledge of where the execution is preempted.

Figure 5.13 takes into account the time required for the number of checkpoint operations at the smallest capacitor where both the DICE-equipped system support and the original design complete the workload, as indicated in Figure 5.8. The overhead due to executing **`record()`** calls is not only compensated, but actually overturn by the gains enabled by the reduction



(a) *Hibernus* **(b)** *MementOS and HarvOS*

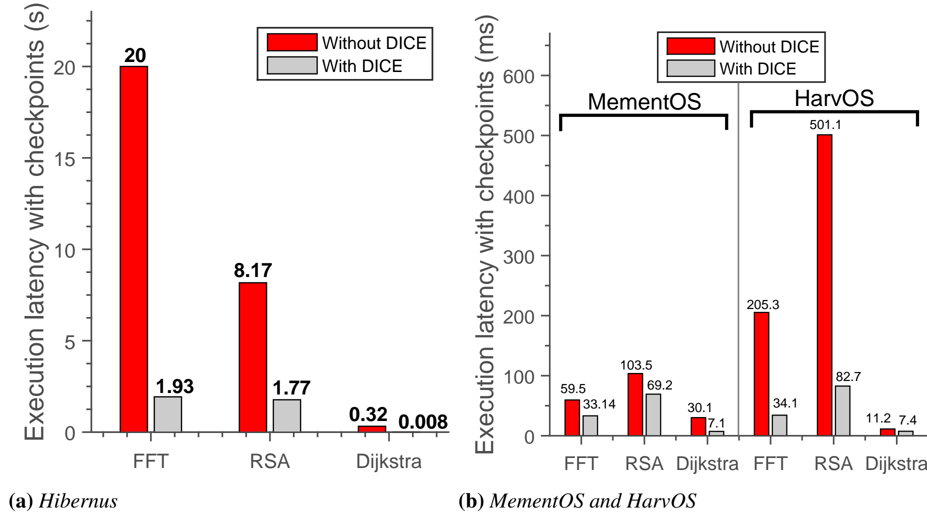
Figure 5.12: *Execution latency without concrete checkpoints.* The additional run-time overhead due to DICE code instrumentation is limited.

of update size, which in turn yields fewer, more rapid checkpoints. Using these configurations, DICE allows the system to complete the workload much earlier.

Differently, Figure 5.14 provides an example of the trends in execution latency against variable capacitor sizes. The improvements are significant with smaller capacitors, again precisely in the intermittent computing setting we target. However, two factors contribute to the curves in Figure 5.14 approaching each other and flattening as the system configuration increasingly resembles a traditional battery-operated device. Larger capacitors allow a benchmark to make more progress on a single charge, so the number of required checkpoints reduces. On the other hand, as more processing happens between subsequent checkpoints, more modifications occur in main memory, forcing DICE to update a larger portion of the checkpoint data. Eventually, these two factors compensate each other, and the time behavior of a DICE-equipped system approaches the one of the original system support.

5.7 Related Work

We can effectively divide the existing literature on system support for transiently-powered computing in two classes.



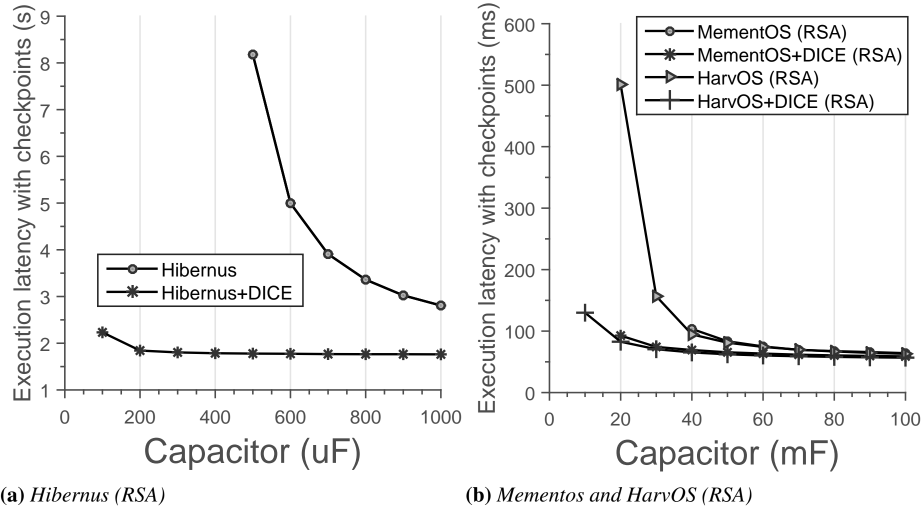
(a) *Hibernus* **(b)** *MementOS and HarvOS*

Figure 5.13: *Complete execution latency at the smallest capacitor size.* The run-time overhead due to DICE is overturn by great gains due to reducing size and number of checkpoints, ultimately resulting in a reduction of execution latency.

The first class essentially includes checkpoint-based solutions that may benefit from DICE as a complement, such as Hibernus [21], Hibernus++ [22] Mementos [172], and HarvOS [29]. These target mainstream device architectures employing a volatile main memory for efficient processing, and external storage as NVM. DICE complements them by reducing the time and energy overhead of checkpoints. Closest to our work in this area is the *copy-if-change* strategy [28], described in Section 5.7 and demonstrated in Section 6.5 to provide much worse energy performance than DICE.

The second class of solutions [93, 131, 214] targets device architectures that employ non-volatile processors [200] or non-volatile main memory [80]. The former relieve the system from checkpoints altogether, yet require dedicated processor designs still far from massive production. The latter still needs to checkpoint MCU registers. In both kinds of solutions, reduced reliance on checkpoints is traded for increased energy consumption during normal computations, due to the use of FRAM as main memory.

Compared to the solutions targeting mainstream device architectures, this class of solution therefore requires further investigation to understand what are the conditions—for example, in terms of energy provisioning patterns—where the trade-off above plays favorably. Nevertheless, additional challenges also arise, including how to handle data consistency is-



(a) *Hibernus (RSA)*

(b) *Mementos and HarvOS (RSA)*

Figure 5.14: *Execution latency against variable-size capacitors.* DICE-enabled gains are higher in an intermittent computing setting. Larger capacitors approximate the behavior of traditional batteries, and DICE provides little benefit.

sues [44] that could lead to incorrect executions [214], and what programming models are best fit to such computing platforms [131]. Addressing the latter challenge most often requires programmers to learn new programming abstractions and language constructs [44,131], which may slow down adoption, as opposed to the first class of solutions that mostly operate transparently from programmers.

In the same way as the solutions we complement [22,29,172], we focus on supporting transiently-powered computing for the MCU. Other device components, such as sensors or radios, may operate through separate energy buffers [79] or techniques such as radio backscattering [124]. The former effectively decouples the energy management of peripherals from that of the MCU. The latter enables networking through energy-neutral operations. Dedicated solutions for checkpointing peripheral states also exist [64, 132], and are likely to integrate with a DICE-equipped system support as much as with the original designs.

5.8 Summary

DICE is a set of compile-time techniques for transiently-powered computers to reduce the amount of data written on NVM during checkpoints. To

this end, we conceive different ways to track changes in main memory, depending on what memory segment these reside on, as well as lightweight code instrumentation strategies with dedicated customization depending on how checkpoints are possibly triggered. DICE helps existing system support to complete a given workload with *i)* smaller energy-buffers, *ii)* fewer checkpoints, and thus better energy efficiency, and *iii)* reduced execution latency. Our experimental evaluation, based on a combination of three benchmarks across three different existing system support and two different hardware platforms, provides quantitative evidence. For example, using DICE, HarvOS can complete the execution of the RSA algorithm with 86% fewer checkpoints, resulting in better overall energy utilization and a 34% reduction in execution latency.

CHAPTER 6

"*When and Where?*" : HarvOS

In this chapter, we present code instrumentation strategies to allow transiently-powered embedded sensing devices efficiently checkpoint the system's state before energy is exhausted. Our solution, called HARVOS, operates at compile-time with limited developer intervention based on the control-flow graph of a program, while adapting to varying levels of remaining energy and possible program executions at run-time. In addition, the underlying design rationale allows the system to spare the energy-intensive probing of the energy buffer whenever possible. Compared to existing approaches, our evaluation indicates that HARVOS allows transiently-powered devices to complete a given workload with 68% fewer checkpoints, on average. Moreover, our performance in the number of required checkpoints rests only 19% far from that of an "oracle" that represents an *ideal* solution, yet unfeasible in practice, that knows *exactly* the last point in time when to checkpoint. This chapter is published as *"HarvOS: Efficient code instrumentation for transiently-powered embedded sensing"* in [29].

6.1 Introduction

In chapter 4, we explained the design of an efficient checkpointing mechanism which enable transiently-powered embedded devices to efficiently checkpoint the system's state on non-volatile memory [28, 171] whenever energy is about to be exhausted. This allows a device to resume operation from the saved state as soon as energy is newly available. *Where* and *when* to perform the checkpoint, which is an energy-expensive operation per se, is crucial. Doing so too early would essentially correspond to a waste of energy that could be usefully employed in further computations. In contrast, excessively postponing a checkpoint may yield a situation where insufficient energy is left to complete the operation. Because of the unpredictable supply of energy from the environment and the varying run-time execution of programs, striking an efficient trade-off is challenging.

In this chapter, we present code instrumentation strategies to place calls to *trigger* functions that, based on the current system state, decide whether to perform the checkpoint before continuing the execution [171]. Different from existing approaches, we look at the control-flow graph (CFG) of a program and place triggers according to different strategies depending on the programming constructs, for example, branching statements as opposed to loops. Simultaneously, we aim at reducing the size of the checkpoint itself by placing triggers where the size of the allocated memory is reduced. The decision on whether to checkpoint is based on available energy as well as the worst-case estimation of the energy required to reach the next trigger call.

Such a scheme, which we call HARVOS, completely operates at compile-time and dynamically adapts to varying levels of remaining energy at run-time, while capturing the actual program execution through the CFG. The underlying design rationale also allows the system to spare energy-intensive probing of the energy buffer, for example, through ADCs, whenever possible.

HARVOS is largely independent of programming language, OS, and underlying platform. It is generally applicable to imperative programming languages. The execution of checkpoints is transparent to the OS as long as a way to make these operations atomic is somehow provided. Our solution applies both to platforms where traditional volatile memory is employed for normal processing and a separate non-volatile memory is reserved for checkpoints, and to platforms where non-volatile memory is used in place of volatile one; for example, when FRAM chips replace normal SRAM chips.

Our evaluation considers modern 32-bit MCUs and three increasingly complex benchmark codes commonly employed in embedded sensing. We execute the programs mutiple times against varying size of the underlying energy buffers to measure the performance of HARVOS against existing approaches. The results we collect indicate that, for example, HARVOS allows a device to complete a given workload with 68% fewer checkpoints, on average compared to existing approaches. Moreover, such a performance rests 19% far from that of an "oracle" that represents an *ideal* solution, yet unfeasible in practice, that knows *exactly* the last point in time when a checkpoint is required.

The benefits are not, however, limited to the number of required checkpoints. Our evaluation also shows that, because checkpoints in HARVOS happen much closer to the last practical point in time when the system should take a checkpoint, we can also reduce the processing that would go wasted as its results would not become part of any checkpoint. We further demonstrate that, unlike existing approaches, our performance is largely robust against different program structures. Ultimately, this means that energy utilization is improved in a larger set of applications, as it is employed more for useful computations than for checkpointing.

The rest of the chapter unfolds as follows. Section 6.2 places our work in context. Section 6.3 describes the design rationale and the foundations of our approach. Section 6.4 describes the compile-time rules we apply to decide on the placement of trigger calls depending on the program structure. Experimental results are reported in Section 6.5. We end the paper with brief concluding remarks in Section 6.6.

6.2 Related Work

Relatively little research exists on enabling transiently-powered computing on embedded devices. Recent work comprehensively describes existing approaches and quantitatively compares them against each other [19]. Here we focus on the aspects most relevant for code instrumentation. We recognize two classes of such approaches.

One class is based on separate memory areas for normal computations and for checkpointing. Examples are MementOS [171] and Hibernus [21]. The MementOS prototype uses flash memory for checkpointing. Trigger calls are executed periodically or placed using a *loop-latch* or *function-return* strategy. The former places trigger calls at the end of loop iterations; the latter places trigger calls at function return points. These are the locations where one may expect the stack to store less data, which would then

reduce the size and energy cost of moving data to flash. The decision to checkpoint is based on a voltage threshold obtained through repeated emulation experiments that eventually determine a single program-wide threshold based on average run-time behavior and user-supplied energy traces.

Hibernus [21] uses FRAM instead of flash memory, and triggers a checkpoint based on a hardware interrupt firing if the operating voltage drops below a threshold. Because of the higher energy efficiency of FRAM compared to flash memory, the latter can be statically defined because Hibernus can afford to copy the entire RAM segment independent of the current memory occupation. The energy to perform such a fixed-cost checkpoint is stored in a separate decoupling capacitor, in turn driven by an external voltage regulator. In contrast, one of our goals is to enable efficient checkpointing without requiring hardware modification. Furthermore, FRAM chips are still limited in overall size. It is then difficult to store multiple checkpoints; for example, to ensure that at least a complete consistent checkpoint is always available.

