N°d'ordre NNT : 2021LYSEI021

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
L'INSA de Lyon

Ecole Doctorale N° 512
Informatique et Mathématiques de Lyon

Spécialité/ discipline de doctorat :
Informatique

Soutenue publiquement le 29/03/2021, par :
Gautier Berthou

Operating system dedicated to NVRAM-based low power embedded systems

---

Devant le jury composé de :

Puaut, Isabelle	Professeur des Universités	Université de Rennes 1	Président.e
Torres, Lionel	Professeur des Universités	Université de Montpellier	Rapporteur.e
Lucia, Brandon	Associate Professor	Carnegie Mellon University	Rapporteur.e
Guérin Lassous, Isabelle	Professeur des Universités	Université Claude Bernard Lyon 1	Examinateur.rice
Sentieys, Olivier	Professeur des Universités	Université de Rennes 1	Examinateur.rice
Miro-Panades, Ivan	Docteur Ingénieur	CEA LETI	Examinateur.rice
Risset, Tanguy	Professeur des Universités	INSA LYON	Directeur.rice de thèse
Marquet, Kevin	Maître de Conférences	INSA LYON	Co-directeur.rice de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** http://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr INSA : R. GOURDON	**M. Stéphane DANIELE** Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX directeur@edchimie-lyon.fr
E.E.A.	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** http://edeea.ec-lyon.fr Sec. : M.C. HAVGOUDOUKIAN ecole-doctorale.eea@ec-lyon.fr	**M. Gérard SCORLETTI** École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 gerard.scorletti@ec-lyon.fr
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** http://e2m2.universite-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES secretariat.e2m2@univ-lyon1.fr	**M. Philippe NORMAND** UMR 5557 Lab. d'Ecologie Microbienne Université Claude Bernard Lyon 1 Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX philippe.normand@univ-lyon1.fr
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** http://www.ediss-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE secretariat.ediss@univ-lyon1.fr	**Mme Sylvie RICARD-BLUM** Institut de Chimie et Biochimie Moléculaires et Supramoléculaires (ICBMS) - UMR 5246 CNRS - Université Lyon 1 Bâtiment Curien - 3ème étage Nord 43 Boulevard du 11 novembre 1918 69622 Villeurbanne Cedex Tel : +33(0)4 72 44 82 32 sylvie.ricard-blum@univ-lyon1.fr
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 infomaths@univ-lyon1.fr	**M. Hamamache KHEDDOUCI** Bât. Nautibus 43, Boulevard du 11 novembre 1918 69 622 Villeurbanne Cedex France Tel : 04.72.44.83.69 hamamache.kheddouci@univ-lyon1.fr
Matériaux	**MATÉRIAUX DE LYON** http://ed34.universite-lyon.fr Sec. : Stéphanie CAUVIN Tél : 04.72.43.71.70 Bât. Direction ed.materiaux@insa-lyon.fr	**M. Jean-Yves BUFFIÈRE** INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 jean-yves.buffiere@insa-lyon.fr
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** http://edmega.universite-lyon.fr Sec. : Stéphanie CAUVIN Tél : 04.72.43.71.70 Bât. Direction mega@insa-lyon.fr	**M. Jocelyn BONJOUR** INSA de Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69 621 Villeurbanne CEDEX jocelyn.bonjour@insa-lyon.fr
ScSo	**ScSo*** http://ed483.univ-lyon2.fr Sec. : Véronique GUICHARD INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 veronique.cervantes@univ-lyon2.fr	**M. Christian MONTES** Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 christian.montes@univ-lyon2.fr

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Acknowledgments

Carrying out a PhD thesis is no easy task and there were helping hands in my close environment.

First, I would like to thank my advisors Kevin Marquet and Tanguy Risset for finding interesting conferences and journals to publish our work. They also enabled the fulfilment of my desire to formally prove systems by getting me in touch with Delphine Demange and Pierre-Évariste Dagand, whom I thank as well for their patience and for introducing me to formal methods. My colleagues at CITI lab offered me support and great memories, in particular Guillaume Salagnac, François Lesueur, Tristan Delizy, Guillaume Bono, David Kibloff, Jonathan Tournier, Yohann Uguen and others whom I shared laughs and scientific conversations with. The administrative personnel also did a great work at maintaining my ship afloat against procedures and prevented it from sinking down to the benthic floor. Special thanks to my thesis committee for their reviews and interesting discussions during the presentation and offline.

I would also like to thank friends and family, loving and supportive. In particular, I could have deep discussions with Jason Lecerf, we helped each other out in difficult moments and did live utter friendship regardless of the distance from one another. Gary Cottancin and Yannick Marion were supportive and had wholesome ideas, besides being available and always willing to go to gigs or on vacations together. I shared cheering freetime with expedition-comrades Pierre Godard, Douglas Raillard and Sergueï Lallement. Above all, we have had such nice moments together for almost ten years now, and it certainly helped the ship safely come ashore.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Embedded systems are increasingly used and their application range swiftly grows. Many objects become equipped with computation and communication abilities, thus widening further the family of embedded devices. Cloth tags, crop plant sensors, wearables, medical implants and sensors for hard-to-reach areas are examples of such items. More generally, this thesis focuses on constrained devices. The greatest constraints are form factor, *i.e.*, the volume occupied by the object itself, and physical access to the device once deployed. Within the scope of the present thesis, the considered objects are small and subject to limited maintenance. Indeed, an individual would not want a massive medical implant, for the benefits brought by the implant might be diminished by the discomfort it would generate. In addition, when deploying sensors in closed environments or hostile to the human body, such as inside walls or in radiation-covered areas, physical access to the devices is problematic and thus, must be avoided. Embedding batteries is limited to some extent, for batteries take space and require the intervention of a human operator from time to time, thus reducing the autonomy of the object itself. That statement also stands for rechargeable batteries. One gets the insight that alternative power supplies must be investigated to supplant the usage of batteries.

Full autonomy is achieved when the device is capable of harvesting energy from its environment. There are many sources of energy, and as many kinds of energy harvesters: light, electromagnetic waves, temperature, mechanic movement, motion of the living, air flow, *etc.* Harvesters yield a variable power, depending on the nature of the phenomenon to harvest, on environmental conditions and on the harvester itself. A small solar panel typically delivers less power than a wider one, however some applications might need a small form factor and hence have to cope with small harvesters.

Apart from solar energy, that usually provides long time windows of stable power, other sources do not allow that comfort. Eventually, power fails and the platform shuts down until the environmental conditions are favorable again. With weak energy sources, such as harvesting radio waves from a distant low-power device, the off-times may last hours or even days. When the energy availability of a platform is constrained to that extent, it becomes crucial to stop fighting against power outages but, on the contrary, to compose with them and to integrate off-times as inherent parts of the life of the platform, for power outages are unavoidable at that point.

Widespread state-of-the-art approaches use Wireless Power Transfer, that consists in purposing an external device to beam energy into the platform, as seen for credit cards where the card reader beams energy to whatever card reaches inside its range. Wireless Power Transfer, however, creates an unnatural environment, for the energy source is made on purpose for that deed, while that energy field would not be observed in nature otherwise.

This thesis hence sides with alternative solutions, that actually harvest energy from a natural environment. Unlike Wireless Power Transfer, a natural environment probably cannot deliver as much energy and thus, on-times are expected to run shorter and to be less stable as the power demands of the platform evolve alongside application progress. Hence, applications designed for natural energy harvesting can no longer afford to restart from the very beginning every time the platform reboots. As a result, considering the energy constraints that complement the size and access constraints, compliant platforms can only embed low-power components. Computation capabilities cannot exceed that of low-power micro-controllers and accelerators, hence limiting possible applications to sensor-related and monitoring features, alongside

basic processing.

Forward progress grows into a key concept of platforms that harvest energy from a natural environment, whereas it is always discarded within the credit card model. There needs to be a way of storing and loading forward progress so that the application can go on after every power outage. This is correlated to the concept of checkpointing. It creates a need for embedding non-volatile memories, that are memories which do not lose their contents whilst not powered. Non-volatile memories are usually purposed to store the application instructions and the initial values of data sections, often in read-only memories. On the other hand, the last decades witnessed the emergence of non-volatile Random Access Memories, that are designed to be used as regular, volatile Random Access Memories. These non-volatile memories are byte-addressable and, more importantly, have similar read and write access times than their volatile counterparts. Although they are still behind volatile Random Access Memories in terms of performance and do not yet have an infinite endurance, they are the most promising hope that energy-harvesting devices have to establish a persistent data storage.

Considering power outages is not straightforward in common programming languages. It raises a core question: how to handle every outcome induced by power outages into a programming paradigm? That question could be addressed in sundry ways, notably language-wise. This thesis proposes to sand off the sharp edge by providing a lightweight operating system support for traditional embedded software programming under intermittent power conditions. Hence, the resulting programming paradigm feels really close to what was done in battery-equipped embedded systems, without the discomfort of manually managing power outage constraints and without throwing away decades of programming experience.

Providing operating system support to spread application execution across power outages comes along with its share of challenges and trade-offs. Indeed, operating systems provide useful services for applications, but the price of that model in comparison to *ad-hoc* bare-metal baselines is an overhead due to function calls and to the genericity of the proposed services. It is legitimate to wonder whether such overheads are acceptable given the energy scarcity. This works shows that a lightweight operating system layer is actually worth using, for a rigorous automated resource and power outage management is beneficial in terms of development effort, run-time execution and confidence in application robustness.

Confidence in the behavior of such applications is a major concern when it comes to transiently-powered systems. Beyond testing systems against a finite benchmark suite, it seems important to take a step back and retrospectively analyze the concepts developed by such operating systems. The world of transiently-powered systems clearly lacks proof material to underpin the assumptions that those persistence mechanisms work and maintain an overall consistency to spread an application across power outages.

The present thesis identifies problematics related to intermittent power and proposes mechanisms for operating systems to solve those issues. An implementation of the investigated mechanisms within a lightweight operating system is Sytare. This work further investigates optimizations of such systems, brings a proof of correctness under intermittent power and builds an accurate energy consumption estimation tool.

Chapter 2 states the general problem and explores related works. Chapter 3 extensively studies existing operating systems for transiently-powered systems and positions them with respect to the problems depicted in Chapter 2. Chapter 4 describes in detail Sytare, an operating system for transiently-powered systems developed and improved during the course of the present thesis. Chapter 5 exposes a novel peripheral-aware energy model for embedded systems that encompasses changes in platform power state, a simple power measurement platform to populate the model and an accurate simulator for such platforms. The simulator both reproduces the behavior of the platform in software and hardware and yields time- and energy-related metrics using the model. Chapter 6 proposes an incremental checkpointing scheme using a common micro-controller peripheral as an optimization of former checkpointing schemes. Chapter 7 lays the groundwork for the first formal proof of correctness of checkpointing schemes for intermittent systems. Finally, Chapter 8 concludes this work and discloses some of its perspectives.

Chapter 2

Problem Statement and Related Works

Systems supplied with intermittent power must face new challenges with respect to continuously-supplied systems. Power outages are a kind of failure that makes almost every part of the platform fail, but not necessarily at the same time. They add complexity to run-time conditions and thus, the correct execution of long-running applications is not trivial to establish. The present thesis addresses the global problematic of being able to execute long-running applications despite power outages.

This chapter first discusses energy harvesting, energy sources, power managers and the outcomes of solely using harvested energy without an additional energy provider. Then, usual platforms that run under intermittent power are presented. A brief introduction to embedded programming and to embedded operating systems follows. Non-volatile memories are described after. This chapter then exposes recurrent problems that arise with intermittent power, either from intermittent properties themselves, or from application specifications. Those problems lay a baseline in the domain of transiently-powered systems and are referred to throughout this thesis. Finally, a categorization of existing hardware architectures equipped with non-volatile RAM is given.

2.1 Energy Harvesting

Harvesting energy consists in transforming an ambient physical phenomenon into energy intended to be used by some device. The conversion from the physical phenomenon into an electric pulse is the role of the transducer. Transducers are devices that transform one kind of energy into another. An interesting property of transducers is that they often work conversely with almost no modification. For instance, a speaker can be used as a microphone, a Light-Emitting Diode (LED) as a light sensor or harvester, *etc.* In general, apart from large solar panels, the raw output of the transducer cannot directly supply a device. Rectifier circuits are often required. Indeed, depending on the phenomenon and on the transducer, the raw output voltage may be high but cannot stand a current load, or both voltage and currents are low. The usage of a rectifier enables to convert the raw output voltage of the transducer into another voltage, more stable and suited to the device. The nature of the harvested energy, the performance of the transducer and the efficiency of the rectifier are parameters to study when designing a platform supplied by energy harvesting. Hence, figures exposed hereafter are for indicative use only, enabling to reason about orders of magnitude. In addition, many experiments have been conducted, at different time periods, using different technologies and devices, under different experimental conditions. Some works report the power density surface-wise, others volume-wise and the last simply report power. It is nonetheless not necessary to get an exact value as the orders of magnitude speak for themselves.

2.1.1 Energy Sources

There are many natural sources of energy and as many types of transducers to produce electric energy out of them.

Light Light is certainly the most popular natural source of energy. It may be harvested using photodiodes or photovoltaic cells and generate electrical power between 15 $mW.cm^{-2}$ and 100 $mW.cm^{-2}$ [1, 2, 3]. The light source may be natural as sunlight, or synthetic as indoor light or LEDs from another device. Visible Light Communication systems [4] could benefit from light harvesting using the same device for communicating and harvesting, *i.e.*, an LED. It was further shown that a red LED could deliver 133 µW under outdoor sunlight [5], however ambient light reduces light communication efficiency.

Radio and electromagnetic waves An important energy source is radio and other, non-visible electromagnetic waves. Energy is harvested using inductors and antennas. GSM can provide 0.1 $mW.cm^{-2}$ of electrical power, Wi-Fi delivers between 10 $nW.cm^{-2}$ and 10 $\mu W.cm^{-2}$ [1, 2] and a vibrating magnet 40 $pW.cm^{-2}$ [6]. The sensitivity of the harvester, that is the minimal radio power required for the harvester to operate, is the most critical and the most constraining property. A plethora of works proposes different designs for radio energy harvesting. Most of them can deliver 1 V DC output when the incoming radio power is at least -22 dBm (6 µW) to -14 dBm (40 µW) [7]. As of light, it would be interesting to build platforms that use the same antenna for both energy harvesting and communication [8].

Temperature Spatial and temporal variations of temperature may also be used to generate energy. A 5 K difference in temperature generates up to 60 $\mu W.cm^{-2}$ [9, 1]. The applications are numerous and notably wearables can take advantage of temperature gradients of the living.

Mechanic movement While some mechanical energy can be managed using magnetic waves, such as a vibrating magnet or coil, another technology may be used to convert mechanical energy into electrical energy. Piezoelectric properties achieve such a deed and may provide power, when struck or twisted, in the order of magnitude of 200 $\mu W.cm^{-2}$ to 500 $\mu W.cm^{-2}$ [9]. People move and body movement can thus provide energy. For instance, a heel strike, a finger motion and even blood pressure.

Air flow The flow of gas, like any fluid, can generate energy using, for instance, an anemometer or a turbine. Wind turbines may produce 28.5 $mW.cm^{-2}$ [1]. Human breath can generate up to 0.4 W [3] with a face mask as energy harvester.

2.1.2 Power Outages

Some energy sources are quite stable, for instance, when using large solar panels. This thesis addresses small objects with, among others, form factor constraints and thus, cannot embed large energy harvesters. As a consequence, regardless of the harvested energy source, the harvested power is expected to be low. More importantly, the harvested power is expected to be lower than the power consumption of the supplied platform in active mode. Active mode depicts an operating mode of the platform when it performs some operation, *e.g.*, the micro-controller is computing or a piece of data is being sent over radio. Sleeping modes may consume less instantaneous power than harvested, but performing any useful operation would result in energy depletion as energy is consumed faster than harvested. Inevitably, that situation incurs power outages. Given that there is no degree of freedom regarding form factor, considering medical implants for example, the platform must then face and cope with power outages as an inherent part of its life. Furthermore, battery-less platforms are designed to overcome the limitations of replacing batteries over time, hence they are destined to be deployed once and never, or seldom, physically accessed again. Battery-less platforms are *de facto* on their own regarding energy supply and must thus accept power outages.

2.1.3 Power Managers

Aside from the transducer itself, an energy harvester embeds some circuitry, including voltage rectifiers and power managers. The harvested energy may be used as is or be aggregated into a small storage such as a small-sized capacitor. A power manager may take responsibility for handling the energy storage: it dictates when to supply or not the platform. It also provides a befitted power supply to the platform.

Energy Management Policy

There are two major policies of powering the platform: *harvest-use* and *harvest-store-use* [10, 3, 11, 12, 13, 14]. Under the harvest-use policy, the platform is directly wired onto the harvester. It is simple and may lead to the smallest form factor designs. However, the harvester must provide a high voltage, *i.e.*, voltage above the operating threshold of the platform plus the voltage drop due to the current consumption of the platform. When the output of the harvester drops below that threshold, the platform is abruptly turned off. The other policy, harvest-store-use, introduces an energy storage, such as small capacitors or super-capacitors. The energy storage acts like a low-pass filter, it enables the platform to be powered more steadily than under the harvest-use policy. Using a harvest-store-use approach allows the usage of smaller harvesters and of lower harvested power levels. The downside of harvest-store-use is that the energy storage must charge up to a certain energy level to power the platform and, depending on the capacity of the energy storage, the charging phase may take time. The storage element is important for, in practice, it is not ideal and presents current leakage. Its capacity must be carefully chosen for the rate at which it charges and discharges is directly incurred to capacity. A trade-off between reactivity and the ability of being powered on during long times must be established [15]. UFoP [16] derives this methodology and proposes to decouple capacitance by using several smaller capacitors: one for the micro-controller and one for every single peripheral.

A hybrid between harvest-use and harvest-store-use has been designed to reconcile both approaches. Confusedly named either harvest-use(store) [17, 18], harvest-store(use) [19] or harvest-store-use [20], it tries to use the harvested energy in a harvest-use manner as much as possible and stores the potential exceeding harvested energy into an energy storage. Once the harvester can no longer provide sufficient power to the platform, the energy storage takes over so that powered periods are lengthened.

The three power management policies can be implemented in numerous ways. The bulk of existing works candidly wire the transient power source directly onto the platform to supply, regardless of the chosen policy [21, 22]. The platforms thus have to cope with a supply voltage that evolves over time. Fluctuating supply voltage impacts clock frequencies and current consumption [23]. Hence, while directly wiring a transient power source onto the platform still enables the execution of applications, there are some pitfalls to avoid. First, all electronic components on the same platform do not necessarily have the same voltage requirements and the scenario where the power supply is able to supply a given component but fails to supply another may happen. For instance, low-power micro-controllers usually operate at 1.8 V or even less, whereas radio chipsets might require at least 2 V. In that case, if the voltage of the power supply, in either of the harvest-use, harvest-store-use or harvest-use(store) modes, stays between 1.8 V and 2 V for some reason, the software application might think that the platform is working well since the micro-controller is able to execute instructions, however any attempt to use the radio chipset would inevitably fail. Another pitfall is that varying supply voltage complexifies the task of estimating the energy consumption of a system, should it be software or hardware.

In order to tackle the two downsides of directly wiring the power source onto the platform to supply, some works propose to add a voltage regulating stage between the energy storage and the platform in a harvest-store-use scenario [24, 25, 26]. This approach is also adopted by industrials [27], even for very constrained energy sources such as ambient radio waves. [1] Using a voltage regulator adds little complexity to the circuit, but introduces an additional component which efficiency is, in practice, not ideal. Some energy is wasted due to the fact that voltage regulator are not ideal, yet it is viable to design harvesting systems with a voltage regulator. A Game Boy that harvests solar energy and button press energy was designed thus [28].

Improving Harvesting Efficiency

Regardless of the kind of energy source, power managers also work close to the transducer in order to find an optimal configuration that would maximize the harvested energy. They often embed a Maximum Power Point Tracking (MPPT) circuit that is designed for that purpose. Harvesters are studied against I-V curves, *i.e.*, the relationship between voltage and the current that can be drawn from the transducer. These curves have peaks and MPPT circuits work towards the objective of getting closer to the peaks. MPPT consists in dynamically adapting the impedance seen by the transducer [29, 30]. To do so,

[1] `https://e-peas.com/types/energy-harvesting/rf/`

DC-DC converters can be used for their duty-cycle is controllable and directly affect the impedance. In order to compute the optimal impedance and duty-cycle, the MPPT circuit must first locate the instantaneous performance of the circuit on the I-V curve of the transducer. Traditional techniques of MPPT involve periodically disconnecting the transducer to automatically measure its open-circuit voltage. The fractional open-circuit voltage technique simply models the optimal voltage to be linear with the open-circuit voltage. The coefficient of that linear relationship is dependent on the transducer. For instance, a photovoltaic cell may optimally work at 75 % of its open-circuit voltage. Finally, measuring the open-circuit voltage must not be performed very often nor for very long, since the transducer is disconnected from the rest of the circuit and thus cannot deliver its energy to the platform or to the energy storage, if any.

2.2 Traditional Platforms with Intermittent Power

Today's most widespread devices that harvest energy are devices such as Radio-Frequency IDentification (RFID) and Near-Field Communication (NFC) tags [31]. Throughout this chapter, such tags are informally referred to as credit cards. Credit cards embed a micro-controller programmed with an application. The application has to run to completion without power outage. Upon reboot, the application restarts from the beginning to initiate a new transaction. In order for that model to be viable, a card reader steadily provides a sufficient amount of energy to perform the transactions, without interruption during the entire process. Hence, the credit card model is a two-fold approach that requires (*i*) the credit card itself to communicate information and perform some operations and (*ii*) an external device that beams energy onto the credit card. The presence of the external device, here the card reader, is a very strong assumption for the credit card does not harvest energy from a *natural* environment, but rather from an environment *made on purpose* for the credit card. This is called Wireless Power Transfer. Credit cards may thus be energy neutral as defined in Section 2.3.2, however the card reader is not.

Harvesting radio energy often generates small power for the device may be far away form the source or health norms would not enable higher transmission power. PoWiFi [32] observes that Wi-Fi router have irregular traffic over time and repurposes Wi-Fi networks for power delivery. Wi-Fi routers are modified to inject superfluous broadcast traffic, called power packets, on available and non-overlapping Wi-Fi channels. An advantage of using Wi-Fi as the energy source is that the antenna may be used for both communication and energy harvesting. PoWiFi requires a modification of Wi-Fi traffic and thus is not entirely natural.

Ambient backscatter enables an efficient energy harvesting of radio power that does not require the injection of any additional energy source [33]. It comes from the observation that radio waves, especially television (TV) broadcast waves, are present in most locations. TV waves, in particular, have a large coverage since TV towers are designed for that purpose. The devices may thus harvest TV energy, without backup battery, and the communication between devices may be achieved by backscattering the TV signal. Changing the impedance of the antenna between two impedance values defines an encoding of zeros and ones, for a matching impedance would absorb the signal while a mismatching impedance would reflect it, hence backscatter it. Given that TV energy is always present and not initially designed for harvesting devices, it may be considered almost natural, albeit not being a natural phenomenon unlike sunlight.

2.3 Embedded Programming

The software is often co-designed alongside the hardware. Programming on embedded targets differs from programming on desktop targets. Apart from the computing capabilities being different, the experience of developing is not the same, notably if the software is meant to be executed in a bare-metal manner, *i.e.*, there is no operating system to rely on.

2.3.1 Programming Basics

Throughout this work, a certain amount of programming notions are referred to, notably CPU registers, peripheral control registers and stack; heap to a lesser extent.

The micro-controller contains, among others, a Central Processing Unit (CPU). The CPU itself contains registers which are fast memory units not mapped in memory. Most of the CPU registers are for general purpose; *e.g.*, arithmetical operations. A few of them are rightfully called special registers. The *program counter* contains the address of the current instruction. The *status register* contains information about run-time execution, such as the interrupt status (*i.e.*, enabled or disabled) and the condition flags used for branching. The *stack pointer* contains the address of the topmost part of the stack.

Stack is a portion of main memory dedicated to storing some of the local variables and information of functions and interrupt handlers. Stack may be ascending or descending, respectively grows towards higher or lower addresses. Orthogonally, stack may be full or empty, the stack pointer respectively points to the actual topmost item or points to the next location after the topmost item. Any combination of these two axes may exist and it is more a matter of implementation details, since the `push` and `pop` instructions, that respectively put an item on top of the stack and retrieve the topmost item, manage the stack pointer accordingly to the stack model of the CPU. At the end of the day, the only concerns about stack management, that truly depend on the stack nature, are its initial value and specific management operations (*e.g.*, stack lookups without popping any value). These are concerns for a bare-metal bootloader or for an operating system, even the most basic ones, but not for the application itself, whose stack is usually managed by a compiler.

A program binary is usually split into several sections. The `.text` section contains the program instructions. Most of the time read-only, it may though be modified if the application needs that feature. Global variables are spread onto two sections: the `.bss` sections for global variables which initial value is zero and the `.data` section for the other global variables. Since the variables of the `.data` section have a non-zero but pre-defined initial value, these variables are *relocated*. The relocation section contains the initial values of all the variables from the `.data` section, in the same order and respecting variables sizes and alignment. The relocation section is allocated apart from the `.data` section, so that the application only works on the `.data` section and the relocation section is read-only. The implicit contract of run-times is that an external entity is responsible for zeroing the `.bss` section and copying the relocation section into the `.data` section, at run-time and before the application actually starts. This is one of the roles of bootloaders and operating systems. The stack is also a section on its own, since it must be reserved for stack purposes. As mentioned, the initial value of the stack pointer must be set before the application starts.

In bare-metal programming as well as in operating system programming, section locations and sizes must be carefully studied. Relying on the compiler, section mapping is usually achieved by handcrafted *linker scripts* that contain memory placement indications for the linker stage of the binary production. Specifically, the linker script must at least place the bootloader code at the address corresponding to the reset value of the program counter, defined by the micro-controller manufacturer.

Heap is a portion of memory dedicated to dynamically-allocated memory, whereas `.bss` and `.data` sections are statically-allocated, *i.e.*, allocated by the compiler. Heap comes with a heap manager, that proposes a certain amount of services. Amongst the most essential services are the memory allocation request, *e.g.*, `malloc`, and the memory liberation request, *e.g.*, `free`. The main concerns with heap is that heap memory becomes segmented over time. Memory segmentation may prevent the system from providing a memory block of the requested size at some point, resulting in a failure. It also makes the memory allocation and liberation slower. Hence, heap memory is often frowned upon by embedded developers albeit highly used in non-embedded programming. Static allocation is more predictable and faster, thus admittedly a good choice for embedded systems.

2.3.2 Intermittent Programming Paradigms

Due to the scarce nature of the harvested energy, using low-power hardware is not sufficient in essence and the software must be energy-efficient by managing the different power-consuming components in a clever way.

Energy neutral systems only use the energy harvested from the environment and do not use any other energy source. Energy neutrality is a property that can be achieved in sundry ways. It depicts a large spectrum including tiny harvest-use systems on one hand, and larger systems equipped with batteries and larger harvesters on the other hand [34, 35]. Only smaller platforms are studied in this thesis, for they have to cope with harsher run-time conditions. The harvest-store-use and harvest-use(store) policies are

closer to the battery-powered systems, however the energy storage can be as small as a simple capacitor of a few dozens microfarads, which is orders of magnitude smaller than complex batteries of several ampere-hours. Existing battery-less systems usually bet on power outage scarcity, due to an efficient energy management and an energy harvesting source capable of delivering enough power to allow an almost infinite execution of the application. A cell-phone [36], eye-tracking goggles [37] and a video streamer [38] are examples of such battery-less systems. In order to reduce energy, these solutions directly wire the raw output of the sensor to a backscatter emitter. Encoding is simple and filtering is performed by the reader platform, which itself is not energy-constrained. Hence, these applications remain rather simple. Power outages occasionally occur, however these applications allow to be restarted from the beginning every time, as long as the power outages are not too frequent regarding the application performance expectations. More complex applications cannot be satisfied by these power-related conditions for they would require some mechanisms to spread across power outages.

Normally-off computing is a paradigm designed to power off components as soon as they are no longer needed [39]. Any component is concerned, should it be the micro-controller or any peripheral. Special attention must be drawn to the power-on times of each component though, for some hardware concerns such as clock stabilization may require dozens or hundreds milliseconds. Ultimately, the whole system can be turned off while waiting for an external event to occur for instance. Power outages are thus inherently part of the design. The usage of non-volatile RAM makes normally-off systems feasible as the memory now needs to be powered on only when reading or writing to it, and keeps its data despite power outages, while the fully volatile counterparts would need to be kept powered on even when no operation is performed on it. Considering the unfortunately still standing performance gap between volatile and non-volatile RAMs discussed in Section 2.5.2, normally-off systems may also embed volatile memory in addition to a volatile micro-controller and a memory hierarchy must be defined. To that extent, normally-off systems need a memory policy and need to guarantee consistency within heterogeneous memories.

2.4 Embedded Operating Systems

After being introduced to embedded programming in general and in a bare-metal manner, one must consider that operating systems for embedded and constrained platforms do exist. Operating Systems provide a set of services to the applications they host. Amongst the most basic services are peripheral handling. The applications access the peripherals of the platform using the operating system's services, through a so-called Application Programming Interface (API). Operating Systems are also responsible for the boot sequence of the hardware platform as well as the software environment in which are executed the applications.

Operating systems generally support several applications, or *processes*, to execute in parallel, even if the micro-controller is not multi-threaded. This is called *scheduling*. The operating system keeps track of the currently running processes and constantly has to choose, at run-time, how much time or energy should be given to each process. Processes may be prioritized or, on the contrary, given the same proportion of execution time.

More sophisticated operating systems isolate processes so that each process cannot access the memory of other processes. More security-related services might be offered by the operating system, depending on its complexity and on whether it is relevant for the application that should run on the platform.

2.4.1 Common Operating Systems

The most widespread operating system for embedded platform is likely any distribution of Linux that can execute on the considered platforms. For instance, Linux is commonly used in single-board computers. Single-board computers are complete systems, including micro-processors, memory and peripherals, that are embedded into a single board, as opposed to traditional computers that have a mother board and many daughter boards for modularity purposes. They are often about as small as credit cards, albeit not as flat. Popular single-boards include, and are not limited to, the Raspberry Pi, Orange Pi, Banana Pi, BeagleBoard and Odroid platforms. Single-board platforms tend to increase their computation capabilities with multi-core micro-processors, higher bus speeds, more RAM, *etc.* Today, the average single-board platform has a quad-core micro-processor and several gigabytes of volatile RAM, enabling

the execution of Linux-based systems. The kernel is often stored in a removable SD card. Some of them even embed Graphics Processing Units. The recent Nvidia Jetson Nano platforms embed 128 Maxwell graphical cores and consumes up to 5 W. Single-board platforms are widely used for teaching purposes and in robotics. They need a power supply capable of delivering several amperes. The energy demands of such platforms are a few orders of magnitude superior to those of the transiently-powered systems considered in this thesis.

A Linux-based operating system requires hundreds of megabytes of RAM, as well as hundreds of megabytes of storage. Of course, the actual requirements of Linux heavily depend on the considered distribution. However, even the most lightweight distributions need a fair amount of memory and this is to take into account when designing embedded platforms. Within the context of transiently-powered systems, today's non-volatile RAM technologies do not allow that much memory within small form factors.

Other operating systems target smaller devices. RIOT [40] fits in a few kilobytes, for both instruction code and RAM usage. RIOT supports multi-threaded applications. Its priority-based scheduler and its low interrupt latency also widen the applications of RIOT to real-time applications. Some operating systems incorporate real-time requirements as an inherent part of their design. A key challenge for real-time embedded systems is to provide deterministic services that execute for a known amount of time, regardless of the state of the platform. FreeRTOS provides a threading library that aims at ensuring a correct execution of a well-specified application. µC/OS-II [41] and µC/OS-III [42] are popular real-time preemptive operating systems. Zephyr is part of the Linux Foundation and inherits some concepts from Linux, including device trees. Unlike traditional Linux distributions, Zephyr provides a real-time environment. Finally, VxWorks is also very popular in the world of real-time systems. However, VxWorks is designed for embedded systems with higher capabilities, such as automotive and robotics.

Although these smaller operating systems consider hardware constraints that arise with the nature of embedded systems, they are still complex for smaller energy-harvesting devices. Furthermore, although real-time properties are useful for many applications, the very definition of time, and more specifically of real-time constraints, is substantially altered and does not have the same semantics when off-times become part of the application.

2.4.2 Legacy Operating Systems for Constrained Platforms

TinyOS [43] targets sensor networks of constrained nodes that must be reactive to events. It proposes an event-driven system as an alternative to multi-threaded systems. The application is designed as separate tasks that are run to completion, *i.e.*, tasks do not hang indefinitely and or not preemptable. Tasks can be spawned upon event occurrence, either external or initiated by other tasks.

Contiki [44] primarily focuses on the ability to deploy applications and updates onto an entire network. Contiki aims at being used in constrained platforms. It thus proposes a library of protothreads [45] to simulate a multi-threaded environment on a single stack. When a protothread blocks, the stack is rewound to enable another protothread to execute. Contiki's model differs from that of TinyOS, mainly due to the protothreads being preemptable. In TinyOS, on the contrary, each task must execute to completion before another is scheduled and thus, lengthy computations may impede the overall performance of the task set and the reactivity of the system to external events.

Dewdrop [46] is a task scheduler, designed for computational RFID systems, that leverages energy-related data to make decisions. The platform is kept in deep sleep when either no action is required or no activity can be scheduled due to the instantaneous energy level. Taking energy concerns into account improves the efficiency of the computational RFID platforms, that can operate at longer ranges. Similarly to TinyOS, Dewdrop asks the application developer to decompose the different activities of the application into separate tasks, each task being designed to run to completion without interruption. However, Dewdrop's task model is more restrictive and more suited to RFID applications: the same task is repetitively executed. Dewdrop aims at starting the execution of the task at the optimal energy level, given the challenge that the radio energy source is unpredictable. QuarkOS [47], built on top of Dewdrop, attempts to address the issue of the long tasks by proposing a decomposition of the tasks into sub-tasks, yet still designed to run to completion without interruption. The platform may enter deep sleep between consecutive sub-tasks, in order to recharge the energy storage in the meantime. The granularity of a sub-task can be as fine as transmitting a single pulse belonging to a bit within an OOK-modulated packet.

At coarser granularity, a sub-task could, for instance, compute a single pixel from a camera sensor being read. This, of course, assumes that the activities are divisible, which constrains the network protocols and the sensing operations. However, an application may run on QuarkOS with a capacitor as small as 0.1 µF.

TinyOS and Dewdrop assume that every task executes to completion before another can execute. Contiki slightly relaxes that assumption by allowing protothreads to be preempted only when explicitly requested by the application code or upon completion. These assumptions are broken when troublesome imminent power outages, that come along with intermittent supply, disturb the run-time conditions. Instead of trying to support intermittent power as a software wart inside existing operating systems that are not designed for such energy constraints, it thus becomes imperative to design operating systems, or at least helpers, for applications for transiently-powered systems. Chapter 3 and Chapter 4 comprehensively study such operating systems designed for transiently-powered systems, against the problematics identified in Section 2.6.

2.4.3 Operating Systems for Energy-Harvesting Systems

Energy-harvesting systems are not regular embedded systems. They, or at least the low-end of the energy-neutral systems, are smaller in size and thus in available power, computation capabilities and RAM size. It is unlikely to see a Linux-based kernel for transiently-powered systems powered by harvesting radio wave in a short-term time-span, for instance. It may happen in a distant future, if the advances in non-volatile RAM technologies and energy harvesting technologies allow it. Before this point is reached, more modest operating systems must be designed to provide services for applications that would execute on such platforms.

Bailey *et al.* discuss the outcomes of integrating non-volatile RAM as an inherent part of the computers, as far as operating systems are concerned [48]. The discussion is not specific to energy-harvesting systems but scales to any system equipped with non-volatile RAM. Depending on how the non-volatile RAM is integrated into the system design, the memory hierarchy may be questioned, alongside the concepts of paging and memory protection. The run-time environment of the software applications completely changes since the notion of stopping the execution of an application is blurred by the potential neverending run-time. Distributing software also might change for the relevance of the concept of installer as defined today is jeopardized and updating software while currently executed may be tricky. Also, fault management could also be impacted for faults made persistent by the usage of non-volatile RAM are problematic throughout the life-time of the application, initially designed to be infinite.

With these problematics in mind, alongside the ones exposed in Section 2.6, Chapter 3 provides an extensive study of operating systems for transiently-powered systems.

2.5 Non-Volatile Memory

The main property of non-volatile memories is data retention, even when not powered. Non-volatile memories depict a large spectrum of technologies and applications, from the slowest ones such as hard disk drives to faster ones like the recent non-volatile RAMs.

Non-volatile RAM, besides being non-volatile, are almost as time-efficient as SRAM and as dense as DRAM [49]. As stated by Bailey *et al.*, the performance of non-volatile RAM is so close to volatile RAMs that it questions the traditional memory hierarchy that makes computations in the fastest volatile memory while the slowest non-volatile memory only served as a data storage to repopulate the volatile memory after exiting an application [48].

In general, memories are required to address a major challenge: how to encode zeros and ones inside the material. Also, the endurance of the memory, *i.e.*, the amount of accesses that are supported, is an important criterion, notably for transiently-powered systems with a potentially infinite life-time. It is not specific to non-volatile memories and each kind of technology comes up with an innovative way to store data and make the distinction between two logical states. More specifically to non-volatile memories, the memory chip must be designed to keep its contents intact while the power is off. This is a complex challenge for the memory can be off during long times and anything could happen during that

off-time. Temperature and surrounding magnetic field are examples of environmental parameters that can change over time. Furthermore, constrained embedded systems often have geometric constraints, hence the chosen technologies must achieve high density, *i.e.*, small form factor. A subsidiary challenge is the financial cost, but that last concern is out of scope of the present thesis.

2.5.1 Traditional Non-Volatile Memory

Non-volatile memories have existed for a long time and are well integrated into computer designs. Used to store software instructions, data or application save-states, they are common components.

ROM memories Read-Only Memories (ROM) are read from but not written to. They can be implemented in sundry ways. For instance, a diode matrix is literally not writable for diodes are not programmable. The matrix is manufactured and from that point onward, its contents are frozen.

Other kind of ROMs emerged, including the Erasable Programmable ROMs (EPROM) and Electrically Erasable Programmable ROMs (EEPROM). They use transistors and thus are theoretically reprogrammable, but the process is either very slow, hardly reachable physically, or the memory itself has a poor endurance and is best used as a read-only memory. EPROMs can be erased using a strong ultraviolet beam. One-Time Programmable memory derived from the EPROM, with the removal of the ultraviolet erasure feature.

Flash Flash is based on the design of an EEPROM. A Flash memory can be read at random addresses without penalty. However, the writes must obey page constraints that do not allow writes to random addresses. Indeed, a Flash memory is composed of several non-overlapping sectors of addresses. The sector to which belongs the address to be written to must be erased and the entire sector must be populated again, with the new value updated. This makes Flash memory a good candidate for read-only sections such as software instructions and constants and, to a lower extent, to seldom-updated variables. However, Flash memory, besides the slow and constrained write protocol, has limited endurance and both downsides makes Flash memory unsuitable to be used as a working memory.

Sundry variants of Flash memory exist, including NOR and NAND, named after the logical gates that their internal gates are analogous to. NOR memories try to relax write constraints, allowing random access writes as long as the values to be written either coincide with the value already present or solely put zeros in the memory. The erased state of a bit is the logical one and the NOR memories allow random writes of zeros. On the other hand, writing a one, *i.e.*, resetting a bit, still involves resetting the entire sector. NAND memory does not allow any random write access, but has a higher density than NOR memory.