The other class of solutions employ non-volatile memory, especially FRAM, as the only memory space. This means FRAM is used both for normal computations and for saving the system's state in periods of energy unavailability. The advantage is that application data already resides on non-volatile memory, so only registers and program counter need to be saved when checkpointing. QuickRecall [93] is an example in this class. These solutions are especially indicated for scenarios characterized by very short energy bursts, as checkpoints can happen quickly. However, they suffer from an increase of energy consumption during normal computations due to the use of FRAM in place of SRAM, and from potential data consistency issues that require specialized compiler techniques [131].

HARVOS is independent of the underlying memory architecture, and applies to both classes of approaches with only minor changes. The trigger placement rules we describe next may replace or complement the heuristics or periodic trigger calls employed in the aforementioned systems. Our design is rooted in the unbalance between normal computation and the energy-hungry operation of checkpointing, and seeks to reduce the overhead of the latter.

6.3 Overview

Calls to trigger functions placed anywhere in the code essentially represent an overhead compared to the normal computation. In existing systems, two operations are performed every time the execution encounters a trigger

call. First, the system verifies some condition that indicates whether it is time to checkpoint. MementOS, for example, uses a voltage threshold as explained in Section 6.2. If the condition is verified, the checkpoint takes place. The energy cost of checking whether a checkpoint is necessary is normally constant.

The energy cost of the actual checkpoint, on the other hand, depends on the underlying memory architecture. For platforms that only employ a single non-volatile memory area [93], the size of checkpoints is fixed and independent of where the checkpoint takes place throughout the program execution: only registers and program counter need to be saved. Thus, the energy cost of checkpointing is fixed.

In platforms employing separate memory areas for normal computations and for checkpointing [21, 171], the entire allocated memory needs to be saved, including stack and heap, at the time of checkpointing. As a result, the size and therefore the energy cost of checkpointing depend on where in the program the checkpoint takes place, making this energy cost typically proportional to the size of the allocated memory [28]. For example, the higher the stack at that point in the execution, the larger is the energy cost of checkpointing.

6.3.1 Challenge

Our objective is to minimize the energy overhead due to checkpointing operations. This means *i)* to minimize the number of trigger calls that are uselessly executed, that is, to verify no checkpoint is needed, and *ii)* to postpone the actual checkpoint to a moment where the available energy is strictly sufficient to that end, that is, one can not perform further computations without jeopardizing the ability to checkpoint later.

The two needs are at odds with each other. Postponing the checkpoint, in fact, requires to frequently check how close is the execution to when no sufficient energy is left to perform the checkpoint. However, trigger calls need to probe the energy buffer, for example, through ADC operations. Therefore, frequently performing this operation may become prohibitive because of the energy cost. The negative effects are not limited to energy consumption. Trigger calls might, in addition, change the execution timings. Using resource-constrained devices, this may introduce subtle software bugs [230].

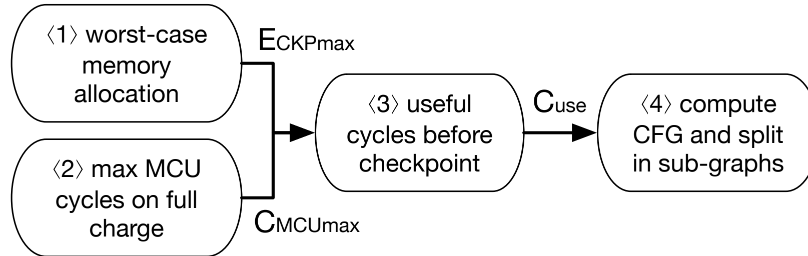


Figure 6.1: *Compile-time operation of* HARVOS.

6.3.2 Rationale

To optimize the point in time when the actual checkpoint takes places and its energy cost, we rely on compile-time information on memory allocation patterns. Static code analysis techniques exist that can return accurate information on the evolution of the stack and, in many cases, of the heap as well [10, 83, 84]. The latter techniques especially apply when the size of heap-allocated data structures is known at compile-time; for example, whenever objects are dynamically allocated in languages such as C++.

Similar to existing works [21,93,171], we focus on supporting transiently-powered computing for the main MCU. Other components on the device, such as sensors or radios, may operate through separate energy buffers [79] or techniques such as radio backscattering [124]. The former technique effectively decouples the energy management of peripherals from that of the MCU, which is in charge of driving the entire system and thus requires ad-hoc techniques to operate across periods of energy unavailability. The latter techniques enable networking among embedded devices and between embedded devices and surroundings infrastructure through energy-neutral operations. These ensure that the amount of energy consumed for transmissions does not exceed the harvested RF energy.

6.3.3 Operation

Figure 6.1 illustrates the compile-time operation of HARVOS. Given a program, at step ⟨1⟩ we estimate the worst-case memory usage throughout the code and use this to obtain an estimate of the highest energy cost E_{CKPmax} for checkpointing at any point in a program's execution. The latter step is simple; for example, as the energy consumption of flash chips obeys to specific trends dictated by the manufacturing characteristics.

In case the underlying platform employs non-volatile memory also for normal computations, the energy cost E_{CKPmax} is constant; application data already resides on stable storage, and only the content of registers

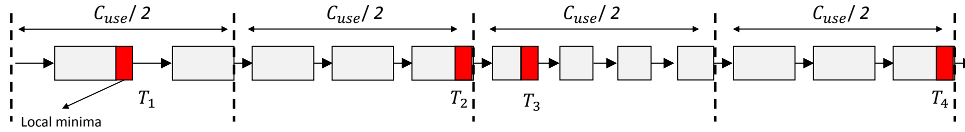


Figure 6.2: *Splitting the CFG in sub-graphs whose required number of MCU cycles is at most* $C_{use}/2$. The picture considers a linear CFG for simplicity.

and program counter needs to be saved. In the following, we consider the more complex case of variable energy cost for checkpointing operations, germane to platforms that employ separate memory areas for normal computations and for checkpointing.

At step ⟨2⟩, we calculate the maximum number of cycles C_{MCUmax} we can execute whenever the device wakes up with a freshly charged energy buffer supplying energy $E_{wake-up}$. Many transiently-powered devices include a wake-up circuit that boots the device when the voltage level of the on-board capacitor surpasses a certain threshold. Knowing this value, computing C_{MCUmax} is also simple, according to an MCU's datasheet. In doing so, we consider the maximum power consumption of the MCU. This likely underestimates the value of C_{MCUmax}, and yet allows us to reason in a worst-case setting that shields us from unexpected power failures.

Steps ⟨1⟩ and ⟨2⟩ above are independent of each other. Their outputs are fed as input to step ⟨3⟩ where we compute the number of *useful* cycles C_{use} the MCU can execute in a worst-case scenario where: *i)* the device starts afresh with energy $E_{wake-up}$, *ii)* it does not receive any additional energy contribution from the environment afterwards, and *iii)* it needs to spend E_{CKPmax} right before dying to checkpoint the system's state. The number C_{use} of cycles are therefore those the MCU can execute with an amount of energy $E_{wake-up} - E_{CKPmax}$, and can be computed similarly to step ⟨2⟩. These cycles are, in practice, those the MCU can execute to make progress in the program.

Next, in step ⟨4⟩ we compute the CFG of the program and associate every block in the graph to the number of cycles required to execute it with the target MCU. A combination of mature code inspection and emulation tools, such as Understand (`www.scitools.com`), Emul8 (`emul8.org`), and Kiel uVision (`www2.keil.com`) can be used to this end. We then split the CFG in sub-graphs whose total stretch in number of cycles is at most $C_{use}/2$, as intuitively shown in Figure 6.2. The picture considers a linear CFG for simplicity; we discuss the general case next.

Within each sub-graph, we identify the block corresponding to the mini-

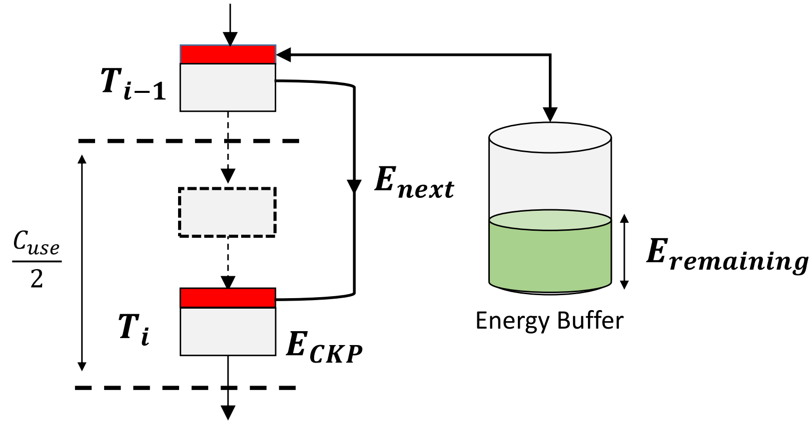


Figure 6.3: *Decision logic to take a checkpoint.* At the T_{i-1}-th trigger call, the system checks if sufficient energy remains to reach the next trigger call at T_i and to checkpoint at T_i. If so, the execution continues. If not, a checkpoint takes place at T_{i-1}.

mum size of allocated memory, and place a trigger call right at the end of it. This means we aim at possibly checkpointing the system's state whenever the cost of the checkpoint operation is reduced, as the amount of data to copy over stable storage is minimal within a sub-graph. In doing so, we do not add any instrumentation other than the trigger calls themselves. As a result of these procedures, provided the code can execute entirely on a single charge, that is, the C_{use} cycles are sufficient to cover the entire execution, no trigger calls are placed anywhere in the code, which remains unaltered.

Step $\langle 4 \rangle$ explained above is crucial in the general case. Because of our placement strategy, the maximum number of MCU cycles separating any two trigger calls, for example, T_3 and T_4 in Figure 6.2, is bound to be less than C_{use}. The extreme case is where the T_{i-1}-th call is at the start of a sub-graph and T_i-th call is at the end of following sub-graph; in this case as well, the cycle distance is at most C_{use}. Thus, even in a worst-case situation, a device that starts afresh with energy $E_{wake-up}$ from the location of a previous checkpoint can reach the next trigger call with sufficient energy to complete the checkpoint before dying again.

As a result of the placement logic and based on the information collected up to step $\langle 4 \rangle$, at every trigger call the system can take an informed decision on whether to checkpoint. Say E_{next} is the energy to execute the required MCU cycles from the T_{i-1}-th call to the T_i-th call, whereas $E_{CKP(i)}$ is the energy required to checkpoint the system's state against the size of the allocated memory at the T_i-th call, as intuitively depicted in Figure 6.3. A

checkpoint at the T_{i-1}-th call is required if

$$E_{remaining} \leq E_{next} + E_{CKP(i)} \tag{6.1}$$

where $E_{remaining}$ is the energy left in the buffer when executing the T_{i-1}-th trigger call.

The condition in equation (6.1) essentially checks if the remaining energy is sufficient to reach the next trigger call *and* to checkpoint there. This reasoning assumes that the environment provisions no additional energy between T_{i-1} and T_i, that is, we are operating in a worst-case situation in terms of energy provisioning. At run-time, we can obtain the value of $E_{remaining}$ through software-based techniques [33, 198] or hardware solutions [59, 149] with negligible overhead.

The ability to reason on whether the system can reach the *next* trigger call is one of the key of traits of our approach, and a major source of improvements compared to previous work, as we discuss in Section 6.5.

6.3.4 Generalization

CFGs are generally not linear as the example of Figure 6.2. On the contrary, they show complex structures reflecting the variety of available programming constructs, such as branching statements, loops, and function calls. This means there may be multiple places in a sub-graph corresponding to the minimum allocated memory, as a function of different execution paths. Moreover, embedded devices often operate in an interrupt-driven manner, that is, the execution through a CFG may be arbitrarily preempted and temporarily re-directed through the CFG of interrupt handlers. The latter case does not appear to be taken into explicit account in existing systems [21, 171].

To address these issues, the next section describes a set of trigger placement rules that, depending on the program structure, dictate where to place the trigger call and what to consider as the E_{next} energy to reach the next trigger call. We identify *branches*, *loops*, *function calls* and *interrupt handlers* as the fundamental structures of the CFG, and design specific rules for them. The complete set of rules is *recursively* applied until an elementary block in the CFG is reached. The rules also determine the conditions when probing the energy buffer for the value of $E_{remaining}$ is strictly needed, or $E_{remaining}$ can be inferred from compile-time information. The latter situation allows the system to spare operations that may be energy-expensive per se, such as probing ADCs.

6.4 Placement Rules

We illustrate the set of rules used to place trigger calls for arbitrary program structures. In conceiving these rules, our reasoning is based on a worst-case analysis among the possible program executions. In addition, interrupt handlers require special care, as it is generally impossible to predict the point in time when they preempt the execution.

6.4.1 Branching

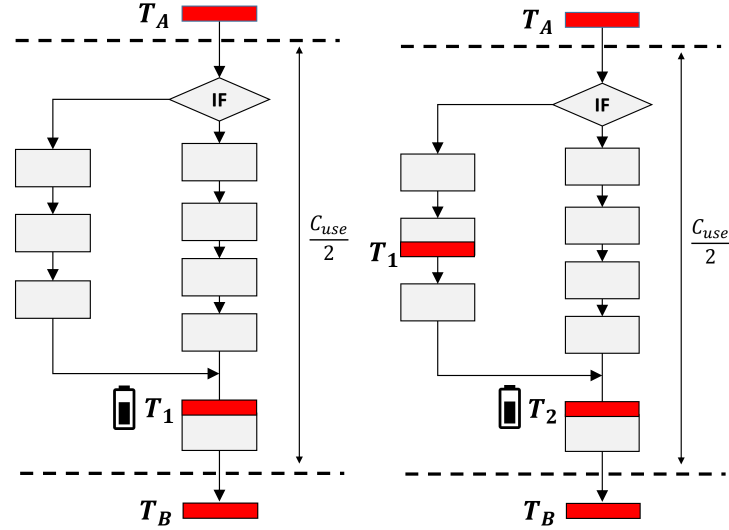
The challenge here is to account for the lack of compile-time information on what path is taken at run-time. To address this, one may decide to instrument the code to trace the actual execution. Doing so, however, would add further overhead and greatly complicate the instrumentation strategy; as the complexity of the code grows, the number of possible paths increases exponentially. We rather adopt a worst-case approach and avoid any further code instrumentation besides the trigger calls. We demonstrate in Section 6.5 that this is not necessarily detrimental to performance.