Hard disk drives Hard disk drives store data in a ferromagnetic material, in the shape of a disk. Reading and writing is performed using magnetic fields created by a magnetic head. The magnetic head points to a particular point of the disk, thus mechanical action is required to access all locations of the disk. Hard disk drives require high supply voltage, up to 12 V and consume about a few watts, making them unfit for transiently-powered systems.

2.5.2 Non-Volatile RAM

Non-Volatile RAMs are more recent than the aforementioned non-volatile memories. Their ultimate objective is to catch up those older memories, in terms of access latency and density. While read and write accesses are slower than volatile RAM, they are nonetheless closer to that of RAM rather than that of traditional non-volatile memories. They are also quite endurant. Manufacturers display figures as large as 10^{12} or 10^{16} writes, though not infinite yet a sound base for transiently-powered systems [49].

There is a tremendous amount of technologies investigated as many implementation alternatives of the same global specification. They all have different properties and they have been extensively compared against one another [49, 50]. Some of the technologies appear to be more difficult to design and to manufacture than others and thus, they do not have the same maturity. The most promising technologies, such as Spin Torque Transfer RAM, are slowly emerging but still did not reach a large scale commercialization

and availability today. On the other hand, other non-volatile RAM technologies, such as Ferroelectric RAM, have already been widely available and usable for low-power platforms for years.

Phase-Change Memory Phase-Change Memories (PCM) leverage the physical properties of materials derived from a chalcogenide glass. The chalcogenide glass material is heated at specific temperature levels in order to control its resistivity. Hence, PCMs are resistive. The heat peak sets the physical state of the chalcogenide. As long as the material remains in that state, the memory information is kept consistent. This property is thus used to encode zeros and ones. Indeed, when used as a memory, the material is locally kept either in a high-resistance disorder amorphous state, or in a low-resistance crystalline state. A bit can be written by provoking the adequate heat peaks. A bit is read by measuring the current flowing through the memory cell when a fixed voltage is applied. PCMs have a high write latency because of thermic concerns and a low-write endurance, although better than Flash. Due to the temperature-related properties of the material, PCMs are not suited to all environments.

Ferroelectric RAM A Ferroelectric RAM (FRAM or FeRAM) cell consists of a 1T1C structure: one transistor and one ferroelectric capacitor. The polarity of the capacitor is used to encode zeros and ones. It may be changed by applying a voltage to the capacitor. Polarity retention is achieved thanks to the hysteresis properties of the ferroelectric capacitor. In order to read a bit, the memory controller tries to write a pre-defined value to the bit. If the cell state changes, which is visible as a current pulse, then the read value was the opposite value of the newly-written one. Otherwise, the read value was the same value. This means that the read operation is destructive, so a bit which value is needed after the read operation must be written over again. Despite the destructive read that slows down accesses to the memory, ferroelectric memory is yet considered fast. In addition, it has small power requirements and may operate at low supply voltage. It also has a good endurance, altogether making ferroelectric memory a fair candidate for low-power energy-harvesting platforms. Texas Instruments made the choice of replacing the entire Flash memory by ferroelectric RAM in their MSP430FR micro-controller family.

Resistive RAM This kind of resistive memory leverages the property of a dielectric to become conductive by applying a sufficiently high voltage. A Resistive RAM (RRAM or ReRAM) cell, otherwise considered a memristor, consists of two conductive layers insulated by a dielectric material. A high voltage provokes the creation of a conductive filament between the two plates, which reduces resistance. In practice, the resistance varies as an effect of voltage magnitude and polarity that is applied to it. Its non-volatility is a consequence of its resistance not changing when the memristor is not supplied. To determine whether the memristor contains a logical zero or a one, the current flowing through the memristor is compared to the current flowing through a fixed reference resistor while supplied with the same voltage. RRAM has a high density thanks to its grid-shaped structure. There exists several variants of resistive RAM. The Oxide-based RAM (OxRAM) uses voltage polarization instead of voltage magnitude. Based on the polarity applied to the memory cell, a conductive filament is either created or removed between both conductive layers. Hence, the resistance of the whole memory cell changes following the polarity of the voltage applied to it. The 1T1R-RRAM technology comprises one transistor and one resistor, where the resistor is the memory cell itself. It has improved density.

Magnetic RAM Magnetic RAM (MRAM) leverages electron spin to encode zeros and ones. The flow of electrons inside a ferroelectric material is modified by external magnetic fields and the perturbation depends on the spin state of each electron. At macroscopic level, this results in a variation of overall resistance. Two ferromagnetic plates are separated by an insulating dielectric layer. This is called a Magnetic Tunnel Junction [51]. One ferromagnetic layer has fixed magnetization direction while the other one may vary. When the magnetization of both layers are equal, the resistance of the memory cell is low. On the contrary, when the magnetization of both layers are opposite, the resistance is high. The Spin Torque Transfer MRAM (STT-MRAM) uses a spin-polarized current to change the orientation of the magnetic field. Spin-Orbit Torque MRAM (SOT-MRAM) isolates the read and the write lines at the cost of using an additional source of magnetic field for determinism [52]. SOT-MRAM thus achieves better endurance, density and lower latency that STT-MRAM. A general issue of magnetic RAMs is that, to some extent, it will be difficult to increase density since magnetic field could impact neighboring cells.

Nanotube RAM Nanotube RAM (NRAM) embeds a matrix of carbon nanotubes. Nanotubes may either touch another nanotube or not. Contacts between nanotubes impact the resistance of a given nanotube, thus encoding zeros and ones. Non-volatility arises from the combination of the mechanical properties of the nanotubes in the absence of contact, and the Van der Waals force operating locally at contact points [53].

2.6 Intermittent Computing Problematics

The main constraint that intermittent computing adds to traditional continuously-powered computing, is the occurrence of power outages. Volatile memories are not designed to retain their contents when not powered. Hence, their contents can be considered completely lost upon power outage. In practice, depending on the memory properties, data retention may last for a small duration and *some* of the data *might* be unaltered if the power went back on soon enough after the power outage [54]. However, volatile memories only guarantee data retention as long as they are powered on, so it is correct to assume that their contents can no longer be trusted upon power outage and it is incorrect to assume that their contents are preserved, even partly.

In regular credit card scenarios, the application would always run from the beginning every time the platform was powered on. Indeed, the platform assumed that an external reader would beam enough power to perform an entire transaction in a contiguous manner, without any power outage.

Transiently-powered systems do not leverage this assumption and must thus accept power outage as part of the platform's life. Any application is likely to last longer than what a small energy harvester or energy storage could provide to the platform. This observation lays the ground for the problems depicted hereafter. Specifically, an operating system for transiently-powered systems must address the volatility of the CPU and the memory (P1), the volatility of peripherals and interrupts (P2), timeliness and atomicity constraints (P3), while guaranteeing non-volatile memory consistency (P4).

2.6.1 Checkpoint Definition

Throughout this thesis, a *checkpoint* depicts a state where all system components are consistent with each other. The system components are the micro-controller, volatile memories and non-volatile memories if applicable, as well as internal and external peripherals.

The state may be saved, either in a single step or several steps, every component altogether or separately, actively or passively by using the persistence properties of non-volatile memory for instance. The mechanisms designed to save the system state are referred to as *save*.

This state should be loadable in some way, this corresponds to the *restoration*. In the best case possible, the exact point where the platform stopped running because of power outage has all system components consistent and thus, this point itself is a checkpoint. In that case, restoration simply consists in resuming the application exactly where it stopped when power failed. However, in the general case, consistency between system components is likely to require the system to roll-back to a safe checkpoint prior to the point where power failed, as not all nodes in the application's control-flow graph are valid candidates for checkpoints.

In addition, the concept of *checkpoint*, alongside the notions of *save* and *restoration* techniques, are called *checkpointing*.

Some works, especially the ones that leverage a task-based approach, claim that they do not perform nor use checkpoints [55]. However, the definition of checkpoint proposed in this thesis is more generic as it encompasses any technique that isolates points of whole-system consistency. Even for task-based approaches where partial task progress is always discarded, for task beginnings are checkpoints [56, 55, 15, 57]. Hence, according to this very definition of checkpoint, all the works that aim at ensuring long-running application's forward progress across power outages leverage checkpointing.

Definition: Making a Piece of Data Persistent A piece of data is *made persistent* when it is managed by the checkpointing system. The checkpointing system is responsible for saving its contents and restoring it, as well as the consistency of the retained state with respect to the program counter of resumption.

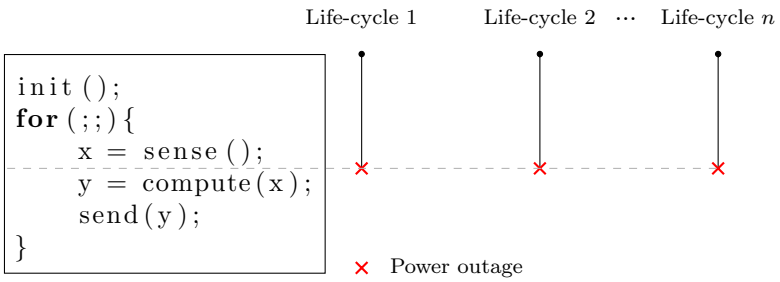


Figure 2.1: Program counter volatility.

2.6.2 P1 — Memory and CPU State Volatility

The most known issue in the area of transiently-powered systems is the persistence of the volatile memory contents, including main memory as well as CPU registers. Upon power outage, all volatile memories are discarded and thus, their contents can no longer be trusted. Some mechanism must be able to repopulate memory and CPU back to a valid, consistent state, so that the application can resume from that point.

The purpose of the memory is to store data, so saving and restoring its state may be as simple as copying its contents to and from non-volatile memory. The copy may be performed using a Direct Memory Access (DMA) component, present in every micro-controller, that enables fast copies, even when the CPU is turned off to save energy.

The CPU registers are not directly addressable so handling their persistence is slightly more challenging. As far as general purpose registers are concerned, using `push` and `pop` instructions allows the CPU to store and load its registers to and from the stack. Either the stack is located in volatile memory and can benefit from the persistence mechanisms of the rest of the volatile memory. Or the stack is located in non-volatile memory, which makes it easier to make persistent, but creates a need for careful handling in order to preserve memory consistency, as discussed in Section 2.6.5. The CPU also contains a handful of special registers, such as the program counter, the stack pointer register and the status register.

Making the program counter persistent enables the application to make forward progress, as shown in Figure 2.1. Without persistent program counter, the application would always reboot from the very beginning and the energy storage being small, only the first few instructions could ever be executed. Forward progress can be achieved only if the program counter is made persistent, however it is not sufficient. As aforementioned, in order to restore the application in a consistent state with regard to the program counter, the memory contents *must* be retrieved from the *exact* point in the application to which corresponds the program counter. In Figure 2.2, the application stops after setting a value to `x`. On reboot, without CPU registers and main memory made persistent, the application tries to use the value of `x` for some computation, but `x` was not re-initialized prior to the computation.

2.6.3 P2 — Handling Peripherals

Peripherals are nowadays still mostly volatile. On boot, they acquire a so-called reset-state, that is a constant state depicted in the device data-sheet.

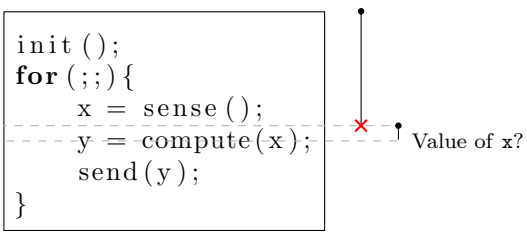


Figure 2.2: Volatility of CPU registers and memory, assuming program counter volatility being solved. The variable `x` could be stored either in a CPU register or on the stack, the issue is yet the same.

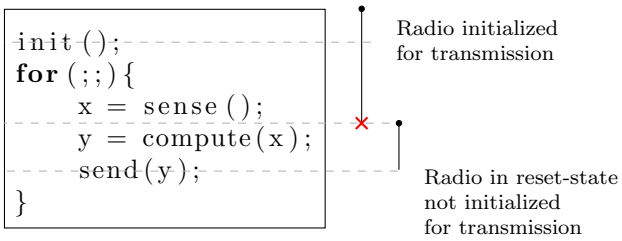


Figure 2.3: Peripheral state volatility, assuming that P1 is sorted out.

This issue was, for a long time, left aside by many systems for transiently-powered systems. However, it gained in popularity, starting from the moment researchers understood that peripherals were crucial to transiently-powered systems. Indeed, if such platforms designed to run under harsh run-time environments were only to perform computations, they would be of no use since any computer or data-center could perform the same computations, with better software and system support, and with sufficient computation capability to get faster results. It makes no sense to design transiently-powered calculators. On the contrary, a valid purpose for this kind of platforms is sensing a physical phenomenon using a sensor, maybe processing the data (*e.g.*, aggregation or filtering) and sending the usable value to a remote, continuously-powered device. Any scenario that would be suitable for transiently-powered systems involves peripherals and thus, peripherals cannot be set aside of considerations. Persistence of peripherals include the persistence of the state of each peripheral (P2.1), as well as the persistence of interrupts and data that arise from interrupts (P2.2).

P2.1 — State Volatility of Peripherals

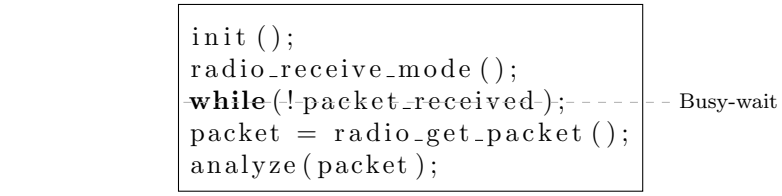
Peripherals, like any other volatile component, lose their state upon power outage. On reboot, they are in reset-state which is not necessarily consistent to their state when the program counter was saved. Figure 2.3 illustrates this issue. The `init` function initializes the hardware, mainly the sensor used by `sense` and the radio used by `send`. During the first life-cycle, `sense` expects the sensor to be initialized, which is the case since it is called after `init`. However, power fails before `send` is called. Upon reboot, the CPU and the memory are assumed to be properly restored in accordance with the program counter when the power outage occurred. The radio is now in reset-state though, which breaks the assumption of the application that the `send` function is called after the radio has been properly initialized. Note that the assumption would not be broken without the power outage, since the assumptions of C-like languages would guarantee that `send` is executed after `init`.

The simplest kind of peripherals are memory-mapped peripherals. They *might* be saved and restored by simply copying their control registers to and from non-volatile memory. However, even memory-mapped peripherals have some complexity. For instance, control registers designed as locks, require to be accessed first (unlocked) so that the other control registers become accessible. In addition, locks usually need a password value written to them in order to successfully lock or unlock them, but reading their memory-mapped value does not yield the password. An example of such locks is the clock register `CSCTL0` of the Texas Instruments' MSP430FR57 micro-controller family. [2]

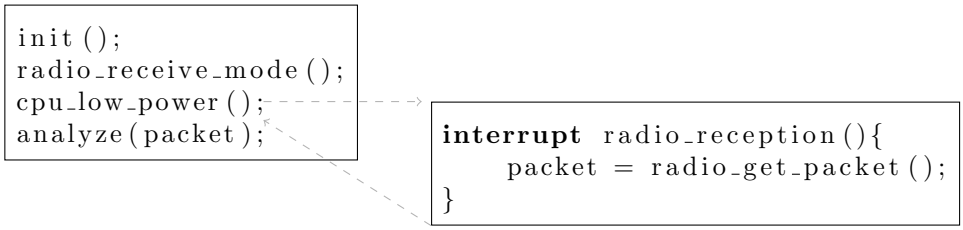
Moreover, memory-mapped peripherals often have write-only control registers. They thus cannot be read to be restored later. This is the case for ARM-based GPIO controllers (*e.g.*, Microchip's SAMS70 [3]) that have a write-only control register to set the GPIOs (*e.g.*, `PIO_SODR`) and another write-only control register to clear the GPIOs (*e.g.*, `PIO_CODR`).

Due to electronic reasons, some control registers might require that a minimum amount of time elapses before doing another operation. Time requirements are often due to the necessity of waiting for a signal to stabilize, an oscillator to settle, a capacitor to charge, *etc.* Voltage references for Analog-to-Digital Converters (ADCs) are examples of peripherals that use this kind of control registers. The reference module of Texas Instruments' MSP430FR5739 [4] requires 30 µs to settle, before being usable by any

[2] `https://www.ti.com/lit/ug/slau272d/slau272d.pdf`
[3] `http://ww1.microchip.com/downloads/en/DeviceDoc/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527D.pdf`
[4] `https://www.ti.com/lit/ds/symlink/msp430fr5739.pdf`



(a) Example of application without support for interrupts.



(b) The same application written with interrupts allowed.

Figure 2.4: Comparison between (a) interrupt-less model and (b) interrupt support.

dependent module, while the ADC itself requires at most 100 ns to settle before the application can issue sampling commands.

Other peripherals are not mapped in memory and must be accessed through a serial data bus; *e.g.*, SPI, I^2C. The data is written to or read from the peripheral by accessing the same address in memory several times in a row. Hence, dumping the contents of such peripherals is not as easy as memory-mapped peripherals. Furthermore, since the data bus is serial, reading the whole peripheral state would take a long moment, which is not desirable when a power outage is imminent.

Peripheral persistence is simply one step towards correct execution for transiently-powered systems. But by itself, it does not solve much. Since the persistence of the CPU program counter *dictates* where to resume the application, all the other components to be saved and restored, including peripherals, must be kept consistent with the checkpoint. If, for some reason, the state of the peripherals changed between the last checkpoint and the actual power outage, the state of the peripherals in the checkpoint image *must* reflect their state when passing through the checkpoint, not the last state observed when the power actually failed. Such situations arise if the system lets the application run after a checkpoint with no guarantee that the next checkpoint is reachable.

P2.2 — Interrupt Handling

Interrupt handling is a key notion in embedded systems. It enables reactivity and asynchronous operations that either come from the application itself, or that come from environmental conditions. Interrupts are preponderant in programming for embedded systems, hence inherently part of the programming model. In addition, in low-power systems, it is a common practice to halt the CPU while the platform is waiting for some wake-up signal on an interrupt line. Figure 2.4 shows the same application without interrupt support and with interrupt support. Interrupts are twofold: (*i*) interrupt occurrence and (*ii*) interrupt-bound peripheral data.

Current volatile micro-controller designs encode the interrupt occurrence in the peripheral control registers. For a given peripheral, one or several control registers contain the nature of the interrupt that occurred, if any. These control registers are volatile and reset upon reboot, meaning that the platform does not naturally keep track of interrupt occurrence across power outages.

Also, some data are bound to interrupt occurrence. For instance, upon receiving a radio packet, the contents of the packet are interrupt-bound data. These data are stored inside the peripheral memory, that is most likely to be fully volatile. Hence, the interrupt-bound data are lost upon power outage.

Overall, it must be an application design choice to choose whether to make an interrupt persist. Indeed, in portions of applications that require timeliness constraints (*cf.* problem P3), keeping the interrupt occurrence and data after a power outage would not make sense if the interrupt-bound data were meant to be used in a short time-span. However, other portions of the application might not have

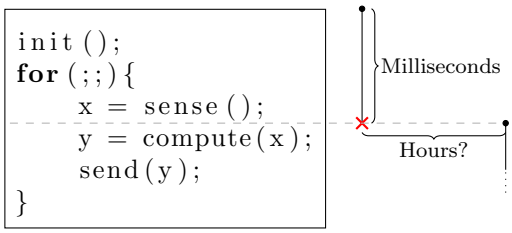


Figure 2.5: Illustration of the timeliness constraint, assuming that P1 and P2 are sorted out.

such timely requirements and thus would benefit from making interrupt-bound data persistent across power outages.

2.6.4 P3 — Timeliness and Atomicity

Intermittent power may lead to long periods of time when the platform is not powered. This is not supported by traditional programming languages such as C. Furthermore, power outages break assumptions of such languages: a sequence of instructions is not guaranteed to execute within a consistent time window. In a continuously-powered scenario, an assembly instruction would take a certain amount of clock cycles and the next instruction would be executed right after the current instruction completes (out-of-order execution and instruction pipe-lining are out of scope here). In a transiently-powered scenario, two instructions might be separated by many hours or days of power outage, as shown in Figure 2.5.

Depending on the application, it might not make sense to process sensed data or to send them after such a long pause. For instance, computing the average value of the output of a sensor makes sense only when the samples were taken during a time window where the physical phenomenon to be measured does not vary much. Consider an application that averages temperature measurements and sends the result over radio. If the sampling operation starts at the middle of the day, when the sun strongly shines, and resumes during the night, in that case the computed average has no meaning.

Other applications might however not need timeliness constraints, either depending on the nature of the application itself, or because the platform is so energy-constrained that a long time-consistent region would never fit in a single life-cycle.

Nonetheless, timeliness is not solely needed for specification reasons, but also for technical reasons. Operations on peripherals may take time, and it is often not desirable, in case of an interruption due to power outage, to attempt to resume exactly where the program stopped, but on the contrary to replay some of the former operations in order to make the peripheral operation work. A typical example of such peripheral operations is sending a radio packet. External radio chip-sets such as Texas Instruments' CC2500 communicate with the micro-controller through a serial bus (SPI in that case). Transmitting a packet requires to populate the radio FIFO and to send a strobe command. Every piece of data is exchanged through the serial bus and this takes time. If power fails during this operation, the platform cannot simply resume at the very same assembly instruction, even under the assumption that the radio state was properly restored. Instead, the application control-flow must be rolled back to the beginning of populating the radio FIFO. Another example of such peripheral operations is clock and system stabilization. When powering a peripheral on, or when making a peripheral change its state from one state to another, the data-sheet may indicate that the system must wait during a fixed duration until a piece of circuitry stabilizes before utilization. In case of power outage occurring during the wait phase, the control-flow must roll back to the beginning of the wait in order to ensure that the system waited at least during the right amount of time.

The key point to ensuring timeliness is offering atomic sections to the run-time environment, as well as exposing atomic section specification to the developer, through dedicated API functions or by providing a language better suited to transiently-powered systems.

2.6.5 P4 — Using Non-volatile Memory as Main Memory

Solving timeliness (P3) is crucial to transiently-powered systems for their application scenarios are based on peripherals and their correct behavior must be ensured by timeliness support. As seen in Section 2.6.4, timeliness support eventually brings roll-backs to application control-flow, in locations where there is no roll-back in a contiguously-powered scenario. In addition, some solutions to application state volatility (P1), especially compiler-based ones that statically insert checkpoints, might also induce roll-backs. Indeed, if the application opportunistically executes until the very end of the energy storage, the application will perform partial progress between the last checkpoint taken and the actual power outage. Then, on boot, the application resumes at the last checkpoint, introducing a roll-back in the control-flow.

Data retention across power outages is what makes non-volatile memory attractive in transiently-powered systems. However, in some situations, data retention requires additional care in order to keep the memory consistent. These situations arise from the roll-backs introduced by either timeliness support or partial progress between checkpoints.

All these situations can be gathered under a single concept: write-after-read data dependencies.

Write-After-Read Dependencies Hazards

Write-after-read dependencies are usually dangerous in concurrent execution contexts, for they require a particular order of instructions. A simple write-after-read dependency is `++i`, which purpose is to increment the value of `i`. In this example, a new value, dependent on the value of `i`, is written back to `i`.

In the context of transiently-powered systems, DINO [58] proposes to see intermittence as concurrent execution: the power outage makes the current task hang forever and a reboot creates a duplicate task that starts at the last checkpoint. Let us consider that a checkpoint was created right before `++i` and the platform did manage to complete `++i` before the energy storage depleted but without being able to reach the next checkpoint. If `i` is stored in volatile memory, partial progress is lost and the persistence mechanism must reinitialize its value so that it is consistent with the last checkpoint reached. If `i` is stored in non-volatile memory, it may be tempting to consider that no additional persistence mechanism is to use [59] and thus, to keep its value that includes the partial progress made between the last checkpoint and the power outage. Hence, on reboot, the `++i` instruction is executed again, with `i` being already incremented with respect to the expected value. As a result, a single `++i` instruction may increment `i` many times, instead of once, if power fails during this write-after-read dependency. The operations that bring such write-after-read dependencies are *non-idempotent*. When manipulating non-volatile memory, a correct execution in ensured if non-volatile variables (P4.1), non-volatile stack (P4.2) and non-volatile heap (P4.3) are protected against non-idempotent accesses.

Mathematically speaking, idempotence is the property of an operation that may be run multiple times and yield the same results. For instance, the absolute value, ceiling and floor functions are idempotent. In the programming world, idempotence has the same meaning, and it is important to note that idempotent functions *may* have side effects. However, an idempotent function that has side effects must ensure that the results of the side effects will always be the same after one or several repetitions. For example, a function accessing global variables is idempotent if and only if the values of the accessed global variables are the same after one or several executions of the function. Non-idempotence is the contrary; *i.e.*, non-idempotent operations do not lead to the same state when performed repeatedly. Less abstractly, `i = 4` is idempotent while `++i` is not.

Note that this situation is not only specific to non-volatile memory. Anything that may have persistent consequences is candidate to such vulnerabilities. Besides non-volatile memory, actuators are examples of systems with persistent consequences [60]. If a radio packet was sent once during partial progress, it will be sent another time after reboot. If a motor did move during partial progress, it will move again after reboot. While Phoenix [60] considers such roll-backs in actuator handling a fault, DINO [58] emphasizes that, in general, it is not possible to "un-launch a rocket or un-toast bread". As far as radio packets are concerned, networking protocols usually support packet redundancy, so it is not an actual problem. Regarding high-power actuators such as motors or servomotors, it is not realistic to consider that transiently-powered systems can provide for their high power requirements. As a result, this is an issue in high-power embedded systems that are subject to peripheral failures, but it cannot be observed under transiently-powered system assumptions.

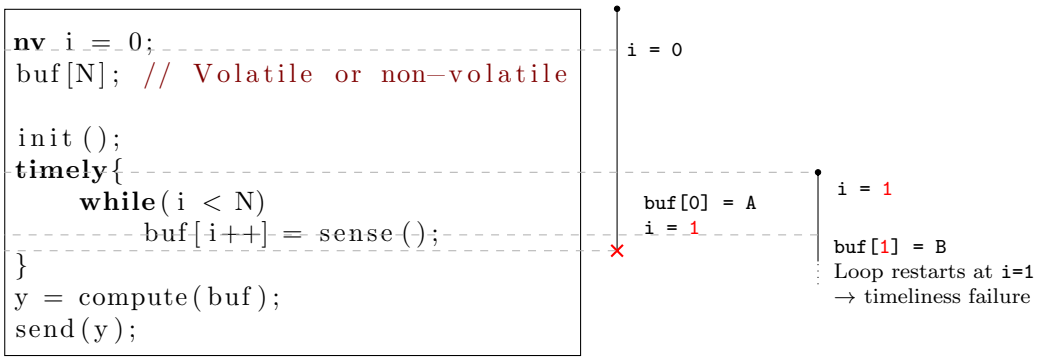


Figure 2.6: Non-idempotent access to non-volatile memory, assuming that P1, P2 and P3 are sorted out. The `timely` block is only conceptual and indicates a portion of code that must be made time-consistent. The `nv` decorator specifies that a variable is stored in non-volatile memory, while unspecified variables can be stored in any kind of memory.

P4.1 — Non-idempotent Accesses to Non-volatile Memory

A well-known issue introduced by the usage of non-volatile memory as main memory is brought by non-idempotent accesses [58, 61, 55, 62, 63]. Without a proper checkpointing mechanism, when power outage occurs in the middle of a write-after-read dependency to a variable stored in non-volatile memory, the platform may be exposed to an issue of non-idempotence. If the software is subject to roll-backs, and more specifically to the dominating read operation of the write-after-read dependency, the newly read value will be the one to be written afterwards. In that case, if the written value did change the variable state, then the newly read value is corrupted which hinders correct execution. Figure 2.6 illustrates this issue. The application requires a timeliness constraint, here materialized by a `timely` syntax block, for a buffer to be filled with sensed values within a short and consistent time-span. The `timely` syntax is not necessarily part of the actual syntax of the language, albeit similar to Chinchilla's `atomic` block [64], the `timely` syntax is purely conceptual for illustration purposes. At the beginning of the application, `i`, stored in non-volatile memory, is 0. The variable `buf` can be either stored in volatile or non-volatile memory, as long as there exists some mechanisms to make its contents persist across power outages. At this point, `buf` is not initialized. Then, the application starts the `timely` block, which means that a checkpoint stands here. The loop manages to run only one iteration before power fails, hence `buf[0]` has a valid value and `i == 1`. On reboot, the program counter, the volatile memory and CPU and the peripherals are assumed to be consistently restored to their state at the beginning of the `timely` block (P1, P2 and P3 solved). However, without rolling back non-volatile memory as well, in particular `i`, `i` is still equal to 1. The first iteration of the loop will hence write directly to `buf[1]` instead of `buf[0]`, meaning that either `buf[0]` is left uninitialized (if `buf` is stored in volatile memory) or with the old value of the former life-cycle (if `buf` is stored in non-volatile memory). Regardless of the placement of `buf`, the behavior is incorrect with respect to the expectations of the application developer in a continuously-powered scenario.

Peripheral Reads Introduce Idempotence Issues IBIS [63] further investigates non-idempotence by reasoning at peripheral level. Whenever a write to non-volatile variables is determined by a condition that depends on a value read from a peripheral, there might be an idempotence issue. After a power outage, if the control-flow is meant to roll back to before reading the peripheral, a second execution of the read operation might yield a new, different value with respect to the one that was read before the power outage. In that case, from the application's perspective, the control-flow successively took several branches while it would never happen in a continuously-powered scenario. If the concerned branches update non-volatile variables, then after the new branch has been followed, the non-volatile memory contains the stigmata of all the branches that were taken, hence leading to an inconsistent memory state.

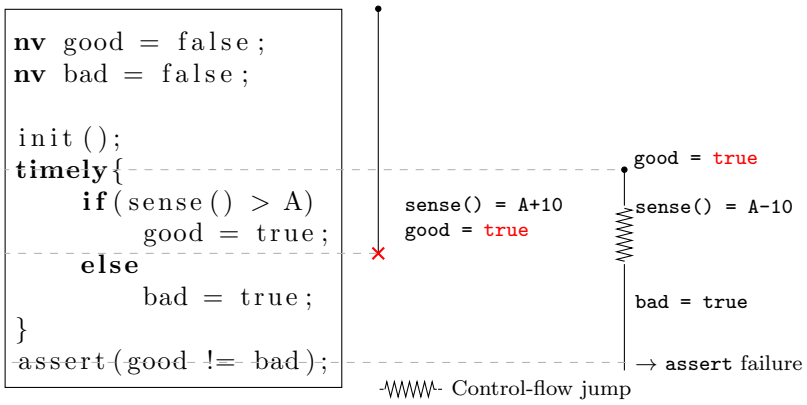


Figure 2.7: Non-idempotent access to non-volatile memory due to the influence of peripherals on the control-flow, assuming that P1, P2 and P3 are sorted out. The `timely` block indicates a portion of code that must be made time-consistent.

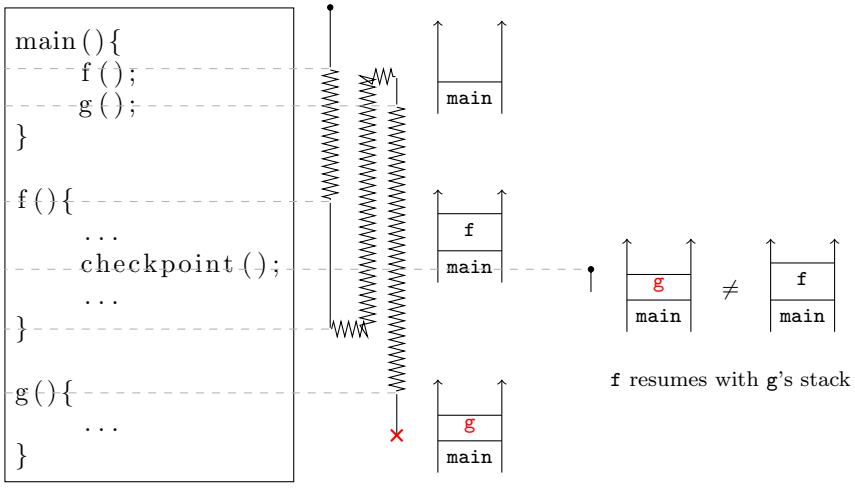


Figure 2.8: Non-idempotent access to non-volatile stack, assuming that P1 is sorted out.

Figure 2.7 shows an example application that, in a block subject to timeliness constraint, senses a sensor and updates either the non-volatile `good` variable or the non-volatile `bad` variable. Executing this application under continuous power would always lead to a state where either `good` or `bad` is `true`, but both of them cannot be `true` nor `false` together at the end, hence the `assert` condition. During the execution under intermittent power, the first life-cycle reaches the point where `good` is set to `true` after `sense()` yielded a value greater than the arbitrary value `A`. Then, the power fails and the application resumes at the beginning of the `timely` block, again assuming that P1, P2 and P3 are solved. Since `good` is stored in non-volatile memory and its value was not rolled back, `good` keeps its value, that is `true`. Now, `sense()` yields a value lower than `A` and the application control-flow takes the `else` statement, resulting in `bad` being updated to `true`. The control-flow eventually reaches the end of the `timely` block and now, both `good` and `bad` are `true` even though it never should happen.

P4.2 — Non-volatile Stack

Problems can occur when the stack is fully non-volatile [62]. This issue is very similar to non-idempotent accesses to variables (P4.1). The only difference being that, unlike P4.1, the accesses are not explicitly written by the developer. Indeed, programming languages such as C rely on Application Binary Interfaces (ABI) and their matching compilers automatically insert `push` and `pop` instructions at the beginning and at the end of the functions, as well as when registers spill. As a result, unprotected non-volatile stack frames are vulnerable to non-idempotence issues and thus, to stack corruption. Figure 2.8 shows three

functions with their own stack frames: `main`, `f` and `g`. During the first life-cycle, the application starts at the `main` function, then executes `f` which creates a checkpoint, returns to `main` and starts executing `g`. Power fails during the execution of `g` and the last checkpoint was inside `f`. Since the stack is non-volatile, it keeps its last contents, that is the stack frame of `g` on top of the stack frame of `main`. On boot, the control-flow is rolled back to `f`'s checkpoint but the stack is not. This leads to `f` being unexpectedly executed with the stack frame of `g`, hence the execution is incorrect.

P4.3 — Non-volatile Heap

Roll-backs might lead to the usage of variables that were freed between the last checkpoint and the power outage, as shown in Figure 2.9. In a contiguously-powered scenario, the variable `x` is valid when its contents are accessed. If power fails after `x` being freed, then on reboot the application resumes from the last checkpoint. The heap is non-volatile and was not rolled back to a consistent state with regard to the last checkpoint, hence `x` is still freed and `x[0] = 4` will not run correctly.

Since the concept of heap is not popular in very constrained embedded systems, non-idempotence in non-volatile heap management is not an actual issue for today's transiently-powered systems. The only work that cares about this concern to this day is ScEpTIC [62]. However, it might become an upcoming concern as future transiently-powered systems will embed more memory, have more computation capabilities and will be likely to allow the use of fully non-volatile heap.

Problems P1, P2, P3 and P4 encompass all sources of bugs, identified to this day, that may emerge from the intermittent nature of power supply in transiently-powered systems. Knowing the threats of intermittent supply, it is now natural to formulate the following global problematic: how to address these problems to build correct applications that can spread across power outages? This thesis specifically orients its point of view towards operating systems. Thus, the problematic may be further tailored as: what services should an operating system for transiently-powered systems provide in order to guarantee the correct execution of applications across power outages? To spread application progress across reboots, the system needs to make the CPU and volatile memory persistent and consistent with one another (P1). Peripherals must also be kept consistent with the CPU program counter (P2). The system needs to provide atomic regions for peripheral purposes and for specific time-related application requirements (P3). The usage of non-volatile memory must take into account consistency issues caused by a clash between non-idempotent regions and the run-time environment forcing reboots (P4).

2.7 Hardware Architecture for Intermittent Computing

2.7.1 Some Choices are not Deliberate

Design space for embedded platforms can be wide. Any micro-controller, any component is to be picked with caution in response to the purpose the platform or the application was made for.

With the arrival of volatility-related problems in intermittent computing, design space should be even wider with a new degree of freedom that is how to integrate non-volatile memory to the whole design. While some technologies become mature, such as FRAM and PCM, other ones are still emerging and not available to all researchers. Non-volatile processors, or micro-controllers with non-volatile flip-flops,

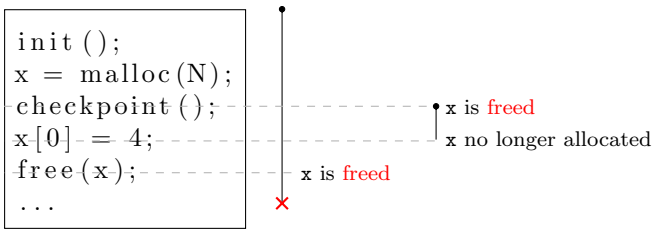


Figure 2.9: Non-idempotent access to non-volatile heap.

are not yet commercially available. Their usage might help in solving the volatility issues, while it would require interesting mechanisms to avoid non-idempotence issues (P4).

The main hardware architectures that emerged from state-of-the-art approaches are gathered hereafter. Architectures A1 and A2 depict available designs, while architectures A3 and A4 leverage interesting dedicated hardware solutions that are less accessible.

2.7.2 A1/A3 — Volatile Processor, SRAM and NVRAM

The oldest and most used hardware architecture for transiently-powered systems consists of a volatile micro-controller, SRAM and NVRAM. This architecture, named A1 here, is supported by all Texas Instruments' MSP430FR family micro-controllers. They are specifically designed so that NVRAM is incorporated into the micro-controller chip, hence achieving small form factor and good energy performance. Another way to build such a platform would be to use a micro-controller that exposes its address and data buses on external pins, and wiring those pins to an external NVRAM, as discussed in Section 4.7. It has the advantage of using any interesting micro-controller architecture, even from manufacturers that do not embed NVRAM in their designs, for intermittent computing scenarios. However, it increases form factor and is less energy-efficient.

Volatile and non-volatile RAMs present differences in energy consumption and access times. In general, volatile RAMs are both more energy-efficient and faster than non-volatile ones. Some works use RAM as working memory for partial progress and only commit progress to NVRAM when reaching a checkpoint [56, 55, 57, 65, 64, 66, 67].

Older works use Flash technology instead of NVRAM [8, 21], mainly because of the immaturity of NVRAM at that time. Since NVRAMs have no access constraints like Flash does, it is easy to adopt NVRAM instead of Flash. The only aspect that could hinder such a change is that NVRAM chips usually do not exceed a few megabytes, however it is sufficient for transiently-powered systems.

Some works propose the usage of dedicated hardware on top of architecture A1, either as new electronic features [15] or as new logic to be implemented in future processors [68]. The resulting architecture is referred to as A3.

2.7.3 A2/A4 — Volatile Processor and NVRAM

Another architecture, named A2, solely consists of a volatile micro-controller and NVRAM. Such an architecture can be obtained by using the same platforms as A1 and by simply ignoring the on-chip SRAM.

Other works also propose dedicated hardware [69, 70, 71] on top of architecture A2, the resulting architecture is called A4.

Architectures A2 and A4 expose fewer degrees of freedom regarding memory placement: every piece of data outside the micro-controller is non-volatile. By essence, these architectures expose all applications to P4.

2.7.4 A5 — Non-volatile Processor and NVRAM

Non-volatile processors are processors which parts, up to the entire unit, are made of non-volatile flip-flops. The basic non-volatile CMOS flip-flop is twofold: a so-called master latch is the traditional volatile flip-flop and a slave latch contains the non-volatile logic [72]. The slave latch is controlled using logic switches in order to either save the value from the master latch into the non-volatile cell or reversely load the value from the non-volatile cell into the master latch. This architecture is a consequence of non-volatile cells having limited endurance and write performance. The non-volatile processor loads all latches on boot and saves them all upon imminent power outage, hence current peaks occur during those phases.