In general, what rule to apply depends on whether branching is fully included in a single sub-graph or not.

Branching in a single sub-graph Figure 6.4 shows the situation. We call T_A (T_B) the last (first) trigger call in the preceding (following) sub-graph. We consider two cases:

1. The minimum amount of allocated memory is outside the branching statement, for example, at location T_1 in Figure 6.4a. If so, at trigger call T_A we consider the E_{next} value corresponding to the most energy-consuming branch. This means that, if the environment provisions no additional energy since T_A and the least energy-consuming branch is taken, at T_1 we are going to find a higher value for $E_{remaining}$ than we expect. This is the reason why at T_1 we are forced to probe the energy buffer to find out the exact value for $E_{remaining}$ before taking a decision to checkpoint.

2. The minimum amount of allocated memory is found in either of the two branches, say at location T_1 in Figure 6.4b. According to Section 6.3, we place a trigger call at T_1. However, an issue arises if the *other* path is taken, where no trigger calls are placed. To cater for this, we place a trigger call right outside the branching statement, say at location T_2 in Figure 6.4b. The E_{next} value at trigger call T_A is set to the most energy-consuming path between those leading to either T_1 or



(a) *The location of minimum allocated memory is found outside of the branching statement.*

(b) *The location of minimum allocated memory is found in either of the two branches.*

Figure 6.4: *Placement rules for branching statements fully included in a single sub-graph.* The battery icon indicates where probing the energy buffer is mandatory.

T_2. At T_1 we can spare probing the energy buffer because, provided only one execution path exists, the energy necessary from T_A to T_1 is fixed. Differently, we still need to probe the energy buffer at T_2. In fact, without additional instrumentation, we cannot determine what path is taken at run-time.

Branching across multiple sub-graphs Figure 6.5 illustrates a general example. Following the sub-graph where the last trigger call is T_A, different sub-graphs may correspond to different paths. We may thus identify a block in the CFG with the minimum amount of allocated memory in every involved sub-graph. The E_{next} value at trigger call T_A is then set to the most energy-consuming path between the one leading to either T_1 or T_2. If a single execution path exists, the trigger call along this path does not require probing again the energy buffer, as the energy required from T_A up to the trigger call is fixed. The same does not hold for the path requiring the least energy.

Based on a similar reasoning, the E_{next} value at trigger call T_1 or T_2 is set to the energy required to reach T_3. At the latter, however, we necessarily need to probe the energy buffer; again, we cannot determine what

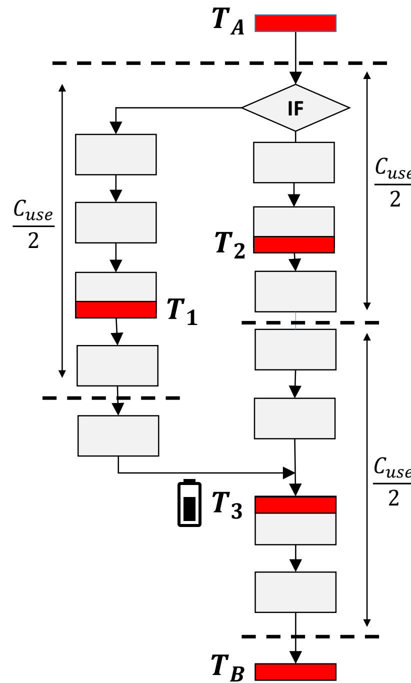


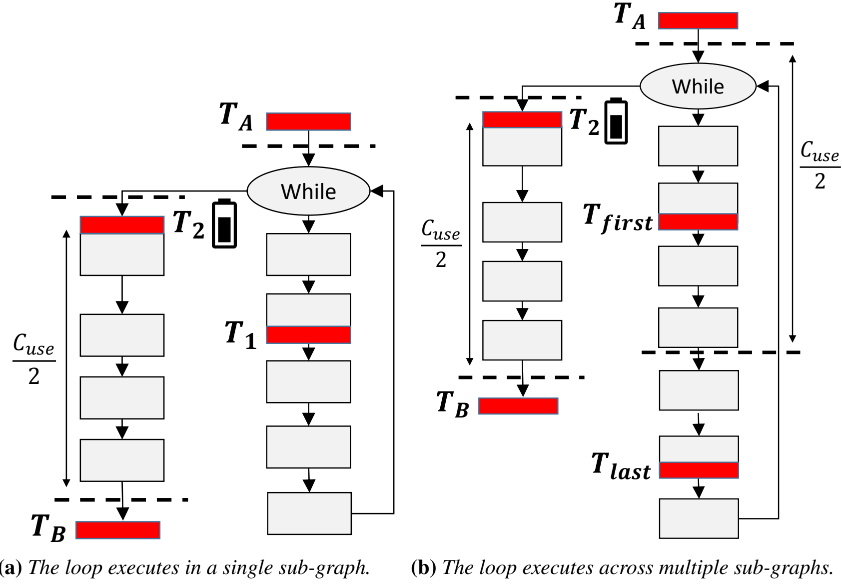
Figure 6.5: *Placement rules for branching statements executing across multiple sub-graphs.* The battery icon indicates where probing the energy buffer is mandatory.

path the execution is coming from. Cases may exist where combining these rules recursively might introduce a slight overhead. For example, if multiple branch statements are nested inside each other, all of them are forced to probe the energy buffer at the end. A single probe operation may be, however, sufficient.

6.4.2 Loops

When the number of iterations is known or can be statically determined, placing trigger calls when a sub-graph includes loop statements is not an issue. One may first exercise existing loop unrolling schemes [50]; then apply the remainder of the techniques in this section to the resulting CFG.

Whenever the number of iterations is determined by run-time information, however, we are faced with two challenges. First, we need to decide whether the last (or single) trigger call inside the loop should consider as the E_{next} value the energy to reach the first trigger call *inside* the same loop, that is, the loop continues with the next iteration, or rather the energy to reach the first trigger call *outside* the loop, that is, the execution exits



(a) *The loop executes in a single sub-graph.* **(b)** *The loop executes across multiple sub-graphs.*

Figure 6.6: *Placement rules for loops.* The battery icon indicates where probing the energy buffer is mandatory.

the loop. Second, whenever the latter happens, it is impossible to know how many iterations were executed without additional instrumentation, for example, in the form of counters, which would increase the overhead.

The lack of run-time information at compile-time prevents us from taking an accurate decision here. However, loops that depend on run-time information are likely to yield some number of iterations. Again, we need to distinguish whether the loop is fully included in a single sub-graph or not.

Loop in a single sub-graph Figure 6.6a illustrates the situation. Consider T_A (T_B) is the last (first) trigger call in the preceding (following) sub-graph. Inside the loop, T_1 indicates the trigger call corresponding to the location of minimum allocated memory.

The E_{next} value at T_A is set to the energy to reach T_1, which is fixed if a single execution path exists; therefore, the trigger call at T_1 may not need to probe the energy buffer. This already implicitly considers the case that the execution enters the loop at least once, which may be considered as the most frequent case.

Because of the above observation, at T_1 we consider the E_{next} value to reach the next trigger call as the one corresponding to the execution contin-

uing with the next iteration. In Figure 6.6a, this corresponds to the energy to execute from T_1 back to T_1. In this case, if the code inside the loop shows a single execution path, we can spare probing the energy buffer, as the energy to reach T_1 from T_1 itself is fixed.

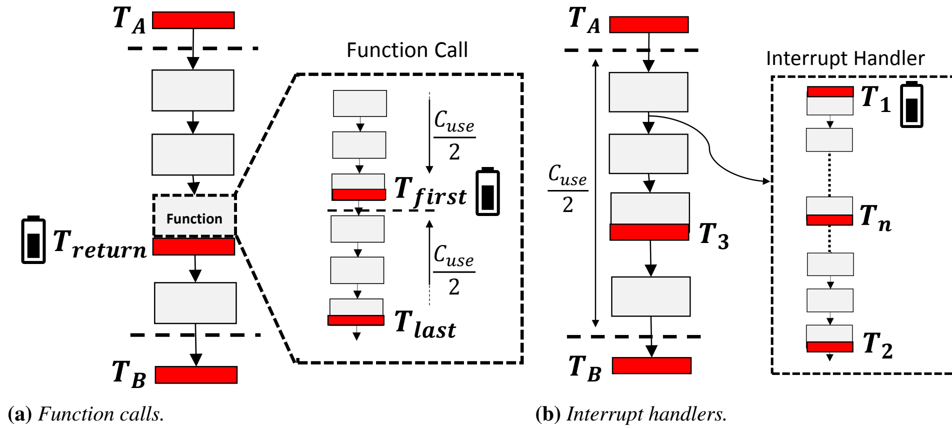
Note that the other option here would be to consider the E_{next} value corresponding to the execution exiting the loop, that is, the energy necessary to reach the first trigger call outside the loop. If this was higher than the one to reach T_1 again, the system would likely over-checkpoint without any need. Every time the execution reaches T_1 with little energy, the system would detect there is no sufficient energy to reach the first trigger call outside of the loop and it would checkpoint. However, the energy may be sufficient for another iteration of the loop if the individual iterations are less energy-expensive than reaching the first trigger call outside of the loop. We expect this to be the most frequent case.

Finally, we place another trigger call, indicated with T_2 in Figure 6.6a, right outside the loop. This is necessary because we have no information on how many loop iterations executed before exiting. The last time the trigger call at T_1 is executed before the loop breaks may not checkpoint, as the system thinks one more iteration is possible. As this reasoning is no longer applicable when the execution exits the loop, at T_2 we are forced to probe the energy buffer to possibly checkpoint. The E_{next} value at T_2 considers the energy to reach T_B, that is, the first trigger call in the following sub-graph.

Loops across multiple sub-graphs Figure 6.6b illustrates a general example. We handle this case as an extension of the previous one, with the added complexity that multiple trigger calls are necessarily included in the loop body because it spans multiple sub-graphs.

In this case, the last trigger call in the previous sub-graph considers the E_{next} value to reach the first trigger call inside the loop, indicated with T_{first} in Figure 6.6b. Inside the loop body, the last trigger call, that is, T_{last} in Figure 6.6b, considers the energy value E_{next} corresponding to continuing with a further loop iteration, which leads to T_{first}. Again in the case of a single execution path, the E_{next} value at T_{last} to reach T_{first} is fixed, so we can spare probing the energy buffer. This again considers the case of the loop continuing with the following iteration as the most probable.

Right outside the loop, we still need to place a further trigger call T_2, required for the exact same reasons as the case of Figure 6.6a.



(a) *Function calls.* **(b)** *Interrupt handlers.*

Figure 6.7: *Placement rules for function calls and interrupt handlers.* The battery icon indicates where probing the energy buffer is mandatory.

6.4.3 Function Calls and Interrupt Handlers

Placement rules for function calls and interrupt handlers follow a similar rationale as the previous cases.

Function calls Whenever a given execution path includes a function call, the situation is equivalent to "inlining" the CFG of the function within the CFG of the caller, as shown in Figure 6.7a.

If the execution of a function call spans multiple sub-graphs, at least one trigger call is placed at a local minimum of allocated memory inside the function. The energy value E_{next} at the last trigger call in the previous sub-graph considers the energy to reach the first trigger call inside the function, that is, T_{first} in Figure 6.7a.

The issue here is that same function may be called at multiple places in the code. Without resorting to additional code instrumentation, it is impossible to differentiate these cases. Therefore, at T_{first} we necessarily must probe the energy buffer, as the function's execution is unaware of where the caller code issued the call; thus, we cannot know uniquely what is the amount of energy spent since an arbitrary T_A.

Because of the same reason, we need to place a further trigger call right after the function returns, indicated with T_{return} in Figure 6.7a. Chances are that the local minimum in allocated memory within the sub-graph is found right when the function returns, that is, the time when the function's activation record—including the memory allocated for local variables and to resume execution of the main code—is de-allocated. The E_{next} value

at the last trigger call inside a function considers the energy spent to reach T_{return}.

Interrupt handlers The case of interrupt handlers adds another challenge to those for function calls. Without additional instrumentation, interrupt handlers have no information on what was the situation in the execution the handler preempted. Consider Figure 6.7b as an example. The trigger call at T_A decides not to checkpoint as the condition in equation (6.1) indicates the execution may reach T_3. An interrupt handler preempts the execution between T_A and T_3. Coping with this case requires two rules in addition to those normally applied to the code in the interrupt handler:

1. When it starts, the interrupt handler has no information on what decision was taken at T_A. Therefore, we place a trigger call right at the beginning of the interrupt handler to verify that, based on remaining energy, the next trigger call within the interrupt handler is reachable. Probing the energy buffer is thus mandatory at T_1.

2. When the interrupt handler finishes, it has no information on where the next trigger call is located in the main code. It may happen that the execution at T_A was actually expecting to checkpoint at T_3, as we predicted we could reach T_3 with just the right amount of energy. As the interrupt handler consumes some energy per se, T_3 may be unreachable now, and we need to checkpoint before returning to the main code. To capture these situations, every trigger call raises a flag if it expects to checkpoint at the immediately following trigger call. In this example, T_A would raise the flag. The flag prompts an additional trigger call at the end of the interrupt handler, shown as T_2 in Figure 6.7b to checkpoint.

Finally, we need a second flag to indicate that the main code was preempted. The trigger call at T_3 may be one that does not require probing the energy buffer, according to normal placement rules. This decision must be superseded if an interrupt handler executed in between, whose energy cost is generally not known.

6.5 Evaluation

We assess the effectiveness of HARVOS along multiple dimensions. In the following, Section 6.5.1 describes the experimental settings, whereas Section 6.5.2 reports on the results. Our key findings are summarized as follows:

- HARVOS allows transiently-powered devices to complete a fixed workload with with 68% fewer checkpoints, on average compared to existing approaches;

- HARVOS performance rests 19% far from that of an "oracle" that, while unpractical in reality, knows exactly the last point in time when to checkpoint;

- compared to existing approaches, HARVOS reduces the amount of MCU processing whose results do not eventually become part of a checkpoint;

- compared to existing approaches, HARVOS allows transiently-powered devices to complete the same fixed workload using smaller energy buffers;

- HARVOS performance is robust against implementations of different complexity and structure, unlike existing approaches.

The following sections provide quantitative evidence of these findings.

6.5.1 Settings

Benchmarks and setup We consider publicly available C implementations of Kalman filter [9], finite impulse response (FIR) filter [8], and Advanced Encryption Standard (AES) [7] with key length of 256 bits. Kalman filters are often used in embedded sensing to process accelerometer values; for example, to predict future trends. FIR filters are equally used in embedded sensing to filter out noise, especially when the input signal includes multirate components. AES is a block-cipher algorithm widely employed in embedded systems.

The implementations we consider include a variety of branching statements, loops, and function calls; therefore, they exercise most of the trigger placement rules described in Section 6.4. These implementations also exhibit different degrees of complexity, with the Kalman filter code being the simplest, the FIR filter code being the longest in number of lines of C code, and the AES implementation being the one structurally most complex. Overall, the benchmarks are arguably on par, and sometimes more complex, than those considered in existing literature [19, 171]

The Kalman filter code executes a fixed workload of 1000 iterations using 48 bytes of dummy data as input, emulating the use of Kalman filter

to process consecutive acceleration readings. Instead, we consider a single iteration of both the FIR filter and AES implementations as they perform enough processing in a single iteration to raise the problem of resets with limited energy buffers. We feed the FIR filter code with 116 bytes of dummy data, whereas AES processes 100 bytes of dummy data for both encryption and decryption. We consider these sizes as they are comparable to the size of a radio packet.

These benchmark codes do not take decisions based on sensed values or perform actuation. To introduce some degree of unpredictability, we experiment with a further custom version of the Kalman filter code, where we artificially modify the executions so that there is a 75% probability that a dummy interrupt handler worth 10000 MCU cycles preempts the execution. The latter version serves to measure the performance of the rules in Section 6.4.3 in an extreme case.

We consider an ARM Cortex M3 MCU aboard an ST Nucleo L152RE board, equipped with a standard flash chip as stable storage. The board is not specifically designed for transiently-powered operation; however, here we are only interested in the MCU, which represents state-of-the-art technology and was recently employed in designs with power consumption comparable to earlier 16-bit platforms [16]. To this end, ST Nucleo boards crucially provide a range of hooks useful to monitor the MCU execution and isolate its power consumption. Using the ST-Link in-circuit debugger, Kiel uVision, and a Tektronix TBS 1072B oscilloscope, we can obtain very fine-grained information on the real hardware execution, including all the information required at compile-time as described in Section 6.3. This is key to the accuracy of the results.

Metrics Based on the information gathered with the setup above, we consider a variable size for a capacitor used as energy buffer that we assume to fully recharge every time is exhausted. Then, similar to existing works [19, 171], we synthetically emulate the execution of the code and compute three key metrics:

1. The *number of times the MCU resets* because the capacitor needs to recharge to complete the fixed workload. This figure is inversely proportional to the effectiveness of a given instrumentation strategy. Given a fixed workload, the more the MCU needs to reboot, the more the checkpointing operation is subtracting energy from useful computations. Lower values are thus better.

2. The *number of wasted cycles* because the MCU exhausts the energy before reaching the next trigger call. This figure is again inversely

proportional to the effectiveness of a certain solution. The higher is this figure, the more a given instrumentation strategy is failing in accurately identifying the last useful moment where to possibly checkpoint. Lower values are thus again better.

3. The *minimum size of the energy buffer* that allows the MCU to complete the workload under a given instrumentation strategy. For example, depending on how the trigger calls are placed, with small capacitor sizes, the execution may not reach even the first trigger call. This means checkpointing never happens and the system is stuck in a live-lock situation, rebooting every time from the initial state.

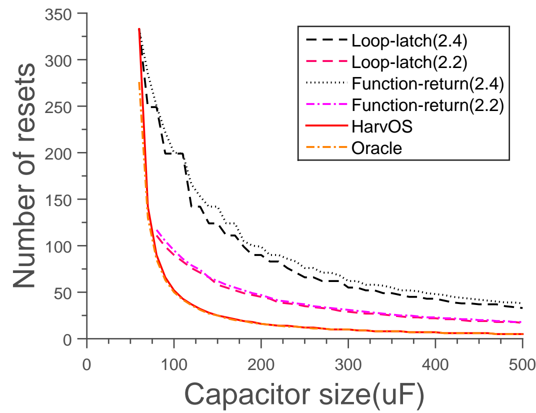
Baselines We consider the *loop-latch* and *function-return* strategies of MementOS, described in Section 6.2, as representative of current approaches.

Note that the voltage threshold MementOS considers to decide whether to checkpoint is the result of repeated emulation experiments ran with a specific application code and user-supplied energy traces [171]. HARVOS does not require users to supply similar traces, which may be hard to obtain in the first place. Our solution is rather based on a worst-case analysis and assumes that the environment provides no additional energy once the device can reboot.

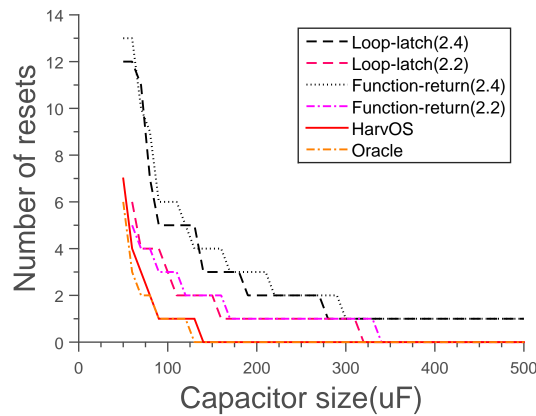
Considering this specific energy supply pattern also in the evaluation, as we do, does not impact the results. Should the environment provide new energy after the device reboots, equation (6.1) in Section 6.3 would capture the new supply of energy as part of $E_{remaining}$. Similarly, the new supply of energy would affect the execution of MementOS as soon as it is sufficient to move the operating voltage above the threshold.

To study the performance of MementOS in a manner orthogonal to the availability of energy traces, in our experiments we manually vary the value of MementOS voltage threshold. This way, we are likely to cover also the specific setting that MementOS would identify given a specific application code, and orthogonally to energy traces. We experiment with multiple such thresholds, always above the minimum voltage that still allows the system to write onto the flash chip. Settings of or below 2.2V, in particular, were never seen for MementOS in related literature [19, 171], and are likely to play favorably to it.

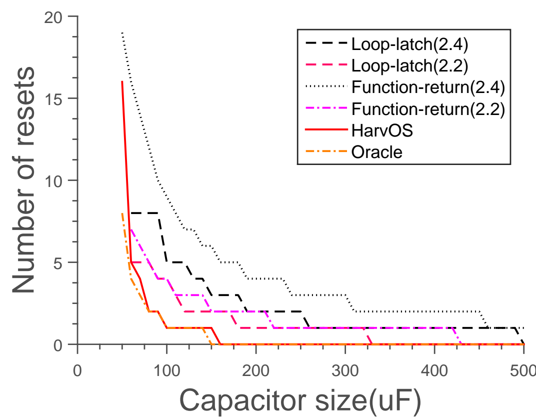
In addition, we apply a brute-force search on all possible executions of the code to identify an *oracle* that, by predicting how the execution is going to unfold in the future, knows the last practical point in time when to checkpoint. This is not feasible in reality; to make it work in a concrete execution, one would theoretically need to place a trigger call after every



(a) *Kalman filter.*



(b) *FIR filter.*



(c) *AES.*

Figure 6.8: *Number of resets necessary to complete a fixed workload.* HARVOS improves by a 69% factor on average, with a peak improvement of 80% compared to MementOS, while performing close to the oracle.

instruction in the code, yielding an unbearable overhead.

6.5.2 Results

Number of resets Figure 6.8 plots the results we obtain in the number of MCU resets for a fixed workload, with no preemption due to interrupt handlers. As expected, bigger capacitor sizes generally correspond to fewer resets, in that individual executions progress farther on a single charge.

Compared to either of the MementOS strategies, HARVOS completes the fixed workload with 69% fewer resets on average, with a peak improvement of 80% fewer resets. On the other hand, certain configurations exist, especially for the FIR filter code in Figure 6.8b and for the AES implementation in Figure 6.8c, where the performance is the same. Yet, HARVOS never performs worse than MementOS in our experiments.

Moreover, in most cases the performance of our solution rests very close to the *oracle*. Notably for the Kalman filter code, the performance is often almost the same. This demonstrates that the rationale explained in Section 6.3 strikes an effective trade-off between opposite needs, ultimately performing similarly to an optimal solution that is, however, unfeasible in practice.

Comparing Figure 6.8a, obtained using the Kalman filter code, with Figure 6.8b that shows the performance with the FIR filter code as well as Figure 6.8c that depicts the performance of the AES implementation, one may note that the performance of HARVOS is robust against diverse benchmark codes. The Kalman filter code is 1053 lines of C code and includes a few function calls at the outmost level of the code structure; the size of a checkpoint is 442 bytes. The AES implementation is 2848 lines of C code and it includes two loops and multiple function calls at the outmost level of the code structure; the size of a checkpoint is 624 bytes. The FIR code is 14986 lines of C code as it includes several digital signal processing functions part of a larger library, but includes two loops with fewer function calls than AES at the outmost level of code structure; the size of a checkpoint is 568 bytes.

MementOS, on the other hand, shows a different behavior in terms of how the program structure affects the performance. For a certain voltage threshold, the performance of both MementOS strategies is very similar for the Kalman filter code in Figure 6.8a and for the FIR filter code in Figure 6.8b. In contrast, *loop-latch* performs distinctively better than *function-return* for the AES implementation, as shown in Figure 6.8c. In fact, one should generally not only emulate different voltage thresholds for config-

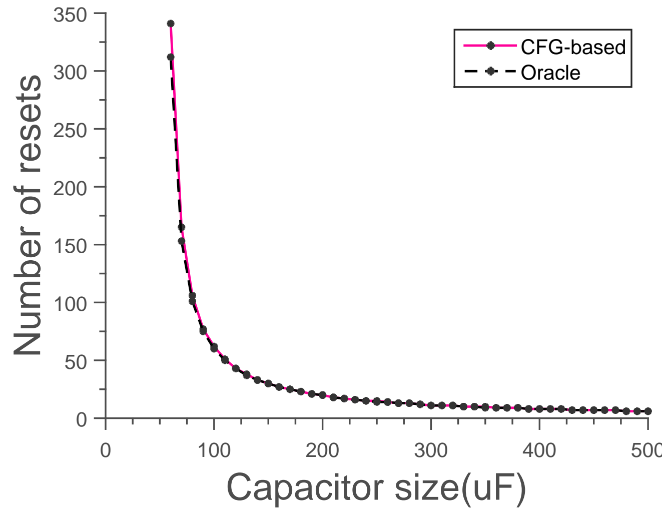


Figure 6.9: *Number of resets necessary to complete a fixed workload with the implementation of Kalman filter in preemptable executions.* We emulate a 75% probability of preemption by an interrupt handler. The performance is close to that of Figure 6.8a with no interrupts. The rules of Section 6.4.3 bear minimal additional impact.

uring MementOS, but also try with different instrumentation strategies to find the best performing configuration [171]. The process may then become laborious.

To complement these results, Figure 6.9 depicts the number of resets necessary to complete the Kalman filter workload with a 75% probability that the main code is preempted by a dummy interrupt handler. As MementOS does not explicitly account for interrupt handlers, we can only compare against the oracle here. Compared to Figure 6.8a, the performance is remarkably similar. The strategy described in Section 6.4.3, as a result, bears minimal additional overhead.

Explaining the improvements The gains over MementOS are intuitively explained in Figure 6.10. MementOS employs a *single* voltage threshold in all of its trigger calls. Such threshold defines a "grey region" of energy supplies that, during the off-line emulation experiments, were found *at least once* to *immediately* require a checkpoint, or the device would die before reaching the next trigger call with sufficient energy to checkpoint. As soon as MementOS enters the grey region and a trigger call is executed, a checkpoint takes place without checking whether it can reach the next trigger call given the available energy.