Self-Write-Termination non-volatile flip-flops consist of a volatile latch and, upon completion of any write to the latch, either set or reset, the new data are copied back to non-volatile memory [73, 74, 75, 76]. Specific modes must, in addition, be implemented to manage the restoration of the latches upon reboot. Self-Write-Termination enables the flip-flop to reduce time wastes arising from the heterogeneity of setting or resetting a flip-flop in ReRAM technologies for instance. At the end of the day, the overall energy

of storing a bit into that kind of flip-flop is orders of magnitude smaller than simply using a traditional non-volatile flip-flop.

While these solutions target the micro-controller aspect of P1, they would also extend P4 to the micro-controller itself: roll-backs would build an inconsistent micro-controller state, where general purpose registers do not match the rolled-back program counter. An undo-log or a double-buffer would be required, as the solutions depicted in Section 3.4.5 tend to propose, for the micro-controller as well. This would limit the incentive of using non-volatile processors, since they would still need to be backed up. In addition, non-volatile processors are to processors what non-volatile RAM is to volatile RAM: slower and more energy-consuming. These conditions altogether make the usage of non-volatile processors unfit for transiently-powered systems for now. Hence, they are not yet used in the main literature for transiently-powered systems.

Chapter 3

Operating Systems for Transiently-Powered Systems

Operating systems may refer to a large spectrum of disciplines and this work focuses on one of them, fundamental to transiently-powered systems: checkpointing. In addition, intermittent computing is a very constraining environment and traditional, fully-featured operating systems are not likely to be usable in this context. An operating system for transiently-powered systems thus resembles a library and a runtime environment for checkpointing purposes, with maybe additional features that mimic traditional operating systems.

3.1 Application Scenarios

In order to grasp the interest and what is at stake when it comes to transiently-powered systems, inspecting application scenarios enables a better understanding of such systems. While many former works solely focused computational applications such as cryptographic applications, it is important to identify better-suited applications for transiently-powered systems. Indeed, peripherals and sensors in particular are central to this kind of systems.

3.1.1 The Dilemma of Benchmarks for Transiently-Powered Systems

As stated in Section 2.6.3, the applications for transiently-powered systems must use peripherals in order to be considered realistic. While they enable to show the real benefits of systems based on checkpointing, they suffer from unescapable portability issues. Benchmarks hence do no longer only consist of software code bases. The software embeds driver code that assumes that a specific platform and a specific sensor are used in the hardware design. As a result, comparing the efficiency of different solutions for transiently-powered systems is difficult as everyone has to re-implement the other solutions on their platform.

Micro-controllers' instruction sets and Application Binary Interfaces differ from one manufacturer to another. Some mechanisms might be implemented in a single or a few instructions at most on a given platform, whereas it would take many instructions on another platform which was not particularly designed for a specific usage. For instance, as seen in Chapter 6, returning from MPU fault on an ARM-Cortex-based micro-controller is straightforward while it would take many instructions on MSP430 including a light software instruction decoder to determine how many words to subtract from the program counter, to replay the faulting instruction instead of resuming at the next one.

In addition, an application may be designed for some off-the-shelf experimental platform. This presents the advantages of being available to other researchers, though financial cost might be an issue in some cases. On the other hand, off-the-shelf platforms are generic and thus, not necessarily designed for transiently-powered systems. As a result, platform energy consumption might not be optimized, or the choice of on-board peripherals might not fit the extreme requirements of intermittent power.

Or, an application is designed for a dedicated platform. In that case, the energy efficiency of the whole platform might be optimal for it was designed for transiently-powered systems on purpose. The

drawback of building dedicated platforms is that it becomes less easy for the other researchers to get that platform or to build it.

The following section shows ideas of applications that take advantage of peripherals, gathered from several works.

3.1.2 Sense, Process and Possibly Send

From the literature, the only kind of peripheral-based applications for transiently-powered systems that emerged consists in sampling data from sensors, processing the data and sending the results over some communication channel or showing the results in some way.

Compressive sensing [21, 59] This application accumulates 64 accelerometer measurements and computes the minimum, maximum, mean and standard deviation of the samples. Raw measurements are stored in volatile memory and the computed statistics in non-volatile memory.

Environmental monitoring [67] Once per minute, this application senses temperature and humidity and computes the average over the last windowed measurements. Then, it sends the result over radio.

Cold-chain equipment monitoring [57, 77, 64, 65, 55, 56] This application senses samples from temperature sensor and compress them using Lempel-Ziv-Welch compression.

Correlated sensing and report [15] This application samples a magnetometer and a proximity sensor. From these pieces of information, it deduces the distance to the magnetic flux source. Then it sends the result over Bluetooth-Low-Energy.

Microphone [67] This application samples a microphone and detects vocal commands whenever they are issued by a human operator. The vocal command indicates what peripheral to sense next. Upon command detection, it senses the corresponding peripheral. Finally, the application sends the sensed data.

Activity recognition [67, 57, 77, 64, 65, 55, 56, 58] This application senses samples from a 3-axis accelerometer. Then, it transforms the samples into features by computing, among others, the mean and the standard deviation. Using a classification model, it then determines whether the device is stationary or shaking.

Gesture-activated remote control [15] This application samples a photoresistor. If an object is detected, it enables a gesture sensor to perform gesture recognition. If the gesture is recognized, it sends data over Bluetooth-Low-Energy.

Temperature alarm [15] This application samples temperature in a manner that preserves timeliness among all temperature samples. If the temperature is outside a certain range, send data over Bluetooth-Low-Energy.

Microphone and accelerometer with interrupts [78] This application is a mix of Activity recognition application and Microphone application, with interrupt support in addition. The software defines two tasks: one manages the microphone, and the other one manages the accelerometer. By default, the system is sleeping and wakes up upon microphone activity. The application then performs command recognition on microphone data. If a command was recognized, it activates the accelerometer task, which periodically senses and classifies the accelerometer data. Whenever the energy is high enough, it generates an interrupt which wakes up a task that sends accelerometer data over infrared.

3.1.3 Benchmarks and Testbenches for Transiently-Powered Systems

Traditional benchmark suites for contiguously-powered systems only comprise code. They are useful to elaborate the comparison between different solutions by setting up a baseline. Ideally, those benchmark suites are portable so that anyone can compile them on their own platform and reproduce the comparisons. Benchmarks for platforms equipped with multi-processors aside, a few benchmarks emerged, targeting embedded platforms: MiBench [79], BEEBS [80] and Embench [1] are examples of such benchmarks. However, to this day, none of them actually use peripherals. Inputs are either hard-coded in data arrays, or obtained through Unix-like `read` and `fscanf` primitives. In addition to using dynamic allocation (*i.e.*, heap), the whole picture tends to show that these benchmarks, although targeting embedded platforms, are designed for platforms capable of running a traditional operating system such as an embedded Linux distribution.

Transiently-powered systems need benchmarks that (*i*) make use of peripherals (*ii*) also come with a physical run-time environment. This physical environment must provide reproducible power outages, thus somehow emulate the behavior of the energy harvester.

On the hazards of pre-recorded load power traces Ekho [81] warns against pre-recorded load power traces. Power traces are hardware dependent. Simply changing the nature or the capacitance of the energy storage has effects on the system's response to the harvested physcial phenomenon. In a given trace, if the energy storage saturates, does it really mean that the physical phenomenon ceased to deliver energy? The energy storage may saturate, and running that power trace with a bigger energy storage would lead to wrong results, since the bigger energy storage would have continued to accumulate energy whereas the smaller one could not. The voltage trace from the transducer's output, either in open-circuit or with a load, is not meaningful neither since the performance of the transducer depends not only on the environmental conditions, but also on the current load attached to it, that is, the energy harvester and the rest of the platform. KARMA [67] uses traces from Mementos [21] and direct outputs from a solar panel. Eon [82] leverages harvested energy traces instead of voltage traces, but again energy storage saturation might prevent energy traces from being transposed from a platform to another.

It would be better, although more difficult, to provide a trace of the physical phenomenon itself rather than the resulting load power trace. For instance, if the energy source is light, a practical trace would be the timely evolution of the ambient luminance. However, in order to be exploitable, such traces need a model of the energy harvester, notably a model of the relationship between the physical phenomenon and the accumulated energy, including energy efficiency ratio.

Ekho [81] proposes to complement the traces information in order to make them transposable to other platforms, leveraged by works such as InK [78]. Traces include I-V curves, *i.e.*, the relationship between the current and the output voltage of the transducer. Ekho is an emulator that records I-V curves in order to emulate them, by dynamically adapting the emulated harvesting current according to the measured energy storage voltage.

3.2 Software Architecture

Intermittent computing is a totally different paradigm in comparison to traditional computation under contiguous power. It is likely that solutions to the identified problems have an impact on the programming model, from the application developer's perspective.

3.2.1 Baseline: Bare-metal Applications

For comparison purposes, let us consider bare-metal applications as the baseline, written in C language.

Memory Organization They may arrange memory to fit their needs, with a dedicated linker script, for instance.

[1] `https://embench.org/`

Peripheral Accesses In bare-metal scenarios, peripheral control registers may be directly accessed. However, it is very common and a good practice, even in bare-metal applications, to encapsulate such accesses onto helpers that can be considered close to Hardware Abstraction Layer and drivers. [2][3] Those drivers often come with lightweight data structures that keep track of the state of the underlying peripheral. Such data structures are referred to as *device contexts*. The device context can either mirror the control registers, or represent the state of the peripheral at a higher level, *e.g.*, the frequency of a clock that would be used to set the value of several control registers of the clock peripheral.

Interrupts Bare-metal applications often leverage interrupts for asynchronous operations. Interrupt handlers may perform any operation, however it is common that the interrupt handlers run as shortly as possible in order to keep a good overall system performance.

Bare-metal Example Figure 3.1 depicts an average scenario of bare-metal application. The software is carefully structured: the peripherals have their own compilation units and their features are encapsulated into simple drivers, so that the application logic becomes a separate matter. The application logic is interrupt-based. Indeed, the CPU is halted most of the time, thanks to the `halt_cpu` routine. Every 100 milliseconds, the CPU is woken up to perform a single measurement and to store it in a buffer. A radio module is configured in receive mode, with a duty cycle of 10%, meaning that the radio is in receive mode 10% of the time and idle the remaining 90% of the time. This feature does not need to be manually managed by the application since it is a common feature of radio chipsets, such as Texas Instruments' CC2500. [4] Upon receiving a radio packet, the application must send its current buffer contents.

3.2.2 Qualitative Metrics of Programming Effort

In order to compare solutions, one can be interested in seeing their impact on programming ease. More precisely, discrepancies with respect to the bare-metal scenario depicted above can be informally evaluated. Since intermittent computing influences both application logic and driver programming, programming effort evaluation is twofold: *impact on application programming* that encompasses language syntax as well as application logic, and *impact on driver programming* that focuses on the way to write drivers.

Note that a heavy impact does not necessarily mean that a solution is bad, nor does it mean that getting into the proposed programming model will be long and difficult. Indeed, intermittent computing breaks some assumptions of the C language and thus, the C language is far from being perfectly suited to intermittent environments. Languages purposely designed for intermittent computing will, by definition, suit the requirements of intermittence, but consequently will be far from a C language baseline.

3.2.3 M — Memory Organization

The choice of the architecture is not sufficient, by itself, to enable application execution across power outages. A memory policy must be designed to support the application. It also identifies the issues that will face the application.

M1a — Compute in RAM, Commit to NVRAM The memory organization M1a consists in working in volatile RAM as contiguously-powered systems do, and in committing progress into NVRAM at some point. This choice may come from NVRAMs being slower to access and more energy consuming than volatile RAM. While the gap tends to shrink with technology improvements, this discrepancy still stands. Since the NVRAM is only used as a storage memory, solutions that adopt M1a are exempt from P4 issues. If some of the system variables, *i.e.*, not the application variables, are stored in NVRAM, they are subject to P4 issues though.

[2] `https://www.gitbook.com/book/arobenko/bare_metal_cpp`
[3] `http://umanovskis.se/files/arm-baremetal-ebook.pdf`
[4] `https://www.ti.com/lit/ds/symlink/cc2500.pdf`

```c
#include <temperature.h>
#include <radio.h>
#include <timer.h>
#include <buffer.h>

static buffer_t *buffer;

int main(void) {
    /* Initialize peripherals */
    temperature_init();

    radio_init();
    radio_set_mode(RECEIVE_DUTY_CYCLE_10_PERCENT);

    /* Temperature sense timer */
    timer_start(100_MS);

    for(;;)
        halt_cpu();
    return 0;
}

interrupt timer_interrupt(void) {
    buffer_add(buffer, temperature_sense());
    timer_start(100_MS);
}

interrupt radio_receive(void) {
    if(buffer_size(buffer)) {
        radio_set_mode(TRANSMISSION);
        radio_send(buffer_contents(buffer));
        radio_set_mode(RECEIVE_DUTY_CYCLE_10_PERCENT);
        buffer_clear(buffer);
    }
}
```

Figure 3.1: Example of bare-metal application that aims at being energy-efficient: the radio is setup with a low duty-cycle and the CPU is halted most of the time. The peripherals wake the CPU up when it is time to either sense the temperature sensor or send the data over radio. The `buffer_t` structure materializes a generic circular buffer type.

M1b — Compute in RAM, Commit to NVRAM and Some Variables in NVRAM The memory organization M1b is similar to M1a. The difference is that M1b enables the application to store its variable directly into NVRAM. This kind of memory organization is only used by task-based solutions [55, 65, 15, 78, 57]. Task-shared variables are stored in non-volatile memory while task-local variables are stored in volatile memory. Task-shared variables are privatized: tasks operate on a local copy and then commit the final value to the original location in non-volatile memory when they complete.

M2 — Use Only NVRAM Under the architectures A2 and A4, relying on the sole use of NVRAM, the memory has no choice but being non-volatile. It is called M2. As a result, apart from the micro-controller that stays volatile (*i.e.*, registers), stack and all variables are mapped into NVRAM. Hence, this design choice only reduces the impact of P1 but does not erase it entirely. Also, adopting M2 automatically brings up P4 issues.

3.2.4 E — Execution Model

Besides the architecture and the memory organization, systems need an execution model. This impacts programming model as well.

E1 — Mono-task Model The most straightforward execution model is the mono-task model, called E1. It directly derives from the bare-metal approach and thus, does not require many changes with respect to the bare-metal baseline.

E2 — Task-based Model Another popular approach is task-based. In transiently-powered systems, task-based approaches follow the same scheme [56, 55, 65, 15, 78, 57]. Tasks are run atomically as a way to address P3. This execution model comes with programming languages, designed for task-based systems, with carefully chosen semantics. Since atomicity is at the essence of such approaches, the tasks are guaranteed to be run atomically. However, atomicity raises constraints on the energy storage, as it has to provide enough energy for the longest task to execute. Using E2, the application developer must handcraft every task *and* hardware. Even computing sections, that usually do not require atomicity, might be spread onto several tasks in order to execute them under intermittent power. Since the application developer is in charge of defining task boundaries, and task boundaries being checkpoints, as a result the application developer is in charge of manually placing the checkpoints. Overall, task-based models fit well the requirements deriving from P3, but need heavy changes with respect to the bare-metal baseline.

3.3 Designing Platforms for Intermittent Computing

3.3.1 Platforms

WISP [8] is probably the most popular platform designed for transiently-powered systems. It specifically targets RFID systems and, as a consequence, harvests radio energy. WISP chose to supply the platform with a 1.8 V voltage regulator, which provides stable supply voltage for the micro-controller and the peripherals. Although the design is getting older now, originally embedding an MSP430F1232 micro-controller with Flash memory, the design may be easily adapted to use a more up-to-date NVRAM-based micro-controller. Moo [83] proposes an upgrade of the WISP platform that includes a new micro-controller, an external EEPROM and tools for measuring harvested current. The micro-controller is replaced with an MSP430F2618, that offers more memory and GPIOs, yet still uses Flash technology.

Capybara [15] is dedicated to dynamically adapting the energy storage to the energy needs of the application. The application is cut down into tasks by the developer and every task is annotated with energy requirements. Capybara comprises several capacitative units and dynamically switches them individually in or out so that the overall capacitance of the energy storage becomes adequate for the task needs. Specifically, Capybara allows the application developer to choose whether an atomic task is reactive with low capacitance or energy-consuming with high capacitance. The duration necessary to fill up the energy storage, as well as the powered-on duration, directly depend on the capacitance. Hence, it

Table 3.1: Comparison between existing systems for transiently-powered systems, with respect to their ability to solve the identified problems.

System	Archi.	Mem.	Task	P2.1	P2.2	P3	P4.1	P4.2	App. eff.	Drv. eff.
KARMA [67]	A1	M1a	E1	Y	Y	Y			+	++
Sytare [66]										+
RESTOP [84]					N	N			+	++
HarvOS [85]				N	Y	N			0	0
MPatch [28]										
Mementos [21]					N	Y			+	0
Hibernus [22, 86]						N			0	0
eM-map [87]		M1b	E1	N	Y	Y	N		+	0
Chinchilla [64]					N	Y	Y	N	+	++
Coati [57]			E2	N	Y	Y	Y	N	+++	++
Alpaca [55]					N	Y	Y	N	+++	0
Chain [56]										
Coala [88]						N	Y		+++	0
Ratchet [61]	A2	M2	E1	N	N	Y	Y	Y	0	0
Samoyed [77]								N	0	+++
TICS [89]									++	0
Quickrecall [59]						N	N	N	++	+
CatNap [90]			E2	N	Y	Y	N		++	0
DINO [58]					N	Y	Y	N	+++	0
InK [78]									++	0
Capybara [15]	A3	M1a	E2	N	N	Y	Y	N	+++	++
Mayfly [65]										
Reli [68]			Indep.	N	N	N			0	0
Clank [69]	A4	M2	E2	N	N	N	Y	Y	0	0
TCCP [91]			Indep.	N	Y	N	Y	Y	0	0

is crucial to manage capacitance for reactivity purposes. Capybara also uses voltage regulators to supply the micro-controller and the peripherals.

3.4 Solutions Brought by Existing Approaches

3.4.1 Literature Overview

Many works try to solve the generic problem of spreading the execution of an application across power outages. They may consider all the problems depicted in Section 2.6, or only some of them. The main works are quickly described hereafter and their answers to the identified problems are further detailed in the next sections. A summary of these works, their properties and their responses to the identified problems is depicted in Table 3.1. The coming literature overview is sorted in an order that is close to the classification from Table 3.1, modulo precedence constraints in the explanations.

Mementos [21] Mementos relies on energy-aware static checkpointing. Checkpoint trigger locations are inserted at compile-time. Then, at run-time, the platform senses the remaining energy by performing an ADC measurement of the energy storage and tests it against a pre-defined threshold. Mementos also provides hints of atomicity, by disabling checkpointing in the code sections that require atomicity.

Hibernus [22] / Hibernus++ [86] Hibernus performs checkpoints when the energy storage voltage drops below a certain threshold. The save voltage threshold, as well as the booting voltage threshold, may be dynamically adapted. If the checkpointing did fail during the last power outage, the save voltage threshold is increased so that the save operation is given more energy to complete. This management of voltage thresholds may enable the utilization of smaller capacitors as energy storage.

KARMA [67] KARMA focuses on the support for asynchronous peripheral operations for transiently-powered systems. It adopts a peripheral model based on finite state machines, where asynchronous operations must be encoded as states. Asynchronous states must encompass all possible combinations, resulting in an exponential expansion of the state machines. The driver developer is in charge of maintaining these state machines. A queue of operations complements the drivers in order to restore the peripherals to a consistent state with the last checkpoint. A peripheral operation is rolled back, upon reboot, if power failed while executing it, as Sytare [66] does, thanks to KARMA's queue of operations. KARMA operates at driver API level, as Sytare does.