In contrast, our solution works in a localized fashion. Once we enter in what MementOS would consider the grey area, every trigger call results in

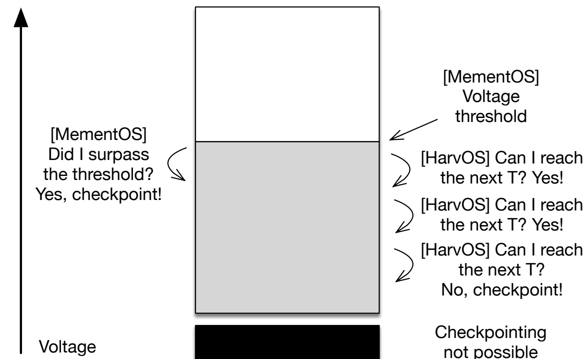


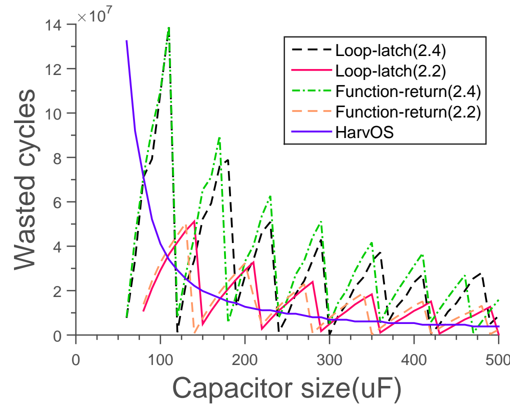
Figure 6.10: *Graphically comparing the behavior of MementOS against* HARVOS. The voltage threshold in MementOS defines a "grey area" that corresponds to a mandatory checkpoint as soon as it is entered. While this threshold applies globally to the whole program and may penalize specific executions by checkpointing too early, HARVOS can postpone the decision to checkpoint based on the specific situation.

a checkpoint *only if* the available energy is insufficient, in the worst-case, to reach the next trigger call and to checkpoint there. This is, in essence, what equation (6.1) in Section 6.3 stipulates. This reasoning allows us to postpone checkpointing in time and space, essentially "digging" down into the grey area as long as possible.

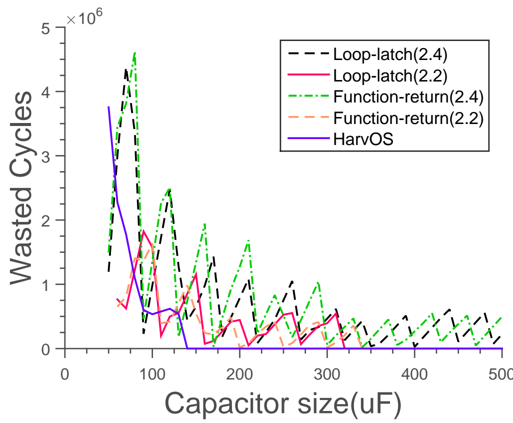
Wasted cycles Figure 6.11 shows the number of wasted cycles against the capacitor size. As expected in light of the discussion of Figure 6.10, at higher threshold voltages, MementOS wastes a higher number of cycles because checkpoints tend to happen too early, leaving unused energy in the buffer.

Moreover, the staircase pattern in MementOS, especially visible for the Kalman filter code in Figure 6.11a and the FIR filter code in Figure 6.11b, appears because both its strategies are too coarse-grained. Either the trigger call is located at the "right" place, and so there are only a few wasted cycles, or it is located at "wrong" place and so a lot of unused energy remains in the buffer. In fact, MementOS placement strategies only look at the structure of the code and not at its energy consumption patterns.

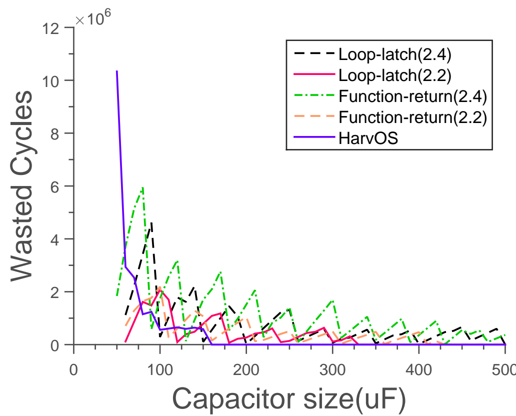
Differently, with smaller capacitors, HARVOS results in a higher number of wasted cycles than MementOS because equation (6.1) in Section 6.3 often finds $E_{remaining}$ insufficient to reach the next trigger call *in the worst case*. The value of $E_{remaining}$ is, in fact, upper-bound by the size of the capacitor. The worst-case analysis we apply is here counter-productive, in that it tends to be excessively pessimistic. The larger is the capacitor, however, the less this issue affects the operation of HARVOS, yielding in-



(a) *Kalman filter.*

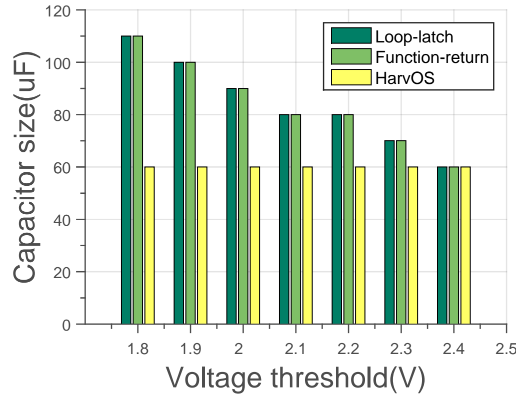


(b) *FIR filter.*

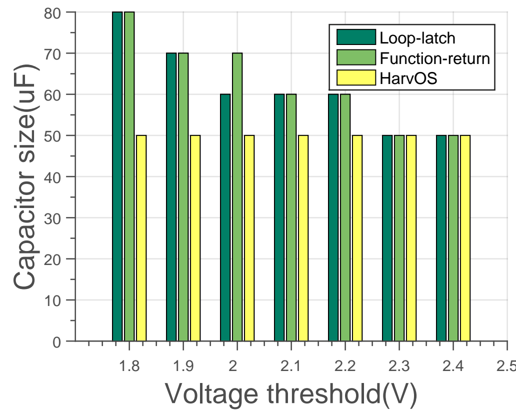


(c) *AES.*

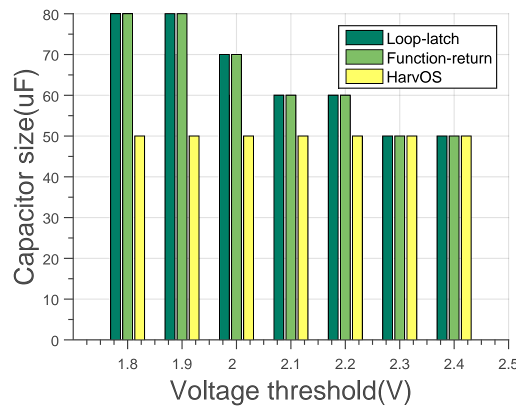
Figure 6.11: *Number of wasted cycles.* Both MementOS strategies are too coarse-grained and oblivious of energy consumption patterns. Differently, basing the checkpointing decisions on the ability to reach the next trigger call yields increasingly precise decisions with bigger capacitors.



(a) *Kalman filter.*



(b) *FIR filter.*



(c) *AES.*

Figure 6.12: *Minimum capacitor size required to complete the three benchmarks.* MementOS places trigger calls too far from each other, and thus requires bigger capacitors to complete the workload. Our placement rules allows the system to complete the workload with smaller capacitors.

creasingly precise checkpoint decisions that reduce the number of wasted cycles.

Minimum size of energy buffer Figure 6.12 reports the minimum capacitor size required to complete the three benchmarks, against different threshold voltages for MementOS.

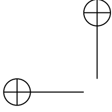
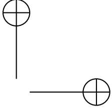
When operated under lower threshold voltages, MementOS finishes the execution only with bigger capacitors. This is essentially a result of its logic for placing triggers and of basing the decision to checkpoint on voltage levels. Without reasoning on the energy necessary to reach the next trigger call, MementOS may place trigger calls too far from each other. Under lower threshold voltages, the execution may then continue "blindly" up to a point when insufficient energy is left to complete the checkpoint, namely it is too late to checkpoint. To remedy this, a bigger capacitor is needed.

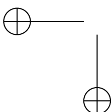
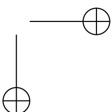
In contrast, our solution allows one to employ smaller capacitors. The splitting of the CFG in sub-graphs whose stretch is at most equal to the number of cycles the MCU can execute in a worst-case situation, as explained in Section 6.3, rules out the possibility of placing trigger calls too far apart. In fact, our performance in Figure 6.12 is independent of the voltage threshold used by MementOS. At each trigger call, we again decide whether to continue the execution or to checkpoint based on the ability to reach the next trigger call with sufficient energy to checkpoint there, as equation (6.1) indicates. The ability to complete a given workload with smaller capacitors may be particularly beneficial in applications requiring quick recharge times.

6.6 Summary

HARVOS operates at compile-time based on the control-flow graph (CFG) of the program. Trigger calls are placed by looking at the worst-case energy cost required to reach the next trigger call and depending on the program structure as represented in the CFG. The information collected at compile-time also enables to spare the energy-intensive probing of the energy buffers whenever possible. The combination of these techniques allows the system to take informed decisions at every trigger call on whether to continue with the normal execution or to rather checkpoint. Our evaluation of the approach, based on three diverse benchmarks, indicates that our techniques allow transiently-powered devices to complete a given workload with 68% fewer checkpoints, compared to existing literature. Our performance also rests only 19% far from that of an "oracle" that would know exactly the last point in time when a checkpoint is required. The performance of HAR-

VOS may further improve by removing, whenever possible, the worst-case assumption at the cost of additional code instrumentation.

CHAPTER 7

Conclusion and Future Directions

This thesis has presented a set of tools that enable checkpointing of the program state on stable storage, along with its later recovery, with minimal latency and energy consumption [28,29] for transiently-powered embedded system. This pushes the boundaries of the embedded sensing and enables richer data collection.

In the first part of the thesis, we presented the detailed analysis of the existing energy harvesting and wireless energy transfer solutions for wireless sensor networks (WSNs) [27]. We defined desirable properties that energy harvesting and wireless transfer techniques must present to enable their use in WSN applications, we surveyed and classified existing solutions, and argued about their applicability in different deployment environments.

Later, we conducted a comprehensive survey of the state of the art for transiently-powered embedded systems and classified them into three categories: "*out-of-place*", "*in-place*" and "*non-volatile processor*" solutions. Overall, "*out-of-place*" checkpointing solutions target mainstream embedded sensing architectures, employing a volatile main memory for efficient processing, and NVM as external storage.

"*In-place*" checkpointing solutions relieve the system from checkpointing main memory but it increases the energy consumption during normal

computations, due to the use of FRAM as main memory. Although they are commercially available, they are not widespread and still far from massive production. On the other hand, "*non-volatile processors*" solutions relieve the system from checkpoints altogether, yet require dedicated processor designs and do not exist commercially. Both these solutions, *"in-place"* and *"non-volatile processors"*, are beneficial only under those scenarios where power interruptions happen very frequently.

In the second part of the thesis, we formalized the challenge of transiently powered embedded systems, i.e., enabling applications to make progress across periods of energy unavailability, into three high-level research questions:

- **How to enable efficient checkpointing mechanism?**
  We presented techniques to checkpoint and restore a device's state on stable storage, catering for scenarios where devices opportunistically harvest energy from the ambiance or are provided with wireless energy transfer mechanisms. Our work aims at reducing the time for these operations and at minimizing their energy cost. We target modern 32-bit MCUs and currently available flash chips (but can extend to any type of NVM), making the checkpoint and restore routines available to programmers through a pair of simple C functions. The three storage modes we designed in support expose different trade-offs that depend on the memory *span*, its *occupation*, the possible *fragmentation*, and the read/write patterns in memory. The experimental results we gathered allowed us to quantify these trade-offs and discern the application's characteristics that would make one storage mode preferable over another.

- **What to checkpoint?**
  Our efforts to design efficient checkpointing mechanism revealed that the key aspect of any system should be its ability to automatically and efficiently estimate the minimal application checkpoint state, which is required to ensure forward progress. We designed DICE, which is a set of compile-time techniques for transiently-powered systems to reduce the amount of data needed to write on NVM during checkpoints. We conceived different ways to track changes in main memory, depending on what memory segment these reside on, as well as lightweight code instrumentation strategies with dedicated customization depending upon how checkpoints are possibly triggered. DICE helps existing system support to complete a given workload with *i)* smaller energy-buffers, *ii)* fewer checkpoints, and thus better energy

efficiency, and *iii)* reduced execution latency. Our experimental evaluation, based on a combination of three benchmarks across three different existing system support and two different hardware platforms, provides quantitative evidence. For example, using DICE, HARVOS can complete the execution of the RSA algorithm with 86% fewer checkpoints, resulting in better overall energy utilization and a 34% reduction in execution latency.

- **Where and when to trigger checkpointing mechanism?**
  We proposed HARVOS that operates at compile-time. It inserts trigger calls by looking at the worst-case energy cost required to reach the next trigger call, depending on the program structure as represented in the CFG. The information collected at compile-time also enables to spare the energy-intensive probing of the energy buffers whenever possible. The combination of these techniques allow the system to take informed decisions at every trigger call on whether to continue with the normal execution or to rather checkpoint. Our evaluation of the approach, based on three diverse benchmarks, indicates that our techniques allow transiently-powered embedded devices to complete a given workload with 68% fewer checkpoints, compared to existing literature. Our performance also rests only 19% far from that of an "oracle" that would know exactly the last point in time when a checkpoint is required. The performance of HARVOS may further improve by removing, whenever possible, the worst-case assumption at the cost of additional code instrumentation.