RESTOP [84] RESTOP specifically targets P2. It consists in a middleware layer between application and drivers. RESTOP keeps a history of peripheral accesses. As a consequence, restoring the peripherals is done by running all entries of the peripheral access history. RESTOP requires the driver developer to manually classify every peripheral routine within a set of categories, in order to optimize the endlessly-growing peripheral access history.

HarvOS [85] HarvOS, despite statically inserting checkpoints, aims at reducing the amount of checkpoints and their durations, which are usually prominent in static approaches. It accomplishes so by postponing checkpoints to whenever it becomes urgent to checkpoint. Actually, it inserts many checkpoint locations but, similarly to Mementos [21], the system performs an energy measurement to decide at run-time if a save operation craves to be performed. HarvOS static analysis enables the estimation of the worst-case memory usage throughout the whole control-flow graph. HarvOS compiler can then deduce the amount of energy needed to checkpoint that worst-case data amount. It deduces the amount of time available for the application for each life-cycle, after subtracting the durations of checkpointing mechanisms. The control-flow graph is divided into sub-graphs that take at most half of the cycles available in the worst-case, ending with a checkpoint location. Function calls are replaced by their corresponding inlined blocks for the energy estimation, and a checkpoint is automatically placed upon returning from the functions. Interrupt handlers are also instrumented: checkpoints are inserted at the very beginning and upon returning from interrupt.

MPatch [28] MPatch is a checkpointing system specifically designed for a battery-free Game Boy. It initiates a checkpointing operation as soon as a power outage becomes imminent and optimizes the amount of data to transfer using harware-assisted detection of modified memory regions. Memory usage is further analyzed to decrease checkpoint size.

eM-map [87] eM-map targets an optimal memory placement to reduce both the energy overhead of the checkpointing techniques and their non-determinism due to power outages occurring at any time. It is a brute-force algorithm that measures the energy consumption of running each function with different variable placements, either in volatile RAM or non-volatile RAM, in order to keep the least energy consuming. Application functions are treated as separate and independent entities. Each function has its own `.bss` and `.data` sections. Checkpointing operations are performed upon returning from functions. However, a realistic application scenario may involve hundreds of functions and furthermore, they might communicate through global variables, hence mitigating the feasibility of such a brute-force methodology.

Chinchilla [64] Chinchilla statically inserts many checkpoints and dynamically selects the checkpoints to actually perform, as Mementos [21] does. The code is decomposed into blocks, in practice basic blocks, and checkpoints are placed on the blocks boundaries. Chinchilla assumes that blocks energy consumption has a low variance. It lets the application developer write special blocks, atomic blocks, that prevent Chinchilla from inserting checkpoints. A variable is stored in non-volatile memory only if its liveness crosses a static checkpoint. Non-volatile variables are instrumented with an undo-log which, on overflow, provokes a checkpoint. Peripherals are not handled as Chinchilla requires the driver developer to provide an initialization function to manually manage P2.

Chain [56] Chain is one of the first task-based approaches for transiently-powered systems that lay ground of further improvements. Tasks are explicitly written by the application developer. They run

atomically, in an all-or-nothing manner. On reboot, the application always restarts the current task, thus re-initializing memory to the beginning of the task prevents the system from inconsistencies. Tasks are chained together, which resembles a data-flow oriented approach: tasks consume data from other task and produce data for other tasks. Intra-task variables are stored in volatile RAM whereas inter-task variables are stored in non-volatile memory and must be accessed through Chain's primitives ensuring idempotence. However, Chain does not support peripherals.

Alpaca [55] Alpaca is based on Chain [56], thus inherits its task properties regarding atomicity and idempotence. Alpaca totally supersedes Chain within transiently-powered system context. Inter-task variables, also called task-shared variables, are analyzed against write-after-read dependencies. Then, only task-shared variables used in a write-after-read dependency are privatized, that is, duplicated as local copies to be committed upon task completion. On boot, peripherals are initialized by an initialization routine, but it only supports a single configuration. As a result, Alpaca provides no peripheral checkpointing mechanism.

Capybara [15] Capybara is a platform that enables dynamic energy storage configuration to suit application demands. It highlights the inevitable trade-off between system reactivity and atomicity. Indeed, atomicity often implies a bigger energy storage in order to run long atomic sections to completion without power outage. Bigger energy storage is slower to charge, which decreases system reactivity. Capybara aims at providing fair reactivity when the application needs reactivity and atomicity when the application needs atomicity. Its software is based on Alpaca [55]. In addition, every task is annotated with an energy mode, that at run-time corresponds to a given energy storage capacitance. Capybara is capable, at the beginning of every task, of selecting capacitive banks in order to increase or decrease the energy storage capacitance in accordance with the application specifications. The voltage thresholds of platform booting and shutdown stay constant, only capacitance changes. The application programmer must measure the energy requirements of the tasks themselves. Experimental results shows that the type of energy storage matters in terms of performance and liability. Capybara is then a smart breakthrough that takes energy demands variations into account.

Coati [57] Coati runs on the Capybara platform [15], using a task-based model similar to Alpaca's [55]. It complements Capybara's approach with the support for event-driven concurrency for transiently-powered systems. Coati adopts a split-phase interrupt model, where a top-half runs immediately and a bottom-half is deferred to the completion of the current task. As a result, tasks can be preempted only by top-halves. The key issue is that the concurrency brought by interrupts can make execution non-idempotent, while originally designed to be idempotent in an interrupt-free context. It thus uses a transactional memory for which the programmer must use specific `read` and `write` primitives. While Coati supports interrupt when not bothered by power outage, it is still unclear whether P2.1 is handled.

Coala [88] Coala is a multi-task system where each task is supposedly atomic, modulo task splitting whenever tasks fail to complete before power outage twice in a row. Energy demands are dynamically estimated based on recent history. That information is used to determine whether tasks can be coalesced or, on the contrary, split. Task coalescing reduces the amount of checkpoints during a single life-cycle and task splitting tries to cut down tasks that are so energy-consuming that they cannot complete within a single life-cycle. However, coalescing tasks introduces new write-after-read dependencies and Coala proposes a virtual memory manager to prevent these new dependencies from endangering memory consistency. The memory manager acts like a cache that commits to non-volatile memory upon page unloading, only if they were modified.

Samoyed [77] Samoyed provides an interesting way of handling peripherals at API-level. Timeliness motivates the peripheral API routines made atomic. The driver developer provides scale-down rules for every peripheral routine when necessary. At run-time, Samoyed tries to run the peripheral operation, and if a power outage occurs in the meantime, the scale-down rule is applied on the peripheral operation upon reboot in order to make the atomic section smaller and likelier to complete without power outage. In addition, non-volatile variables are managed using an undo-log in order to solve non-idempotence issues.

Ratchet [61] Ratchet aims at statically guarantee idempotence properties without imposing a programming model to the application developer. Its compiler cuts down the application into idempotent sections, linked by statically-inserted checkpoints. Code sections are separated at every write-after-read dependency, between the read and the write operations, by a checkpoint that thus creates two idempotent sections. Ratchet also considers `pop` instructions as write-after-read dependencies. The `pop` instructions are thus replaced by load instructions and the stack pointer is updated in a separate instruction, in order to place a checkpoint, which could not be achieved while keeping an all-in-one `pop` instruction. This checkpointing system is complemented by a timer, in case a large section of code does not have any write-after-read dependency.

TICS [89] TICS provides strong semantics for timeliness constraints in a new language. The new semantics resemble `try/catch` blocks, where the exceptions are time-related, based on application-defined annotations. A checkpoint is performed when a time-constrained block successfully completes. TICS further enables the use of pointers and recursion within transient computing context. This is achieved by versioning data. Write operations to non-volatile memory are instrumented. TICS uses a fixed-size non-volatile undo log, which is cleared upon successful checkpoint. An overflow in the undo log triggers a dynamic checkpoint.

Quickrecall [59] Quickrecall assumes no volatile SRAM (architecture A2). Checkpoint occurs when the energy storage voltage drops below a certain threshold. Only peripheral control registers are saved, which was shown imperfect even for simple peripherals in Section 2.6.3. The driver developer must yet specify an initialization routine.

CatNap [90] CatNap considers application tasks, event handlers and even considers energy recharge times as tasks. CatNap schedules event handlers as a traditional Real-Time Operating System would. The highest priority is given to event handlers, then recharge tasks and finally application tasks have the lowest priority. In addition, CatNap encourages the application developer to design degradable algorithms, *i.e.*, through iterative processes that refine a computation, so that the scheduler can decide to abandon the remaining iterations while keeping a fair, yet degraded, result to work with. This limits the extent of applications targeted by CatNap.

DINO [58] DINO targets atomicity and idempotence in order to guarantee data consistency. The application developer must manually place checkpoints in their code. It proposes smart models for intermittence: as concurrency with data races and as control-flow with data-flow induced by power outages.

InK [78] InK focuses on reactivity to events, taking advantage of interrupts rather than polling control registers. Ink leverages a task-based model, where tasks are atomic, idempotent like Alpaca [55] and prioritized furthermore. Preemption occurs on task boundaries, however tasks can be interrupted by interrupts, as in Coati [57]. Tasks belong to task threads and each task thread has a priority alongside an event queue. Interrupts may push events to the event queues. Scheduling is dynamic, the next task to be scheduled is always taken from the task thread of highest priority which has tasks ready to be executed.

Mayfly [65] Mayfly is a task-based system, similar to Alpaca [55]: tasks are atomic, task-local variables are stored in volatile memory and task results in non-volatile memory. The key improvement is that tasks are prioritized and data exchanged between tasks can be tagged with time-related information. The application developer may, using Mayfly's dedicated data-flow language, specify an expiration duration, the minimal delay between samples to limit the throughput and the amount of data to accumulate before running a given task. The time-related specifications leverage an external piece of hardware: an almost persistent timekeeper that is a low-power clock that can run when the rest of the platform is turned off, in order to keep track of time in case a power outage lasts for few minutes only. Mayfly's scheduler implements a greedy algorithm based on task priorities. Task priorities are automatically determined at compile-time, provided task graph analysis. The checkpoint image comprises time, scheduler

state through the task graph, and inter-task data. Peripherals must be manually handled by the driver developer.

Reli [68] Reli is an architectural modification of the CPU at micro-instruction level, that enables a contiguous checkpointing of the system, while not altering the programming model. Reli keeps track of register values and changed data memory locations in two separate stacks. When the execution of a CPU instruction modifies a register, the register and its value is pushed onto the first stack. When it modifies a memory location, the address and the value are pushed onto the second stack. By construction, saving the system is done in an incremental fashion. The restoration process consists in browsing the register modification stack and the memory modification stack in order to write the values consistent with the point of interruption. Depending on the nature of the application, Reli's approach may lead to overhead when the same address or same register is written to several times in a short time-span. Indeed, this would update the stack several times. This approach cannot solve P3 since checkpointing at every instruction does not enable atomic sections larger than one instruction.

Clank [69] Clank is the architecture-level counterpart of Ratchet [61]. The key difference is that the separation into idempotent sections is performed at run-time instead of compile-time. Clank leverages a dedicated architecture that spies on the address and data buses. It physically provides two buffers, namely for read-dominated and write-dominated variables. At run-time, a write-after-read dependency is detected when the CPU tries to write to a read-dominated variable. Clank updates its buffers on the fly, and performs checkpoint when either a write-after-read dependency is actually detected, or when the buffers overflow. Checkpoint operations flush the buffers, allowing further application progress. Clank's buffers also act like redo logs, that enable to roll-back the values of the non-volatile variables upon reboot. The run-time is also complemented by a watchdog timer that, similarly to Ratchet's, breaks large sections without write-after-read dependency into smaller sections so that roll-backs do discard only little progress every time.

TCCP [91] TCCP uses dedicated hardware to dynamically place checkpoints, while being transparent from the application developer's perspective. It leverages a processor where every register is mirrored by a non-volatile counterpart. The memory is entirely non-volatile, however the processor has a volatile cache to reduce the energy overhead of using non-volatile memory as main memory. The volatile cache is committed to non-volatile memory at checkpoint time and upon cache overflow.

Pheonix [60] Phoenix is a peripheral recovery system. It does not target transiently-powered systems but rather more power-consuming platforms such as autonomous vehicles and robots. The failures Phoenix is resilient to are not power outages, however they include communication failures, timeouts and spurious interrupts. Phoenix assumes that the system may identify a failure any time after the failure actually occurred, but not necessarily right after. Driver routines are atomic and drivers must provide a recover function. The difference with other state-of-the-art checkpointing systems that solve P3 is that Phoenix considers that some peripheral operations *must not* be replayed and thus must be skipped. Outputs to a console or actuator operations are typical examples of such peripheral operations that must not be replayed according to Phoenix. Phoenix maintains a peripheral operation log. A log entry is composed of the arguments, the return value of the given driven routine and the roll-back point in case of failure. Whenever a failure occurs, the entire system is rolled back to the point of failed access. The concerned peripheral is recovered and Phoenix re-executes the failed access in addition to all accesses to dependent peripherals.

Helpers for Transiently-Powered Systems

IBIS [63] IBIS is a tool that highlights non-idempotent sections. It comprises a static analysis and a dynamic run-time environment. The static analysis is generic and thus does not assume a specific programming model, but it may yield false positives and false negatives. However, it does not support recursive functions and requires the application developer to manually indicate which routines are

dependent on the peripheral inputs. On the other hand, the dynamic run-time is based on Alpaca's programming model [55] and instruments the code. Variables are constantly under taint tracking analysis, which slows the execution but yields no false positive.

TARDIS [54] TARDIS uses SRAM remanence decay to estimate the duration of power outage. It is an alternative to external real-time clocks since no specific hardware is involved, at the price of allocating some SRAM space to TARDIS. Before power fails, TARDIS loads a fixed-size buffer into SRAM containing ones. On boot, TARDIS estimates the time elapsed without power using the ratio of zeros over the total buffer size. This analysis is complemented by the usage of a temperature sensor, taking into account the fact that a temperature change might have affected the decay rate. Depending on the buffer size and specific SRAM technology, this approach enables to coarsely keep track of time across power outages from seconds to several hours.

CusTARD [92] CusTARD consists in charging an external capacitor when the platform is powered on, and measuring the analog voltage of that capacitor upon next boot. A temperature sensor also complements the approach, since temperature affects capacitor behavior. CusTARD also considers using an external low-power real-time clock, coupled with a small capacitor in the range of 0.1 µF. However, even low-power real-time clocks draw current, need an initialization time in the range of half a second every life-cycle and can measure only a few dozens seconds with a small capacitor.

3.4.2 Solutions to P1

Solutions to Program Counter Volatility

Program counter persistence defines where the application shall resume after a power outage occurred. Existing works can be coarsely spread onto two categories as far as checkpoint insertion is concerned: static and dynamic. Static checkpoint insertion elects specific code locations to be checkpoints whereas dynamic checkpoint insertion lets the application live on its own and makes any code location a checkpoint whenever an imminent power outage is foreseen.

Static Checkpoint Insertion Some works require the application developer to manually place checkpoints, either explicitly inside function bodies [58], or by imposing a task-based vision where the application developer has the responsibility to handcraft atomic tasks which boundaries are, *per se*, checkpoints [56, 55, 65, 15, 78, 57].

Some approaches statically provision the application code with checkpoints, based on energy budget [70, 71], on the likeliness of power outage to occur [21, 85, 64], or on idempotence-related concerns [61] (*cf.* P4). The static provisioning may be performed either as a compiler pass [21, 85, 64, 61] or at architectural level using high-level synthesis [70, 71].

When dealing with static approaches, the program counter of resumption is known at compile-time: it becomes a constant for a given checkpoint location, which makes checkpointing more instinctive in comparison to guessing which program counter to restore. However, compiler-based solutions are subject to checkpoint over-provisioning, due to, amongst others, loops and function calls. Large overheads result from this checkpoint over-provision, which impedes application progress. As an optimization, compiler-based solutions may change the semantics of checkpoints so that, at run-time, checkpoint locations evaluate a condition in order to decide whether to actually perform the save operation or not. This selection condition can be timer-based [64], or based on the measurement of the remaining system energy using an Analog-to-Digital Converter [21, 71, 85, 64]. On the other hand, considering the scenarios where two consecutive checkpoints are so far from each other that the energy storage might never provide sufficient energy to the system to make forward progress, compiler-based solutions might enhance their static checkpoints with timer-based checkpoint watchdogs, that save the system upon timeout and introduce hints of dynamicity [61].

Static analysis further presents the advantage of knowing, at any point in the control-flow graph, the liveness of the variables. However, variables that fit in CPU registers are not really a concern since the CPU usually has a small amount of registers, meaning that the overhead of saving all CPU registers

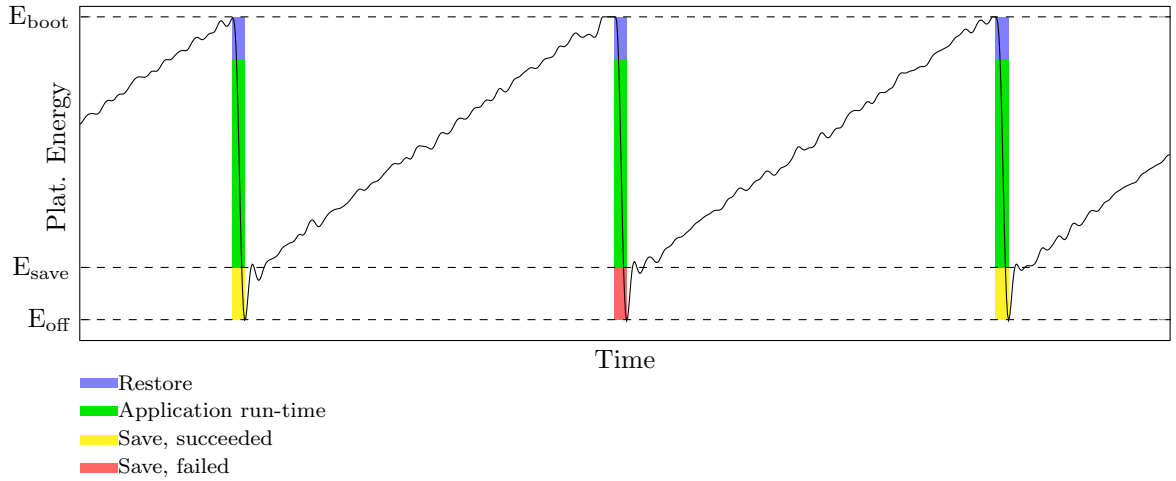


Figure 3.2: Typical run-time of just-in-time checkpointing systems that initiate a checkpointing operation whenever the available energy drops below a threshold. Every life-cycle starts with restoring a valid checkpoint, in blue, then the application resumes, in green, and the checkpointing routine is called once before power fails. Either the checkpointing operation succeeds, in yellow, or fails, in red.

regardless of variable liveness is reasonably acceptable. Variables that spill registers are stored in the stack, so their liveness is already taken care of by stack management.

Dynamic Checkpoint Insertion Other approaches provide just-in-time checkpoints. They usually leverage the measurement of the remaining energy of the energy storage by an analog comparator capable of triggering an interrupt when the energy storage voltage drops below a certain threshold [22, 86, 59, 66, 77]. Figure 3.2 illustrates the just-in-time checkpointing concept. The comparator feature is very common in most micro-controllers, making this solution easy to adopt. However, the voltage comparator must work hand in hand with a voltage reference module. The latter is the most constraining part of just-in-time checkpointing systems since voltage reference capabilities vary from a micro-controller to another. In the worst case scenario, just-in-time checkpointing does not exclude the usage of an external, dedicated voltage reference that would allow a finer configuration of the checkpointing voltage threshold, despite increasing form factor and energy demands.

Reli [68] investigates the interesting area of architecture, by acting at processor micro-operation level. Every instruction that writes into CPU registers or memory is modified so that it logs the written values and their corresponding addresses, either in the register file or in memory, into a log. The log operates at control-block level, hence almost every instruction is candidate to be a valid checkpoint. Clank [69] is Ratchet's architecture-based counterpart that also aims at breaking non-idempotent dependencies by inserting a checkpoint between the write and the read operations. Clank records reads and writes to non-volatile locations and dynamically triggers a checkpoint when the system performs a write operation to a memory location that was read prior to that point. However, although architecture-based solutions are always interesting and clever, they are rarely accessible nor is it realistic to foresee their implementation in common micro-controllers later on.