7.1 Future Directions

As we explained earlier in chapter 6, HARVOS split the CFG of the program into sub-graphs. One logical continuation of our work is to exploit these sub-graphs to further reduce the size of system state that we need to checkpoint to ensure forward progress of application. Within each sub-graph, we can find two types of data structures: those that are local and do not affect the outcome of other sub-graphs, and the others that are shared among different sub-graphs and have a direct effect on the outcome of other sub-graphs. In future, we will further optimize the application checkpoint state size by saving only those data structures that are being shared among different sub-graphs of the program. Saving only such data structures will greatly reduce the checkpoint state size while preserving forward progress. We will automatically identify which data structures are overlapping in different subgraphs and then only save those data structures.

Also, we will provide application-specific customizations by providing simple API, as only developers know what are the conditions that render restored state effectively usable when a device re-starts. The API will allow the programmer to set time-related validity constraints on application state. As the time a device remains without energy is unpredictable, programmers will need to check the state validity before using it. This may be necessary because part of the state may bear time-related validity constraints, as in the case of sensor data.

Bibliography

[1] Honeywell Inc: Honeywell Process Solutions—White Paper. tinyurl.com/honeywell-whitepaper.

[2] mbed. tinyurl.com/pkgoy6d, 2015.

[3] STM32 32-bit ARM Cortex MCUs. tinyurl.com/STM32bitMCU, 2015.

[4] STM32 Cube. tinyurl.com/STM32CubeMX, 2015.

[5] TinyOS CTP tree routing . tinyurl.com/TreeRouting, 2015.

[6] TinyOS Link Estimator. tinyurl.com/LinkEstimator, 2015.

[7] AES C implementation for Mbed platform. goo.gl/PBjhoF, 2016.

[8] FIR Filter C implementation for Mbed platform. goo.gl/yFyUyX, 2016. Accessed: 10-13-2016.

[9] Kalman Filter C implementation for Mbed platform. goo.gl/ikzYFt, 2016. Accessed: 30-9-2016.

[10] mBed OS energy profiler. goo.gl/dghhd4, 2016. Accessed: 11-10-2016.

[11] Yuvraj Agarwal et al. Duty-cycling Buildings Aggressively: The Next Frontier in HVAC Control. In *IPSN*, 2011.

[12] Fayaz Akhtar and Mubashir Husain Rehmani. Energy replenishment using renewable and traditional energy resources for sustainable wireless sensor networks: A review. *Renewable and Sustainable Energy Reviews*, 45:769–784, 2015.

[13] L. Almeida, P. Pedreiras, and J. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Trans. on Industrial Electronics*, 49(6), 2002.

[14] Shaban Almouahed, Manuel Gouriou, Chafiaa Hamitouche, Eric Stindel, and Christian Roux. The use of piezoceramics as electrical energy harvesters within instrumented knee implant during walking. *IEEE/ASME Transactions on Mechatronics*, 16(5):799–807, 2011.

[15] P Anacleto, PM Mendes, E Gultepe, and DH Gracias. 3d small antenna for energy harvesting applications on implantable micro-devices. In *Antennas and Propagation Conference (LAPC), 2012 Loughborough*, pages 1–4. IEEE, 2012.

[16] Michael Andersen et al. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*, 2016.

[17] Robert E Armstrong. *Bio-inspired innovation and national security*. Smashbooks, 2010.

[18] S. Arra et al. Ultrasonic power and data link for wireless implantable applications. In *2nd International Symposium on Wireless Pervasive Computing, 2007. ISWPC '07.*, pages –, Feb 2007.

[19] Alberto Arreola et al. Approaches to transient computing for energy harvesting systems: A quantitative evaluation. In *ENSSYS*, 2015.

[20] Domenico Balsamo, Ali Elboreini, Bashir Al-Hashimi, and Geoffrey Merrett. Exploring arm mbed support for transient computing in energy harvesting iot systems. 2017.

[21] Domenico Balsamo et al. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1), 2015.

[22] 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*, 35(12):1968–1980, 2016.

[23] Xiaoqi Bao, Will Biederman, Stewart Sherrit, Mircea Badescu, Yoseph Bar-Cohen, Christopher Jones, Jack Aldrich, and Zensheu Chang. High-power piezoelectric acoustic-electric power feedthru for metal walls. In *The 15th International Symposium on: Smart Structures and Materials & Nondestructive Evaluation and Health Monitoring*, pages 69300Z–69300Z. International Society for Optics and Photonics, 2008.

[24] Michael Barr and Anthony Massa. *Programming Embedded Systems*. O'Relly Media, 2006.

[25] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[26] Naveed Anwar Bhatti et al. Sensors with Lasers: Building a WSN Power Grid. In *IPSN*, 2014.

[27] Naveed Anwar Bhatti et al. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM TOSN*, 12, 2016.

[28] Naveed Anwar Bhatti and Luca Mottola. Efficient state retention for transiently-powered embedded sensing. In *EWSN*, 2016.

[29] 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 '17, pages 209–219, New York, NY, USA, 2017. ACM.

[30] Suzhi Bi, Chin Keong Ho, and Rui Zhang. Wireless powered communication: opportunities and challenges. *IEEE Communications Magazine*, 53(4):117–125, 2015.

[31] William C. Brown. Experiments involving a microwave beam to power and position a helicopter. *IEEE Transactions on Aerospace and Electronic Systems*, AES-5(5):692–702, Sept 1969.

[32] Davide Brunelli, Luca Benini, Clemens Moser, and Lothar Thiele. An efficient solar energy harvester for wireless sensor nodes. In *Proceedings of the conference on Design, automation and test in Europe*, pages 104–109. ACM, 2008.

[33] Bernhard Buchli et al. Battery state-of-charge approximation for energy harvesting embedded systems. In *EWSN*, 2013.

[34] Bernhard Buchli et al. Dynamic power management for long-term energy neutral operation of solar energy harvesting systems. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 31–45, 2014.

[35] Michael Buettner et al. RFID Sensor Networks with the Intel WISP. In *SENSYS*, 2008.

[36] Scott Calabrese Barton, Josh Gallaway, and Plamen Atanassov. Enzymatic biofuel cells for implantable and microscale devices. *Chemical reviews*, 104(10):4867–4886, 2004.

[37] Bradford Campbell et al. Energy-harvesting thermoelectric sensing for unobtrusive water and appliance metering. In *Proceedings of the 2nd International Workshop on Energy Neutral Sensing Systems*, pages 7–12. ACM, 2014.

[38] Michele Ceriotti et al. Is there light at the ends of the tunnel? wireless sensor networks for adaptive lighting in road tunnels. In *10th International Conference on Information Processing in Sensor Networks (IPSN), 2011*, pages 187–198. IEEE, 2011.

[39] Jayant Charthad et al. A mm-sized implantable device with ultrasonic energy transfer and rf data uplink for high-power applications. In *IEEE Proceedings of the Custom Integrated Circuits Conference (CICC), 2014*, pages 1–4. IEEE, 2014.

[40] James L Chen et al. A novel vertical axis water turbine for power generation from water pipelines. *Energy*, pages 184–193, 2013.

[41] Y. Chen et al. Surviving sensor network software faults. In *SOSP*, 2009.

[42] Eileen Y. Chou et al. Mixed-signal integrated circuits for self-contained sub-cubic millimeter biomedical implants. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010.

[43] W.C. Chye et al. Electromagnetic micro power generator: A comprehensive survey. In *IEEE Symposium on Industrial Electronics Applications (ISIEA), 2010*, pages 376–382. IEEE, 2010.

[44] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. *SIGPLAN Not.*, 51(10):514–530, October 2016.

[45] Peng Cong et al. A wireless and batteryless 10-bit implantable blood pressure sensing microsystem with adaptive rf powering for real-time laboratory mice monitoring. *IEEE Journal of Solid-State Circuits*, pages 3631–3644, 2009.

[46] Haipeng Dai, Lintong Jiang, Xiaobing Wu, D.K.Y. Yau, Guihai Chen, and Shaojie Tang. Near optimal charging and scheduling scheme for stochastic event capture with rechargeable sensors. In *IEEE 10th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS), 2013*, pages 10–18. IEEE, 2013.

[47] Haipeng Dai, Yunhuai Liu, Guihai Chen, Xiaobing Wu, and Tian He. Safe charging for wireless power transfer. In *INFOCOM, 2014 Proceedings IEEE*, pages 1105–1113. IEEE, 2014.

[48] Haipeng Dai, Yunhuai Liu, Guihai Chen, Xiaobing Wu, and Tian He. Scape: Safe charging with adjustable power. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS), 2014*, pages 439–448. IEEE, 2014.

[49] Jianing Dai, Jun-Jian Wang, Alex T Chow, and William H Conner. Electrical energy production from forest detritus in a forested wetland using microbial fuel cells. *GCB Bioenergy*, 7(2):244–252, 2015.

[50] Jack Davidson and Sanjay Jinturkar. Improving instruction level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO*, 1995.

[51] Carlisto de Alvarenga et al. A notification architecture for smart cities based on push technologies. In *Computing Conference (CLEI), 2014 XL Latin American*, pages 1–8. IEEE, 2014.

[52] Marc De Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE Computer Society, 2013.

[53] Antonio Carlos M. de Queiroz. Electrostatic generators for vibrational energy harvesting. In *IEEE Fourth Latin American Symposium on Circuits and Systems (LASCAS), 2013*, pages 1–4, 2013.

[54] Samuel DeBruin, Bradford Campbell, and Prabal Dutta. Monjolo: An energy-harvesting energy meter architecture. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 18. ACM, 2013.

[55] Martin Deterre, Elie Lefeuvre, Yanan Zhu, Marion Woytasik, Bertrand Boutaud, and Renzo Dal Molin. Micro blood pressure energy harvester for intracardiac pacemaker. *Journal of Microelectromechanical Systems*, 23(3):651–660, 2014.

[56] Daniele Dondi et al. A WSN System Powered by Vibrations to Improve Safety of Machinery with Trailer. In *IEEE Sensors*, 2012.

[57] Conrad Donovan, Alim Dewan, Deukhyoun Heo, and Haluk Beyenal. Batteryless, wireless sensor powered by a sediment microbial fuel cell. *Environmental science & technology*, 42(22):8591–8596, 2008.

[58] Nelson I. Dopico et al. Performance assessment of a kinetically-powered network for herd localization. *Computers and Electronics in Agriculture*, 87:74–84, 2012.

[59] Prabal Dutta et al. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN*, 2008.

[60] Energous. WattUp. `http://energous.com/technology/`, 2015.

[61] Deborah Estrin et al. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, 1999.

[62] Deborah Estrin et al. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1, 2002.

[63] H. A. Nguyen et al. Sensor node lifetime: An experimental study. In *IEEE PerCom 2011, 21-25 March 2011*, 2011.

[64] R. Smith et al. Surviving peripheral failures in embedded systems. In *USENIX ATC*, 2015.

[65] Timothy Ewing et al. Self-powered wastewater treatment for the enhanced operation of a facultative lagoon. *Journal of Power Sources*, 269:284–292, 2014.

[66] Luca Filipponi et al. Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors. In *Fourth International Conference on Sensor Technologies and Applications (SENSORCOMM), 2010*, pages 281–286. IEEE, 2010.

[67] Emanuele Frontoni et al. Energy harvesting for smart shoes: A real life application. *2013 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications*, pages V004T08A034–V004T08A034, August 2013.

[68] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *SENSYS*, 2009.

[69] Yanming Gong et al. Benthic microbial fuel cell as direct power source for an acoustic modem and seawater oxygen/temperature sensor system. *Environmental Science & Technology*, 45(11):5047–5053, 2011.

[70] Maria Gorlatova et al. Challenge: ultra-low-power energy-harvesting active networked tags (EnHANTs). In *In Proceedings of the 15th annual international conference on Mobile computing and networking Mobicom '09*, pages 253–260. ACM, Sept. 2009.

[71] Daniel J. Graham et al. Investigation of methods for data communication and power delivery through metals. *IEEE Transactions on Industrial Electronics*, 58(10):4972–4980, 2011.

[72] Jeremy Gummeson et al. On the limits of effective hybrid micro-energy harvesting on mobile crfid sensors. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 195–208. ACM, 2010.

[73] Julian Gutierrez et al. Automated irrigation system using a wireless sensor network and gprs module. *IEEE Transactions on Instrumentation and Measurement*, 63(1):166–176, 2014.

[74] Gerhard Hancke and N.A. Vorster. The feasibility of using resonant inductive power transfer to recharge wireless sensor network nodes. In *Wireless Power Transfer Conference (WPTC), 2014 IEEE*, pages 100–105. IEEE, May 2014.

[75] Abhiman Hande, Todd Polk, William Walker, and Dinesh Bhatia. Indoor Solar Energy Harvesting for Sensor Network Router Nodes . *Microprocessors and Microsystems*, 31(6), 2007.

[76] HART Communication Foundation. WirelessHART. `http://en.hartcomm.org/main_article/wirelesshart.html`, 2015.

[77] Andreas Hasler et al. Wireless sensor networks in permafrost research concept, requirements, implementation and challenges. In *Proc. 9th International Conference on Permafrost (NICOP)*, volume 1, pages 669–674, Jun 2008.