It is important to note that just-in-time checkpointing also supports periodical checkpointing as well as static checkpointing. Just-in-time checkpointing is then more of an optimization of such solutions, aiming at reducing the amounts of checkpoints per life-cycle to its strict minimum, that is, once. As a result, some solutions are not bound to any specific checkpointing paradigm, but are compatible with both static and just-in-time checkpointing [84, 67, 89].

Solutions to Memory Volatility

Architectures A2 and A4 make no use of volatile RAM. As a result, solutions relying on these architectures avoid memory volatility [58, 59, 61, 69, 78, 77, 89].

The other architectures use volatile memory and are thus subject to memory volatility. This discussion is rather technical than conceptual since its essence is largely considered trivial. Only its optimizations, notably to reduce the duration of saving and restoring memory, are studied. Saving the volatile memory may be as simple as a copy of the whole volatile memory in use (*i.e.*, `.bss` and `.data` sections) to non-volatile memory. Restoration can thus be achieved by mirroring the save operation, by copying the data from non-volatile memory back into volatile memory. All micro-controllers embed a Direct Memory Access (DMA) module, a physical part of the micro-controller dedicated to memory transfers. They can be configured to perform fast copies between memory locations. The DMAs are wired to the address and data buses, meaning that they can operate on any permitted address range, regardless of the underlying memory type. They can issue read and write commands to volatile and non-volatile memories, which makes DMAs bare essentials to checkpointing for transiently-powered systems. Some works, related to incremental checkpointing as discussed in Chapter 6, investigate ways to reduce the amount of data to be saved and restored. Handled properly, it does not affect the checkpointing methods correctness, but rather decreases their execution time so that a greater part of time and energy budgets can be allocated to the application itself. On the other hand, TotalRecall [93] stands as an exception for it assumes that the supply voltage never drops below the minimal retention voltage of the volatile SRAM, around 0.4 V, and thus claims that saving volatile SRAM is not necessary, as long as the off-times are not long enough to endanger the assumption. The assumption is very specific and substantially limits design space, however TotalRecall is an outlier worth mentioning.

The micro-controller registers may be directly saved into and from non-volatile memory, using `store` and `load` instructions. Otherwise, they can be saved and stored to and from the stack using `push` and `pop` instructions. If the stack is non-volatile then the issue is settled. If the stack belongs to volatile memory, it may thus be saved and restored alongside the rest of the volatile memory, as aforementioned. Special micro-controller registers might not be handled as easily. For instance, in ARM-based architectures, the least significant bit of the program counter indicates whether the micro-controller runs in thumb mode. Depending on the architecture, the read program counter may correspond to the address of the next instruction instead of the current instruction. In general, the Application Binary Interfaces are designed to handle special registers in some efficient way, upon interrupt occurrence, by pushing the adequate copies onto the stack and thus simplifying the checkpointing. This gives a helping hand to interrupt-based checkpointing systems, either timer-based or just-in-time. Finally, the stack pointer might not be allowed to be pushed onto the stack by itself, thus needs to be treated apart from the other registers, manually stored and restored to and from a non-volatile variable for instance.

3.4.3 Solutions to P2

In many works, the volatility of peripherals is not studied [21, 68, 61, 85, 56, 69, 89]. Other works expect application developers to manually provide peripheral state persistence and hence do not support it [59, 58, 65, 64, 78].

Samoyed [77] assumes that every function re-initializes the peripherals it uses. This makes sure that the peripherals are in the state they are expected to be in. However, it complexifies peripheral programming and introduces a high overhead when re-configuring peripherals is not really required; *e.g.*, if the peripheral configuration did not change since the last time the function was called. At the end of the day, the application developer remains in charge of making peripheral state persistent across power outages.

The Alpaca-based task-oriented systems assume that the system is provided with a routine that initializes peripherals on every reboot [55, 15, 57]. In Alpaca-based languages, the peripherals are assumed to always have the same state for a given task. This, combined with task atomicity, gives homogeneous assumptions to the application regarding peripherals.

Hibernus++ [86] keeps a copy of memory-mapped peripheral control registers. This might be sufficient to *partly* support really simple peripherals. As seen in Section 2.6.3, it is not sufficient to *fully* support peripherals, even simple ones: control registers might require a specific sequence of operations, timing constraints, write-only control registers cannot be read, and some peripherals are accessed through data buses.

Solutions to P2.1 RESTOP [84] logs every peripheral access that happened during the application lifetime. In order to postpone log overflow, RESTOP proposes a log compression technique that leverages the application developer who must manually tag *every* peripheral access. The tag indicates whether to save the operation into the log, replace an old entry or save yet keep a former similar operation. In order to restore the states of the peripherals, RESTOP must go through the entire log and replay every formerly-logged operation, which takes a long time. As a result, the log of peripheral operations must be part of the checkpoint image.

KARMA [67] represents drivers as finite state machines alongside a queue of peripheral operations, similar to that of RESTOP. The queue of peripheral operations aims at keeping the state machines simple, where the parameterization of the operations are left to the operation queue. The queue benefits from compression that eliminates former operations that led to the same driver state. Since the state machines are finite, the operation queue size is bounded. Similarly to RESTOP, the queue of peripheral operations must be executed at some point, to restore the states of the peripherals. Hence, the checkpoint image must contain the state machines and the queues of peripheral operations.

Solutions to P2.2 Only a few works support interrupt handling, despite being at the core of embedded system programming. Interrupt handling brings concurrency which might break assumptions whereas solutions for transiently-powered systems want to make sure that certain properties, namely idempotence-related, are ensured. Handling power outages is already a challenge for such systems and supporting interrupts in such a context can only increase complexity.

KARMA [67] enables the application to register callbacks to respond to hardware interrupts. The "waiting" phase of the peripheral-based asynchronous operations is encoded inside the peripheral state machines. Peripherals that allow multiple asynchronous operations require a state machine which size grows according to all possible combinations, making both run-time conditions and driver programming delicate to maintain. KARMA saves the current state of all peripherals. Interrupt occurrence is not made persistent itself, however KARMA recalls that a given peripheral was waiting for a callback: on reboot, KARMA replays the peripheral operations from the peripheral operation log, leading to the restoration of the corresponding state related to the asynchronous operation. Hence, KARMA only supports asynchronous calls which callbacks can fully execute before the energy storage depletes. The callbacks cannot survive power outages nor are the interrupt-bound data made persistent.

HarvOS [85] takes interrupt handlers into account while statically inserting checkpoints. Since an interrupt handler can be fired almost from anywhere in the code, HarvOS systematically inserts a checkpoint site at the beginning and at the end of every interrupt handler. A *checkpoint site* is a point in the code, statically inserted by the compiler, that first measures the remaining energy of the platform and then decide whether to actually perform a checkpoint operation. Interrupt handlers are treated the same as regular code in HarvOS model and thus, interrupt handlers have checkpoint sites inside their control-flow graphs. Checkpoint sites are inserted so that the energy storage can provide energy for a sub-path between two consecutive checkpoint sites to eventually complete, ensuring forward progress. As a result, the control-flow of interrupt handlers is made persistent by HarvOS. This mechanism enables HarvOS to make interrupt occurrence persistent. Interrupt-bound data can be made persistent if the first action realized by the interrupt handler is storing the interrupt-bound data directly from the peripheral memory into either a variable managed by the checkpointing system, or into non-volatile memory. However, HarvOS focuses on the computational logic so it does not provide explicit mechanisms to save interrupt-bound data.

Coati [57] uses interrupts as a way to schedule tasks: interrupt occurrence generates a Coati event, that tells Coati's scheduler to run some task in the future. Coati uses a *split-phase* interrupt model that resembles the models of Linux tasklets [5], TinyOS [43] and Sytare [66]. In Coati, the *top-half* of an event always preempts the application, while its *bottom-half* is deferred to be scheduled later. The top-half is in charge of acknowledging the interrupt at hardware level as well as storing the interrupt-bound data as soon as possible. The bottom-half is written by the application developer and only serves the application purposes, since the hardware-level interactions were taken care of by the matching top-half. The persistence of Coati's scheduler ensures the persistence of interrupt occurrence. The reactivity of the top-halves and the fact that they are designed to extract the peripheral data at earliest altogether

[5] `https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html`

ensure the persistence of interrupt-bound data.

3.4.4 Solutions to P3

Timeliness constraints need a mechanism that enables a given sequence of operations to run without power outage. All existing solutions propose to opportunistically execute the time-constrained section and provide a roll-back mechanism in case power failed during the execution of the time-constrained section.

Disabling Checkpoints Mementos [21] proposes atomic regions by simply disabling the checkpointing mechanisms for the sections that require timeliness. In some cases, all interrupts may be disabled as well [94].

Chinchilla [64] arbitrarily decomposes the application into blocks that are bounded by checkpoints. Checkpoints cannot be inserted anywhere else than at the beginning and at the end of any block. Hence, on boot the application can only restart at the beginning of a block, not at its middle, ensuring block atomicity. Chinchilla lets the application developer define atomic blocks through syntactic sugar. Atomic blocks defined in such a way prevent Chinchilla's compiler from decomposing these atomic blocks into smaller blocks and thus, accordingly to Chinchilla's definition of block, impedes the insertion of checkpoints inside the atomic block.

TICS [89] enables the application developer to manually annotate portions of code that must be made time-consistent. TICS comes with a language that provides not only atomicity as part of its syntax, but also enables to parameterize the behavior of the application, depending on the discrepancy between the application's timely specifications and the actual run-time execution. Its programming model essentially leverages a `try/catch` approach, where the supported exceptions are time-related. TICS implements the time-related blocks by automatically disabling checkpoints, *e.g.*, to guarantee that the sensed data and their correlated time-stamps are consistent, then placing a checkpoint afterwards. The application developer has to provide fallback functions for each portion of code that requires timeliness and those fallback functions are not themselves subject to timeliness.

Partial Checkpoints KARMA [67] and Samoyed [77], similarly to Sytare [66], allow checkpoints anywhere in the code, but if power fails during a peripheral operation, the application will restart at the very beginning of that peripheral operation. Peripheral operations are defined as API functions, meaning that atomicity is ensured at API function granularity. Samoyed further requires the driver developer to provide, for each peripheral operation, a scaling rule aiming at reducing the energy requirements of that peripheral operation if power failed during its execution. While it enables a better energy utilization, it over-complexifies peripheral programming and increases risks of bug since the developer must design and maintain the actual peripheral operation as well as the burdens and outcomes of scaling them down.

Atomicity by Construction DINO's model [58] requires the application developer to manually place barriers in the code. DINO then sees every possible path between consecutive barriers, even across complex control-flow and across function calls, as an atomic path. Each path then starts and ends with a barrier, materialized by a checkpoint in practice, and partial progress within such a path is always discarded.

Chain [56], the Alpaca-based solutions [55, 15, 63, 57], Mayfly [65] and InK [78] are tasked-based approaches. Each task is designed to leverage a consistent memory snapshot (P1), to run opportunistically and to discard any task-related partial progress whenever power fails during the execution of a task. On boot, the application always restarts at the beginning of the task that was interrupted, thus enforcing atomicity. SONIC [95] relaxes task atomicity by allowing loops with idempotent iterations to be resumed where they were interrupted, thus optimizing neural network applications.

Automatic Atomicity Ratchet [61] is a special case because it does not support peripherals. However, Ratchet provides small atomic regions for memory operations, computed automatically and statically, so

that memory-related non-idempotent operations can be safely executed without endangering the execution. This mechanism solves P4.1 and P4.2, however it is worth mentioning that it is closely related to atomicity.

3.4.5 Solutions to P4

Solutions to P4.1

Undo-logging Chinchilla [64], Samoyed [77] and TICS [89] opt for an undo-log for non-volatile variables. Write-access to pointers is further instrumented by TICS in order to provide protection against non-idempotence for pointers which target addresses cannot be resolved at compile-time. Undo-logging [96] keeps a history of the variable contents, hence an optimized log management is required to fit transiently-powered system conditions.

Double-buffering Task-based approaches for architecture A1 usually place task-local variables in volatile RAM and shared variables in non-volatile RAM [56, 55, 65, 78, 57]. Only shared variables are stored in non-volatile memory, so only shared variables are subject to P4.1. They are double-buffered, meaning that the tasks work on a *privatized* copy of the actual shared variable. As a result, the original values of the shared variables are kept in non-volatile memory, in consistence with the last checkpoint. Variable privatizations may be stored either in volatile memory or non-volatile memory [56, 55, 57]. It does not matter for correctness, but this design choice may have an impact on execution time and energy requirements, since volatile and non-volatile memories do not have the same performance. In these task-based solutions, since atomicity is guaranteed at task level, the copy of the non-volatile variables is consistent with the beginning of the task, that is the point of resumption after reboot. When a task completes, the new values of the non-volatile variables are copied back to their safe locations in a commit manner, becoming consistent with the newly-crossed checkpoint, *i.e.*, next task.

Approaches not based on tasks, such as DINO [58], ensure non-volatile memory consistency by saving the non-volatile memory contents alongside the checkpoint and restoring them to the same state on reboot. The main difference with task-based approaches is that checkpoints could be placed almost anywhere in the code, not necessarily at the beginning, and thus initialization time, of tasks or code execution units.

Breaking Write-After-Read Dependencies Ratchet [61] statically analyzes accesses to non-volatile memory locations. It identifies all write-after-read dependencies and inserts a checkpoint between the read and the write operations. Clank [69] is the architecture-level counterpart to Ratchet whereas Ratchet operates at compiler-level. Accesses to non-volatile memory are dynamically monitored by a specific hardware component spying on the memory address bus. Clank comprises a read buffer and a write buffer, that altogether enable to detect on-the-fly write-after-read dependencies. Upon a write access to non-volatile memory, if the address was formerly stored in the read buffer, a write-after-read dependency is confirmed and Clank triggers a checkpoint between the read and the write operations, as Ratchet does. Finally, Ratchet & Clank double-buffer their checkpoints, so that partial progress can be discarded and the non-volatile state can stay consistent with the checkpoints breaking write-after-read dependencies apart.

Solutions to P4.2

TICS [89] adopts a fully non-volatile stack. In order to optimize checkpoint size, TICS segments the stack and double-buffers the topmost stack segment, that is the portion of stack being currently modified. Older stack segments are already stored in non-volatile memory and not modified, thus do not need to be copied again. Hence, upon reboot, TICS rolls back the topmost stack segment to a state consistent with the last checkpoint.

Ratchet [61] protects non-volatile stack from non-idempotent operations by breaking `pop` instructions into a sequence of instructions that first read the data pointed to by the stack pointer, and then update the stack pointer. From that point onward, non-volatile stack contents can be considered regular non-volatile variables, and Ratchet's systematic protection of write-after-read dependencies applies for this newly-managed non-volatile stack as well.

Chapter 4

Sytare

Sytare is a lightweight kernel that pioneers peripheral handling for transiently-powered systems (P2). Its main objective, besides guaranteeing peripheral consistency across power outages, is to have a minimal impact on programming model, at both application and driver levels.

Sytare [66] is a contribution of this work, initially created a year before this work[97] and further developed within this work. Sytare already existed with its solutions to problems P1, P2.1 and P3. This work has extended Sytare's scope to support interrupts (P2.2), including interrupt handling and the scheduler itself. The present work also revised the evaluation protocol that was hardly reproducible prior to this, by designing a dedicated environment for transiently-powered systems. This chapter describes Sytare in its entirety as a way to evaluate Sytare against the problems exposed in Section 2.6, albeit including former work, for the sake of comprehensiveness.

Section 4.1 presents the architecture and memory organization supported by Sytare. Section 4.2, Section 4.3 and Section 4.4 detail how Sytare respectively addresses problems P1, P2 and P3. Section 4.5 introduces the software behind Sytare and Section 4.6 evaluates its performance in comparison to a bare-metal baseline. Section 4.7 presents ARMorik, a new ARM-based platform that was developed throughout this thesis, with more volatile and non-volatile memory, on which Sytare has been ported. Finally, Section 4.8 concludes.

4.1 Design Choices

Sytare is designed for platforms that have both volatile and non-volatile memories (architecture A1), as shown in Figure 4.1. Considering non-volatile memory slower and more energy-consuming than volatile memory, non-volatile memory is only used for storage purposes and kernel variables, while actual working memory is volatile (memory organization M1a). The application must not create non-volatile variables

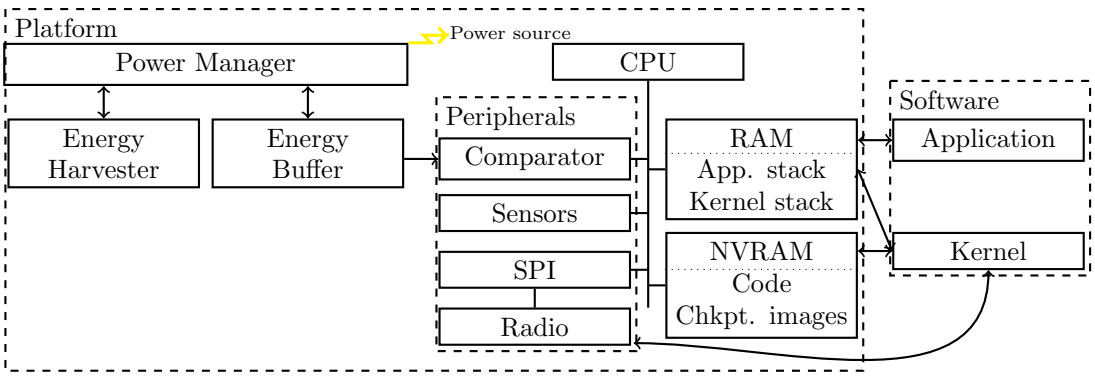


Figure 4.1: Overview of the hardware and software architectures Sytare is designed for.

and, as a consequence, Sytare does not target problem P4. Sytare targets programming ease as well, and thus tends to reduce deviations from the bare-metal baseline discussed in Section 3.2.1 (execution model E1). Programming ease is achieved by letting the application and driver developer write their code as they please with only a few constraints. The driver developer must provide a function for peripheral state restoration and save and, in exchange, Sytare guarantees that the peripherals will be restored transparently and made consistent with the state of the application at its point of resumption. From the driver developer's perspective, Sytare transparently handles variable allocation and only asks them to access peripherals through systematic wrappers, which solely adds a prefix to functions that handle peripheral operations.

4.2 Solution to P1

4.2.1 Just-in-time Checkpointing

Sytare is a just-in-time checkpointing system. It requires an analog comparator channel to be dedicated to power outage anticipation. In practice, Sytare works in scenarios where the platform is directly powered by the energy storage, as simple as a small capacitor, or where the capacitor is gated by a stable voltage regulator such as a low-dropout regulator. In both scenarios, depending on the analog capabilities of the micro-controller, notably the built-in voltage reference to compare the energy storage voltage to, just-in-time checkpointing may require the usage of external components. For instance, the voltage reference of Texas Instruments' MSP430FR5739 being limited, the energy storage voltage needs to be externally divided by two, by using a 2 MΩ voltage divider. The analog comparator triggers an interrupt when the energy storage voltage drops below a software-defined threshold. Hence, no polling is required whilst the application is running. The interrupt system also offers the advantage of pushing the return value onto the stack. Thus, making the stack persistent solves program counter volatility issue. If the imminent power outage interrupt occurs during a peripheral operation, the program counter is handled differently, as seen in Section 4.4, in order to preserve atomicity. The comparator interrupt handler pushes the CPU registers onto the stack, thus again, making the stack persistent solves the issue of CPU volatility.

Upon imminent power outage, the interrupt handler further performs a DMA copy of all used volatile RAM into non-volatile memory. Used volatile RAM includes the application's stack, `.bss` and `.data` sections. Volatile RAM usage is determined using linker script variables and does not take into account variable liveness nor does it evaluate that the variable contents actually changed since the last checkpoint. This latter point, selective checkpointing based on variable changes, is an optimization discussed in Chapter 6.