[78] John Heidemann et al. Research challenges and applications for underwater sensor networking. volume 1, pages 228–235. IEEE, 2006.

[79] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *SENSYS*, 2015.

[80] J. Hester et al. Flicker: Rapid prototyping for the batteryless internet-of-things. In *SenSys*, 2017.

[81] J. Hester et al. New directions: The future of sensing is batteryless, intermittent, and awesome. In *SenSys*, 2017.

[82] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 228–240. ACM, 2017.

[83] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[84] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *ESOP*, 2006.

[85] Te-Chien Hou et al. Triboelectric nanogenerator built inside shoe insole for harvesting walking energy. *Nano Energy*, 2(5):856–862, 2013.

[86] Hongping Hu et al. Wireless energy transmission through a thin metal wall by shear wave using two piezoelectric transducers. In *Ultrasonics Symposium, 2008. IUS 2008. IEEE*, pages 2165–2168. IEEE, 2008.

[87] Yuantai Hu et al. Transmitting electric energy through a metal wall by acoustic waves using piezoelectric transducers. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 50:2165–2168, 2008.

[88] IAR Systems. IAR Embedded Workbench Cortex M Edition. tinyurl.com/arm-m-workbench, 2015.

[89] IAR Systems. Mastering Stack and Heap for System Reliability. tinyurl.com/iar-stack-heap, 2015.

[90] Texas Instrument Inc. Intelligent system state restoration after power failures with compute through power loss utility, jun 2017.

[91] International Society of Automation. ISA100.11a. https://www.isa.org/isa100/, 2015.

[92] O. Iova et al. Rpl: The routing standard for the internet of things... or is it? *IEEE Communications Magazine*, 54(12), 2016.

[93] Hrishikesh Jayakumar et al. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *VLSI Design*, 2014.

[94] 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 Trans. Embed. Comput. Syst.*, 16(3):65:1–65:23, April 2017.

[95] Xiaofan Jiang et al. Perpetual environmentally powered sensor networks. In *4rth International Symposium on Information Processing in Sensor Networks, 2005. IPSN 2005.*, pages 463–468. IEEE, 2005.

[96] O. Jonah and S.V. Georgakopoulos. Wireless power transmission to sensors embedded in concrete via magnetic resonance. In *IEEE 12th Annual Wireless and Microwave Technology Conference (WAMICON), 2011*, pages 1–6. IEEE, 2011.

[97] Salman Kahrobaee and Mehmet C. Vuran. Vibration energy harvesting for wireless underground sensor networks. In *ICC*, 2013.

[98] Nobuo Kaku et al. Plant/microbe cooperation for electricity generation in a rice paddy field. *Applied microbiology and biotechnology*, 79(1):43–49, 2008.

[99] Kai Kang. Multi-source energy harvesting for wireless sensor nodes. 2012.

[100] Stella J. Karavas et al. VoltreePower. http://www.voltreepower.com, 2007.

[101] Zahid Kausar et al. Energizing wireless sensor networks by energy harvesting systems: Scopes, challenges and approaches. *Renewable and Sustainable Energy Reviews*, 38:973–989, 2014.

[102] S. Khanna, S. C. Bartling, M. Clinton, S. Summerfelt, J. A. Rodriguez, and H. P. McAdams. An fram-based nonvolatile logic mcu soc exhibiting 100retention at $rmVDD = 0$ v achieving zero leakage with lt; 400-ns wakeup time for ulp applications. *IEEE Journal of Solid-State Circuits*, 49(1):95–106, 2014.

[103] M. Kluge et al. Remote acoustic powering and data transmission for sensors inside of conductive envelopes. In *Sensors, 2008 IEEE*, pages 41–44. IEEE, 2008.

[104] JeongGil Ko et al. Low power or high performance? a tradeoff whose time has come (and nearly gone). In *Proceedings of the 9th European Conference on Wireless Sensor Networks (EWSN)*, pages 98–114. Springer, 2012.

[105] Richard Koo and Sam Toueg. Checkpointing and Rollback-recovery for Distributed Systems. In *Proceedings of ACM Fall Joint Computer Conference*, 1986.

[106] P. Koopman. *Better Embedded System Software*. CMU Press, 2010.

[107] Haluk Kulah and K. Najafi. An electromagnetic micro power generator for low-frequency environmental vibrations. In *17th IEEE International Conference on Micro Electro Mechanical Systems, 2004*, pages 237–240. IEEE, 2004.

[108] Ye Kuo et al. Mbus: A 17.5 pj/bit/chip portable interconnect bus for millimeter-scale sensor systems with 8 nw standby power. In *IEEE Proceedings of the Custom Integrated Circuits Conference (CICC), 2014*, pages 1–4. IEEE, 2014.

[109] Gudan Kurilj et al. A 2.4ghz ambient rf energy harvesting system with -20dbm minimum input power and nimh battery storage. In *RFID Technology and Applications Conference (RFID-TA), 2014 IEEE*, pages 7–12. IEEE, 2014.

[110] Andre Kurs et al. Wireless power transfer via strongly coupled magnetic resonances. *science*, 317(5834):83–86, 2007.

[111] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[112] Andrew Leakey et al. Physiological and ecological significance of sunflecks for dipterocarp seedlings. *Journal of Experimental Botany*, 56(411):469–482, 2005.

[113] E.A. Lee. Cyber Physical Systems: Design Challenges. In *IEEE ISORC*, 2008.

[114] Charles Leech, Yordan P Raykov, Emre Ozer, and Geoff V Merrett. Real-time room occupancy estimation with bayesian machine learning using a single pir sensor and microcontroller. In *Sensors Applications Symposium (SAS), 2017 IEEE*, pages 1–6. IEEE, 2017.

[115] Marc Leeman, Geert Deconinck, Vincenzo De Florio, David Atienzà, Jose M Mendias, Chantal Ykman, Francky Catthoor, and Rudy Lauwereins. Methodology for refinement and optimization of dynamic memory management for embedded systems in multimedia applications. In *Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on*, pages 369–374. IEEE, 2003.

[116] Ke Li, Hao Luan, and Chien-Chung Shen. Qi-ferry: Energy-constrained Wireless Charging in Wireless Sensor Networks. In *WCNC*, 2012.

[117] Mo Li and Yunhao Liu. Underground coal mine monitoring with wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 5(2):10, 2009.

[118] Zi Li, Yang Peng, Wensheng Zhang, and Daji Qiao. J-roc: A joint routing and charging scheme to prolong sensor network lifetime. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 373–382. IEEE, 2011.

[119] Dawei Liang and Joana Almeida. Highly efficient solar-pumped nd: Yag laser. *Optics express*, 19(27):26399–26405, 2011.

[120] Kris Lin, Jennifer Yu, Jason Hsu, Sadaf Zahedi, David Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani Srivastava. Heliomote: enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 309–309. ACM, 2005.

[121] Bitsch Link et al. Burrowview - seeing the world through the eyes of rats. In *Proceedings of the Second IEEE International Workshop on Information Quality and Quality of Service for Pervasive Computing (IQ2S 2010), Mannheim, Germany*, pages 56–61. IEEE, 2010.

[122] Peng Liu et al. eLighthouse: Enhance Solar Power Coverage in Renewable Sensor Networks. *IJDSN*, 2013.

[123] Vincent Liu et al. Ambient backscatter: Wireless communication out of thin air. *SIGCOMM Comput. Commun. Rev.*, 43(4):39–50, 2013.

[124] Vincent Liu et al. Ambient backscatter: Wireless communication out of thin air. In *SIGCOMM*, 2013.

Bibliography

[125] Markus Lohndorf et al. Evaluation of energy harvesting concepts for tire pressure monitoring systems. *Proceedings of Power MEMS*, pages 331–334, 2007.

[126] Konrad Lorincz et al. Deploying a wireless sensor network on an active volcano. In *IEEE Internet Computing*, volume 10, pages 18–25. IEEE, 2006.

[127] Marianne Lossec et al. Sizing optimization of a thermoelectric generator set with heatsink for harvesting human body heat. *Energy Conversion and Management*, 68:260–265, 2013.

[128] Christopher Love et al. Source of sustained voltage difference between the xylem of a potted ficus benjamina tree and its soil. *PloS one*, 3, 2008.

[129] Xiao Lu et al. Wireless networks with rf energy harvesting: A contemporary survey. *IEEE Communications Surveys & Tutorials*, 17(2):757–789, 2015.

[130] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *PLDI*, 2005.

[131] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *PLDI*, 2015.

[132] G. Lukosevicius et al. Using sleep states to maximize the active time of transient computing systems. In *ENSsys (with SenSys)*, 2017.

[133] Kevin MacVittie, Jan Halamek, Lenka Halamkova, Mark Southcott, William D. Jemison, Robert Lobel, and Evgeny Katz. From "cyborg" lobsters to a pacemaker powered by implantable biofuel cells. *Energy and Environmental Science*, 6(1):81–86, 2013.

[134] Alan Mainwaring et al. Wireless sensor networks for habitat monitoring. 2002.

[135] Andrew Markham et al. Revealing the hidden lives of underground animals using magneto-inductive tracking. In *8th ACM Conference on Embedded Networked Sensor Systems (Sensys 2010). Zurich, Switzerland*, pages 281–294. ACM, November 2010.

[136] Paul Martin et al. Doubledip: Leveraging thermoelectric harvesting for low power monitoring of sporadic water use. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 225–238. ACM, 2012.

[137] Cian O MathÃºna et al. Energy scavenging for long-term deployable wireless sensor networks. *Talanta*, 75(3):613–623, 2008.

[138] Hiroshi Matsumoto. Research on solar power satellites and microwave power transmission in japan. *IEEE microwave magazine*, 3(4):36–45, 2002.

[139] Scott Meninger, Jose Oscar Mur-Miranda, Rajeevan Amirtharajah, Anantha Chandrakasan, and Jeffrey H Lang. Vibration-to-electric energy conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):64–76, 2001.

[140] Unnikrishna Menon et al. Wireless power transfer to underground sensors using resonant magnetic induction. In *Tenth International Conference on Wireless and Optical Communications Networks (WOCN), 2013*, pages 1–5. IEEE, 2013.

[141] Nicole Metje et al. Real time condition monitoring of buried water pipes. *Tunnelling and Underground Space Technology*, 28:315–320, 2012.

[142] K. Mihic et al. Mstore: Enabling storage-centric sensornet research. In *IPSN*, 2007.

[143] Debabrata Mishra et al. Smart rf energy harvesting communications: challenges and opportunities. *Communications Magazine, IEEE*, 53(4):70–78, 2015.

[144] Masaki Mizunoand and D. G. Chetwynd. Investigation of a resonance microgenerator. *Journal of Micromechanics and Microengineering*, 13, 2003.

[145] S. Venkata Mohan et al. Harnessing of bioelectricity in microbial fuel cell (mfc) employing aerated cathode through anaerobic treatment of chemical wastewater using selectively enriched hydrogen producing mixed consortia. *Fuel*, 87(12):2667–2676, 2008.

[146] J.I. Moon et al. Design of efficient rectenna with vertical ground-walls for rf energy harvesting. *Electronics Letters*, 49(17):1050–1052, 2013.

[147] Raul Morais et al. Sun, wind and water flow as energy supply for small stationary data acquisition platforms. *Computers and electronics in agriculture*, 64(2):120–132, 2008.

[148] Luca Mottola. Programming Storage-centric Sensor Networks with Squirrel. In *IPSN*, 2010.

[149] Saman Naderiparizi et al. μMonitor: In-situ energy monitoring with microwatt power consumption. In *RFID*, 2016.

[150] Antwi Nimo, Tobias Beckedahl, Thomas Ostertag, and Leonhard Reindl. Analysis of passive rf-dc power rectification and harvesting wireless rf energy for micro-watt sensors. *AIMS Energy*, 3(2):184–200, 2015.

[151] H. Nishimoto et al. Prototype implementation of ambient rf energy harvesting wireless sensor networks. In *Sensors, 2010 IEEE*, pages 1282–1287. IEEE, 2010.

[152] U. Olgun et al. Design of an efficient ambient wifi energy harvesting system. *IET Microwaves, Antennas and Propagation*, 6(11):1200–1206, 2012.

[153] Fredrik Österlind et al. Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations. In *EWSN*, 2009.

[154] S.A. Ouellette and M.D. Todd. Cement seawater battery energy harvester for marine infrastructure monitoring. *Sensors Journal, IEEE*, 14:865–872, 2014.

[155] Shaul Ozeri et al. Ultrasonic transcutaneous energy transfer using a continuous wave 650khz gaussian shaded transmitter. *Ultrasonics*, 50(7):666–674, 2010.

[156] Joseph A. Paradiso and Mark Feldmeier. A compact, wireless, self-powered pushbutton controller. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, pages 299–304. Springer, 2001.

[157] Chulsung Park and P.H. Chou. AmbiMax: Autonomous energy harvesting platform for multi-supply wireless sensor nodes. volume 1, pages 168–177. IEEE, 2006.

[158] Aaron Parks et al. A wireless sensing platform utilizing ambient rf energy. In *IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet), 2013*, pages 154–156. IEEE, 2013.

[159] Terence Parr. *The Definitive ANTLR 4 Reference*. `goo.gl/RR1s`, 2013.

[160] Yang Peng et al. Prolonging sensor network lifetime through wireless charging. In *IEEE 31st Real-Time Systems Symposium (RTSS), 2010*, pages 129–139. IEEE, 2010.

[161] Andrea Pietrelli et al. Wireless sensor network powered by a terrestrial microbial fuel cell as a sustainable land monitoring energy system. *Sustainability*, 6(10):7263–7275, 2014.

[162] M. Pinuela et al. Ambient rf energy harvesting in urban and semi-urban environments. *IEEE Transactions on Microwave Theory and Techniques*, 61(7):2715–2726, 2013.

[163] G. Poulin et al. Generation of electrical energy for portable devices: Comparative study of an electromagnetic and a piezoelectric system. *Sensors and Actuators A: physical*, 116(3):461–471, 2004.

[164] PowerByProxi. PowerByProxi. `http://powerbyproxi.com/`, 2015.

[165] PowerCast. P2110 receiver and TX91501-3W-ID transmitter. `http://tinyurl.com/powercaster`, 2009.

[166] Shashank Priya, Chih-Ta Chen, Darren Fye, and Jeff Zahnd. Piezoelectric windmill: a novel solution to remote sensing. *Japanese journal of applied physics*, 44(1L):L104, 2004.

[167] Guofu Qiao et al. Remote corrosion monitoring of the rc structures using the electrochemical wireless energy-harvesting sensors and networks. *NDT and E International*, 44(7):583–588, 2011.

[168] Guofu Qiao et al. Heterogeneous tiny energy: An appealing opportunity to power wireless sensor motes in a corrosive environment. *Applied Energy*, 131:87–96, 2014.

[169] Qualcomm. WiPower. `https://www.qualcomm.com/products/wipower`, 2015.

[170] B. Ransford. *Transiently Powered Computers*. PhD thesis, School of Computer Science, UMass Amherst, 2013.

[171] Benjamin Ransford et al. MementOS: System support for long-running computation on RFID-scale devices. In *ASPLOS*, 2011.

[172] Benjamin Ransford et al. Mementos: System support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170, 2012.

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

[174] Clare E Reimers et al. Harvesting energy from the marine sediment-water interface. *Environmental science and technology*, 35(1):192–195, 2001.

[175] Luca Rizzon et al. Wireless sensor networks for environmental monitoring powered by microprocessors heat dissipation. In *Proceedings of the 1st International Workshop on Energy Neutral Sensing Systems*, ENSSys '13, page 8. ACM, 2013.

[176] Maurice Roes et al. Contactless energy transfer through air by means of ultrasound. In *IEEE Conference on Industrial Electronics (IECON)*, pages 1238–1243. IEEE, 2011.

[177] Maurice Roes et al. Acoustic energy transfer: a review. *IEEE Transactions on Industrial Electronics*, 60(1):242–248, 2013.

[178] Shad Roundy et al. A 1.9ghz rf transmit beacon using environmentally scavenged energy. In *Dig. IEEE Int. Symposium on Low Power Elec. and Devices*, 2003.

[179] Shad Roundy et al. A study of low level vibrations as a power source for wireless sensor nodes. *Computer communications*, 26(11):1131–1144, 2003.

[180] Noboru Sakimura, Yukihide Tsuji, Ryusuke Nebashi, Hiroaki Honjo, Ayuka Morioka, Kunihiko Ishihara, Keizo Kinoshita, Shunsuke Fukami, Sadahiko Miura, Naoki Kasai, et al. 10.5 a 90nm 20mhz fully nonvolatile microcontroller for standby-power-critical applications. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 184–185. IEEE, 2014.

[181] Alanson Sample et al. Analysis, experimental results, and range adaptation of magnetically coupled resonators for wireless power transfer. *IEEE Transactions on Industrial Electronics*, 58(2):544–554, 2011.

[182] Alanson Sample et al. Analysis, experimental results, and range adaptation of magnetically coupled resonators for wireless power transfer. *IEEE Transactions on Industrial Electronics*, 58(2):544–554, 2011.

[183] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, and Joshua R Smith. Design of a passively-powered, programmable sensing platform for uhf rfid systems. In *IEEE International Conference on RFID*, 2007.

[184] Emilio Sardini and M. Serpelloni. Self-powered wireless sensor for air temperature and velocity measurements with energy harvesting capability. *IEEE Transactions on Instrumentation and Measurement*, 60(5):1838–1844, 2011.

[185] Edward Sazonov et al. Self-powered sensors for monitoring of highway bridges. *Sensors Journal, IEEE*, 9, 2009.

[186] John Schlesak et al. A microwave powered high altitude platform. In *Microwave Symposium Digest, 1988., IEEE MTT-S International*, pages 283–286. IEEE, 1988.

[187] Thomas Schmid et al. Disentangling wireless sensing from mesh networking. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, HotEmNets '10, page 3. ACM, 2010.

[188] Stefano Scorcioni et al. A vibration-powered wireless system to enhance safety in agricultural machinery. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 3510–3515. IEEE, 2011.

[189] Young-Sik Seo et al. Wireless power transfer by inductive coupling for implantable batteryless stimulators. In *Microwave Symposium Digest (MTT), 2012 IEEE MTT-S International*, pages 1–3. IEEE, 2012.

[190] N. Shinohara. Wireless power transmission for solar power satellite (sps). In *Space Solar Power Workshop*.

[191] D.A. Shoudy et al. P3f-5: An ultrasonic through-wall communication system with power harvesting. In *Ultrasonics Symposium, 2007. IEEE*, pages 1848–1853. IEEE, 2007.

[192] Nuno M. Silva et al. Power management architecture for smart hip prostheses comprising multiple energy harvesting systems. *Sensors and Actuators A: Physical*, 202:183–192, 2013.

[193] F. Simjee and P.H. Chou. Everlast: Long-life, supercapacitor-operated wireless sensor node. In *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, Oct 2006.

[194] Amarjeet Singh et al. Mobile robot sensing for environmental applications. In *Field and service robotics*, pages 125–135. Springer, 2008.

[195] Joshua R Smith, Alanson P Sample, Pauline S Powledge, Sumit Roy, and Alexander Mamishev. A wirelessly-powered platform for sensing and computation. In *International Conference on Ubiquitous Computing*, pages 495–506. 2006.

[196] Ladan Soltanzadeh et al. Highly efficient compact rectenna for wireless energy harvesting application. *IEEE Microwave Magazine*, 14(1):117–122, 2013.

[197] Ladan Soltanzadeh et al. A high-efficiency 24 ghz rectenna development towards millimeter-wave energy harvesting and wireless power transmission. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(12):3358–3366, 2014.

[198] Philipp Sommer et al. Information bang for the energy buck: Towards energy-and mobility-aware tracking. In *EWSN*, 2016.

[199] Thad Starner. Human-powered wearable computing. *IBM Syst. J.*, 35, 1996.

[200] F. Su et al. Nonvolatile processors: Why is it trending? In *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017.

[201] S. Sudevalayam and P. Kulkarni. Energy harvesting sensor nodes: Survey and implications. *IEEE Communications Surveys Tutorials*, 13(3), 2011.

[202] Guodong Sun et al. Events as power source: Wireless sustainable corrosion monitoring. *Sensors*, 13(12):17414–17433, 2013.

[203] Hucheng Sun et al. Design of a high-efficiency 2.45-ghz rectenna for low-input-power energy harvesting. *IEEE Antennas and Wireless Propagation Letters*, 11:929–932, 2012.

[204] Affan A. Syed et al. Energy transference for sensornets. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 397–398. ACM, 2010.

[205] Yen Kheng Tan and S.K. Panda. Self-autonomous wireless sensor nodes with wind energy harvesting for remote sensing of wind-driven wildfire spread. *IEEE Transactions on Instrumentation and Measurement*, 60(4):1367–1377, 2011.

[206] J. Thiele et al. Smart sensors for small rodent observation. In *Sensors, 2008 IEEE*, pages 709–711. IEEE, 2008.

[207] Johannes Thiele et al. *Dynamic Wireless Sensor Networks for Animal Behavior Research*. INTECH Open Access Publisher, 2010.

[208] Array Of Things. Array Of Things. `https://arrayofthings.github.io/`, 2014.

[209] I.M. Tolentino and M.R. Talampas. Design, development, and evaluation of a self-powered gps tracking system for vehicle security. In *Sensors, 2012 IEEE*, pages 1–4. IEEE, 2012.

[210] Gilman Tolle et al. A macroscope in the redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63. ACM, 2005.

[211] Bin Tong et al. How wireless power charging technology affects sensor network deployment and routing. In *IEEE 30th International Conference on Distributed Computing Systems (ICDCS), 2010*, pages 438–447. IEEE, 2010.

[212] E.O. Torres and G. Rincon-Mora. Electrostatic energy harvester and li-ion charger circuit for micro-scale applications. In *49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS).*, volume 1, pages 65–69. IEEE, 2006.

[213] uBeam. uBeam. `http://ubeam.com/`, 2015.

[214] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 17, 2016.

[215] K. Vijayaraghavan and R. Rajamani. Novel batteryless wireless sensor for traffic-flow measurement. *IEEE Transactions on Vehicular Technology*, 59(7):3249–3260, 2010.

[216] H.J. Visser and R.J.M. Vullers. Rf energy harvesting and transport for wireless sensor network applications: Principles and requirements. *Proceedings of the IEEE*, 101(6):1410–1423, 2013.

[217] Hubregt J. Visser et al. Ambient rf energy scavenging: Gsm and wlan power density measurements. In *Microwave Conference, 2008. EuMC 2008. 38th European*, pages 721–724. IEEE, 2008.

[218] Voltree. Voltree Demonstrates the First Wireless Sensor Network Powered by Trees. `http://voltreepower.com/pressReleases/pr_firstWirelessSensor.html`, 2009.

[219] RJM Vullers, Rob van Schaijk, Inge Doms, Chris Van Hoof, and R Mertens. Micropower energy harvesting. *Solid-State Electronics*, 53(7):684–693, 2009.

[220] Jiafu Wan et al. M2m communications for smart city: An event-based architecture. In *IEEE 12th International Conference on Computer and Information Technology (CIT), 2012*, pages 895–900. IEEE, 2012.

[221] Ning Wang et al. One-to-multipoint laser remote power supply system for wireless sensor networks. *IEEE Sensors Journal*, 12(2):389–396, 2012.

[222] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, pages 149–152. IEEE, 2012.

[223] Benjamin H Waters et al. Innovative free-range resonant electrical energy delivery system (free-d system) for a ventricular assist device using wireless power. *ASAIO Journal*, 60(1):31–37, 2014.

[224] WiTriCity. WiTriCity. `http://witricity.com/`, 2015.

[225] Xuan Wu and Dong-Weon Lee. An electromagnetic energy harvesting device based on high efficiency windmill structure for wireless forest fire monitoring application. *Sensors and Actuators A: Physical*, 219:73–79, 2014.

[226] Tianyu Xiang et al. Powering Indoor Sensing with Airflows: A Trinity of Energy Harvesting, Synchronous Duty-Cycling, and Sensing. In *In Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys'13)*, page 16. ACM, 2013.

[227] Liguang Xie et al. On renewable sensor networks with wireless energy transfer: The multi-node case. In *9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2012*, pages 10–18. IEEE, 2012.

[228] Liguang Xie et al. Wireless power transfer and applications to sensor networks. *IEEE Wireless Communications*, 20(4):140–145, 2013.

[229] Guang Yang et al. Challenges for energy harvesting systems under intermittent excitation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 4(3):364–374, 2014.

[230] Jing Yang et al. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *SENSYS*, 2007.

[231] Lohit Yerva, Brad Campbell, Apoorva Bansal, Thomas Schmid, and Prabal Dutta. Grafting energy-harvesting leaves onto the sensornet tree. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 197–208. ACM, 2012.

[232] Jun Yi et al. An energy-adaptive mppt power management unit for micro-power vibration energy harvesting. In *IEEE International Symposium on Circuits and Systems, 2008. ISCAS 2008.*, pages 2570–2573. IEEE, 2008.

[233] Z-WAVE. Z-WAVE. `http://www.z-wave.com/`, 2015.

[234] Chengjie Zhang et al. Steam-powered sensing. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 204–217. ACM, 2011.

[235] Pengyu Zhang, Deepak Ganesan, and Boyan Lu. QuarkOS: Pushing the Operating Limits of Micro-powered Sensors. In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems*, 2013.

[236] Xiaoyu Zhang et al. An energy-efficient asic for wireless body sensor networks in medical applications. *IEEE Transactions on Biomedical Circuits and Systems*, 4(1):11–18, 2010.

[237] Chen Zhao et al. Powering wireless sensor nodes with ambient temperature changes. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 383–387. ACM, 2014.

[238] Wenxing Zhong et al. General analysis on the use of tesla's resonators in domino forms for wireless power transfer. *IEEE Transactions on Industrial Electronics*, 60(1):261–270, 2013.

[239] Dibin Zhu et al. Novel miniature airflow energy harvester for wireless sensing applications in buildings. *IEEE Sensors Journal*, 13(2):691–700, 2013.

Bibliography

[240] T. Zhu, A. Mohaisen, Y. Ping, and D. Towsley. Deos: Dynamic energy-oriented scheduling for sustainable wireless sensor networks. In *Proceedings of IEEE INFOCOM*, pages 2363–2371. IEEE, 2012.

[241] Ting Zhu et al. eShare: A Capacitor-Driven Energy Storage and Sharing Network for Long-Term Operation. In *sensys'10*, pages 239–252. ACM, 2010.

[242] ZigBee. ZigBee. `http://www.zigbee.org/`, 2015.

[243] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17, 1996.

[244] Jia Zou et al. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In *RTAS*, 2009.