4.2.2 Double-buffering

Sytare's checkpoint images are double-buffered so that, at any time during the application's life, a valid checkpoint is always available to roll-back to. It ensures that, since checkpoint building is not atomic and power may actually fail during the process, checkpoint corruption becomes no longer an issue. It ensures that the worst-case scenario simply discards the progress of the last life-cycle, instead of completely crashing or restarting at the very beginning of the application.

4.3 Solution to P2

As discussed in Section 2.6.3, checkpointing memory contents is not enough when the system includes hardware peripherals. Restoring the state of a hardware device requires non-trivial operations like configuring some I/O pins, communicating over a serial bus (which itself should be initialized first), respecting certain timing constraints, *etc.* The solution to this issue is quite simple but requires some cooperation from the driver developer. This cooperation consists in describing the state of the device in a *device context* and in providing an `init`, a `save`, a `restore` and optionally an `on_interrupt` function for each driver.

```c
static struct clock_state_t {
    uint16_t csctl[7];
} context;

void clock_set_csctl1(uint16_t x) {
    CSCTL0 = CSKEY;
    CSCTL1 = x;
    CSCTL0 = 0;

    context.csctl[1] = x;
}
```

(a) Mapped-in-memory structures.

```c
enum clock_dco_freq_e {
    DCO_FREQ_5_33,
    DCO_FREQ_6_67,
    DCO_FREQ_8,
    DCO_FREQ_16,
    DCO_FREQ_20,
    DCO_FREQ_24
};

static struct clock_state_t {
    enum clock_dco_freq_e dco;
    /* Other clocks here */
} context;

void clock_set_dco_freq(
    enum clock_dco_freq_e dco) {
    const uint8_t values[] = {
        0x00, 0x02, 0x06,
        0x80, 0x82, 0x86
    };

    CSCTL0 = CSKEY;
    CSCTL1 = values[dco];
    CSCTL0 = 0;

    context.dco = dco;
}
```

(b) Higher level driver contexts.

Figure 4.2: Examples of drivers for the clock system of Texas Instruments' MSP430FR5739 micro-controller. Driver routines `clock_set_csctl1` and `clock_set_dco_freq` both perform the same operation: they set the global clock frequency, which is the default clock source for the CPU for instance. Semantically speaking, `clock_set_dco_freq` has more meaning and the utilization of the enumeration type prevents, to some extent, the utilization of unauthorized values, while `clock_set_csctl1` is not very meaningful and admits all inputs.

4.3.1 Device Contexts

Sytare chooses to stay as close as possible to the bare-metal baseline. In bare-metal applications, it is common to represent the state of a peripheral through structures; *e.g.*, `struct` in C language. The structures may either mirror all the control registers so that they can be directly mapped onto the physical memory addresses, or represent higher level features. For instance, the clock frequency of the micro-controller usually requires an oscillator source, plus one or several clock dividers. These pieces of information might correspond to separate control registers. The lowest representation level of clock frequency could be an actual mirror of the involved registers. A higher representation level could be the target frequency value in Hz, or an enumeration value that directly maps to a frequency. Figure 4.2 illustrates this concept with the Digitally Controlled Oscillator of MSP430FR57xx micro-controllers. Figure 4.2a shows a low-level device context that mirrors the control registers, whereas Figure 4.2b shows a higher level device context where the clock frequency is depicted by an enumeration value. Depending on the driver, a low-level device context might take a great amount of memory but be simple to copy from and to, while a high-level device context might optimize memory requirements but result in longer save and restoration times. As discussed in Section 3.2.1, high level driver states are more verbose and might complexify drivers. However, such high-level drivers are always gladly welcomed by application developers, that no longer need to dive into all the peripherals' data-sheets. Representing

peripheral-related data using enumerations further adds semantics to the code and the compiler prevents unexpected statically-resolved values from being used, which is a great asset towards driver correctness. Sytare makes no specific assumption regarding which concept of device context is chosen by the driver developer, as long as the following driver routines are consistent with the device context choice.

4.3.2 Initialization Function

In the early steps of the application, the peripherals and their device contexts need to be initialized. Hence, each driver provides an `init` function. In a bare-metal scenario, such a function would simply set the peripheral to a pre-defined or user-specified state, alongside initializing the internal structures, if any, to a consistent state with respect to the peripheral state. This still stands within Sytare, where the internal structure is the device context. However, Sytare requires little cooperation from the driver developer. The *init* function must register the corresponding driver to Sytare, by calling Sytare's `register` function.

Sytare's `register` function registers a driver that is defined by its entry points: (*i*) a `save` function, (*ii*) a `restore` function and a (*iii*) `on_interrupt` function, as depicted hereafter. Sytare records that driver for save and restoration purposes. In addition, Sytare allocates a memory place in both checkpoint images for the device context. The `register` function must thus implement a simple memory allocator. How the allocation is done is implementation-defined and not actually part of Sytare's contribution. Driver memory allocation may be as simple as a growing pointer. No memory free operation is required for applications are likely to use their peripherals throughout the entire application life-time.

Sytare also records the order with which the `init` functions are called. This order will be used for driver restoration purposes, provided that the application developer called the `init` functions in a logical order that respects driver dependencies. For instance, if the radio is accessed through an SPI bus, the `init` function of the SPI peripheral must be called before the `init` function of the radio peripheral. In a bare-metal scenario, the initialization routines are always called in a logical order that respects driver dependencies, meaning that Sytare changes nothing from this perspective. It may be noted that driver dependencies could be encoded in the driver themselves and valid sequences of `init` calls automatically generated, either at compile-time or dynamically. For the sake of simplicity, Sytare chose to rely on the application developer who, as stated before, would have called the `init` functions in a correct order anyway.

4.3.3 Save Function

The driver developer must provide a `save` function for each driver. Sytare calls the `save` function with a target address as the argument. The target address is located in non-volatile memory, inside the `next` checkpoint image, precisely at the offset allocated by Sytare when the driver first registered. The `save` function must simply copy the device context from volatile memory into that target address range. It may be noted that `save` functions can be automatically generated, which limits the burden on the driver developer's shoulders.

4.3.4 Restore Function

The driver developer must also provide a `restore` function for each driver. It must first mirror the behavior of the `save` function; *i.e.*, copy the device context from the dedicated location of the `last` checkpoint image back into the volatile device context.

The `restore` function is also responsible for setting the state of the peripheral to the one depicted by the newly-copied device context. In practice, that second part of the `restore` function is a variation of the `init` section. However, the driver must not register itself again. It must solely call the adequate driver routines to navigate from the reset state of the peripheral, to the state to restore.

Doing so is an alternative to undo-logs and other history-based approaches. It has the advantages of being simple, having a light memory footprint (*i.e.*, only code size) and running rather fast in comparison to replay a history of actions, while not being memory-growing. On the other hand, it requires the driver developer to design every path from the reset state to the states that are possible to restore.

4.3.5 Interrupt Callback

In order to provide a programming environment as close to bare-metal programming as possible, interrupts become a major feature that is seldom supported in today's operating systems for transiently-powered systems. Originally not part of Sytare before this thesis [97], adding interrupt support to Sytare is one of the contributions of this work. When considering sensor networks, the application ideally does not want to lose packets and must thus be kept reactive to external events. In addition, energy is scarce and applications cannot afford to actively poll events. It is a common optimization to set the micro-controller into a low-power mode and to react upon an interrupt, for instance.

The interrupt handlers must be short enough not to hinder application reactivity, however the operating system layer should let the application developer register their own handlers. In general, such handlers access peripherals, for instance to dump the contents of a received radio packet. Knowing in advance nothing of the application developer's interrupt handlers, it seems natural to adopt a two-phase approach, one phase purposed to perform the critical operations while the second one allows longer yet postponable operations, as done in Linux systems.

Sytare manages the interrupt's top-half with little cooperation from the driver developer. The top-half immediately runs with highest priority, acknowledges the interrupt and calls the `on_interrupt` function of the concerned peripheral. The sole purpose of the `on_interrupt` is to copy the interrupt-bound data from the peripheral memory into the volatile device driver. The driver developer has to keep in mind that `on_interrupt`, for it is written by the driver developer, is called from the top-half of the interrupt, meaning that `on_interrupt` should be meant to execute swiftly, in order to maintain a good system quality of service. It also means that the `on_interrupt` routines inherit their run-time properties from top-halves, *i.e.*, they are run on the kernel stack.

Lengthy processing of the interrupt-bound data must be handled by the bottom-half of the interrupt. The bottom-half is written by the application developer and thus runs on the application stack, in order to be made persistent should power fail during the execution of the bottom-half. The application developer must tell Sytare to register the bottom-half using a dedicated Sytare call, *i.e.*, `syt_register_event_handler`. Sytare proposes a handful of event identifiers that connect to different events and their related interrupts, *e.g.*, GPIO pin change, radio packet reception. The bottom-half is added to the scheduler by Sytare upon interrupt occurrence, right after the top-half completes.

The scheduler policy is rather simple. If the application is currently running a bottom-half, then defer the new bottom-half to the termination of the current bottom-half, and after the completion of any other waiting bottom-half that were inserted before the considered bottom-half. If the application is running bare application, then the bottom-half is executed right after the top-half completes and before the bare application resumes. In other terms, bare application has the lowest priority, interrupt top-halves the highest and bottom-halves have varying priority depending on whether a bottom-half is already running. As a consequence, the bare application may resume if and only if all top-halves and bottom-halves completed. The key idea of this scheduling policy is that the application must be kept as reactive to events as possible and nested bottom-halves are forbidden in order to keep the control-flow fairly simple and to reduce access races to variables and peripherals. Note that the choice can be made to allow nested bottom-halves, however this would further complexify system design that is already complex enough with actual interrupts and power outages.

Scheduler data, *i.e.*, system state and the queue of pending bottom-halves, are managed by Sytare and benefit from an ad-hoc management in order to prevent P4 from corrupting scheduler data.

4.3.6 Driver Routine Wrappers

Sytare acts as an intermediate layer between application and drivers. Notably, Sytare defines a generic driver routine *wrapper*, that takes effect both at the entry and upon returning from driver routines. The application cannot directly call the driver routine, but must call the corresponding wrapper instead. In particular, the wrapper exit is in charge of saving the device context changes from the volatile working copy into the `next` checkpoint image. A more exhaustive description of driver routine wrappers is given in Section 4.4, as a consequence of wrappers tackling both P2 and P3.

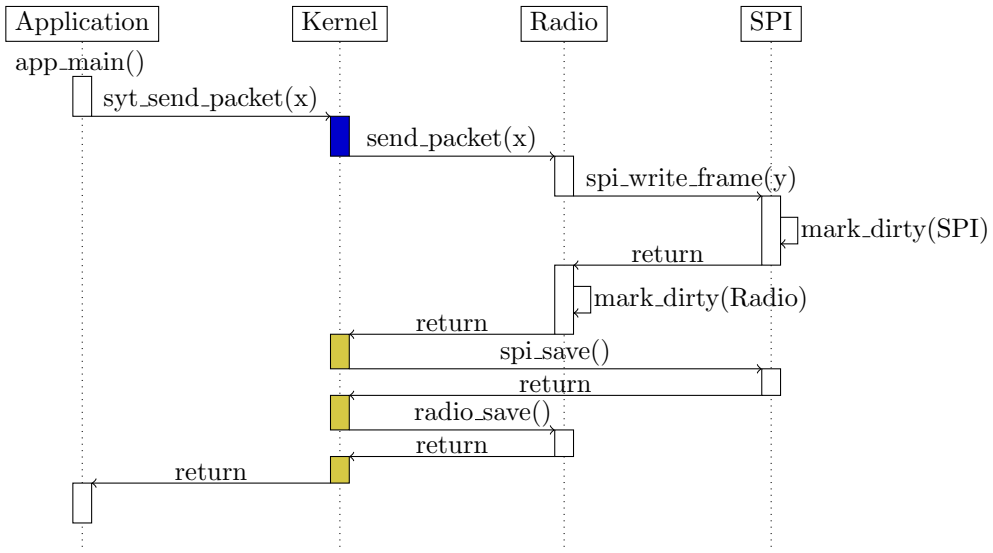


Figure 4.3: Sequence diagram of a nested driver call, showing the interactions between Sytare, the drivers and the application. The blue activity is the wrapper entry and the yellow one is the wrapper exit.

4.3.7 Nested Driver Routines

It is not uncommon that a driver routine calls another driver routine. For instance, a driver routine that operates on a radio chipset accessed through an SPI bus must issue SPI commands. In that example, the radio handling operation is encapsulated in a specific radio driver routine, while the SPI handling operations are encapsulated in specific SPI driver routines.

Wrappers are designed to ensure atomicity of the entire peripheral operation, regardless of the order of driver routine nests. Wrappers thus need to operate on the outer driver routine.

In Sytare's model, the wrappers perform a transition from application context to kernel context on entry and from kernel context to application context on exit. Then, all driver code is run in kernel mode. As a consequence, when a driver routine needs to call another driver routine, regardless of originating from the same driver or not, it must not use wrappers but must directly call the other driver routine. This policy enables nested driver routines to provoke a single context change at the beginning of the outer driver routine and a single context change at the very end of the outer driver routine. It also ensures that no partial progress is saved, for the commit operation is performed only when all driver routines have completed. Figure 4.3 displays how the application interacts with the drivers through the usage of Sytare's wrappers.

4.4 Solution to P3

Timeliness and atomicity constraints require the application to be rolled-back in case power failed during the execution of such constrained code sections. All kinds of memory must be consistent with one another. Specifically, peripherals must be rolled-back to their state at the beginning of the atomic section in addition to CPU and memory states.

It is important to note that Sytare solely considers timeliness constraints within peripheral accesses. Indeed, Sytare does not directly provide atomic semantics to the application developer. As a result, a computation is allowed to take several hours when taking off-times into account. Sytare supports timeliness and atomicity constraints for peripheral operations however; *e.g.*, if power fails while a radio packet is being sent, Sytare repeats the packet sending attempt upon reboot instead of trying to send "the rest of the packet". However, when reading data from sensors, Sytare does not timestamp values nor does it guarantee that the usage of the data will be completed in a given time-span. In Sytare's model, sensor data usage is managed by the application, while Sytare simply ensures that the sensing itself is time-consistent. But this limitation may be easily bypassed by encapsulating the timely code into

a function and calling it using Sytare's wrappers.

4.4.1 Application and Kernel Stacks

Sytare has two stacks: the application stack is dedicated to the application and the kernel stack, a smaller one, is dedicated to Sytare's own mechanisms. The application always runs on the application stack while Sytare always runs on the kernel stack. The application stack is part of the checkpointing image whereas the kernel stack is not. They are both stored in volatile memory, meaning that it is part of the design that the application stack is made persistent and that the kernel stack is always lost upon power outage. This property lays ground for atomicity, as atomic sections would benefit from running on a disposable stack.

Theoretically, it is not absolutely necessary to execute kernel and application onto separate stacks. In a scenario without P2.2, the bottommost part of the stack is owned by the application. The stack grows but is always owned by the application. An imminent power outage would push some kernel interrupt data on top of the stack. Hence, it is straightforward to distinguish stacks and, since the application stack is the only one made persistent, storing the application's stack pointer value is sufficient. Indeed, upon reboot, the kernel may restore the stack only up to the recorded stack pointer and reset the stack pointer to the recorded value. With the introduction of P2.2 and its solution based on kernel-managed top-halves and application-managed bottom-halves, a single stack would have kernel-owned and application-owned variables interleaved. Using a single stack would thus not be as straightforward as in a scenario without P2.2.

4.4.2 Driver Organization

As aforementioned, Sytare requires the driver developer to provide a device context structure for each driver, containing whatever information the driver developer considers necessary to be able to restore each driver to any restorable state. The device context has one volatile instance, stored in volatile memory, that is the working device context. The device context has two non-volatile instances, stored in the `last` and `next` checkpoint images. To grant atomicity to the application at driver level, device contexts are managed in a commit-based manner.

4.4.3 Driver Routines

Sytare asks the driver developer to work only on the volatile working instance. If a driver routine changes the state of the peripheral and thus, of the driver, the driver routine must set a volatile flag that indicates that the driver is *dirty*, *i.e.*, modified. The dirtiness flag is stored in volatile memory, alongside the working device context. Each driver has its own dirtiness flag.

4.4.4 Driver Routine Wrappers

Sytare acts as an intermediate layer between application and drivers. Notably, Sytare defines a generic driver routine *wrapper*, that takes effect both at the entry and upon returning from driver routines. The application may not directly call the driver routine, but must call the corresponding wrapper instead.

The wrapper entry has two missions: (*i*) recording the address and arguments of the targeted peripheral call and (*ii*) switching stacks from application to kernel stack. Indeed, since the wrapper is always called from the application context, the wrapper may safely assume that it is called from the application stack. Furthermore, in order to benefit from the disposable kernel stack designed for atomicity purposes, the wrapper entry changes stack.

The wrapper exit (*i*) commits modified drivers to the `next` checkpoint image, then mirrors the wrapper entry: (*ii*) the stack is switched back from kernel stack to application stack and (*iii*) the peripheral call record is cleared now that the `next` checkpoint image is consistent.

struct {
- save: **address**
- restore: **address**
- onInterrupt: **address**

} *Driver*

(a) Sytare's Driver structure.

struct {
- bottomHalves[]: **array of address**
- bottomHalf: **address**
- driverCall: **address**
- events: **array of integer**

} *Interrupt*

(b) Sytare's Interrupt structure.

struct {
- stack[]: **array of word**
- data[]: **array of word**
- bss[]: **array of word**
- sp: **address**
- driverCall: **address**
- driverContexts[]: **memory area**
- interrupt: **Interrupt**

} *Checkpoint*

(c) Sytare's Checkpoint image structure.

Data:

drivers[]: **array of Driver in NVRAM**
next: **pointer of Checkpoint in NVRAM**
last: **pointer of Checkpoint in NVRAM**

(d) Sytare's global variables, stored in non-volatile memory.

Figure 4.4: Data structures and global variables of Sytare.

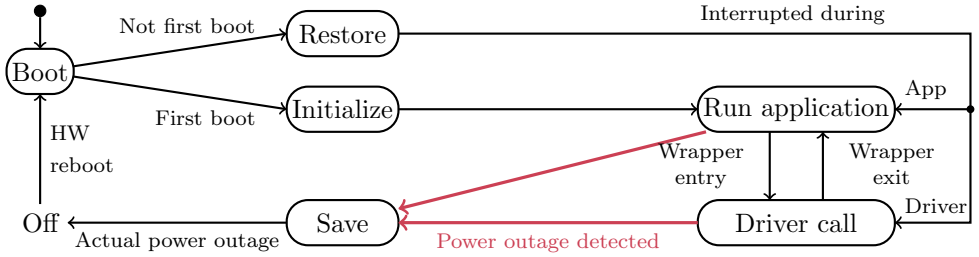


Figure 4.5: Run-time state machine of Sytare, with interrupts left aside.

4.5 Integration

In order to address problems P1, P2 and P3, Sytare needs some data structures to be defined, as illustrated in Figure 4.4.

The application run-time may be represented as a state machine that clearly shows when occur the persistence mechanisms and when the application may run. This is the purpose of Figure 4.5. Note that the distinction between the "Restore" and "Initialize" states may be removed provided that the compilation pass or the platform programming pass is able to populate a valid checkpoint image, *i.e.*, the `last` checkpoint would correspond to the application state at the beginning of the `main` function. Figure 4.5 is intentionally simplified not to represent the interrupts and scheduler policy. Given that simplification, the only states that have interrupts enabled are "Run application" and "Driver call" states. The other states cannot be interrupted by any interrupt, however the platform may obviously run out of power while run-time is in those states. Checkpoint image double-buffering makes the overall system resilient to unexpected shutdowns during those states.

When an imminent power outage is detected, Sytare finalizes the `next` checkpoint image and eventually swaps `next` and `last` checkpoint images, as illustrated in Algorithm 4.1. Checkpoint image finalization consists in copying run-time data, namely the application stack, `.data` and `.bss` sections into the `next` checkpoint image. In practice, only half of the checkpoint image swap is performed at this point, to make the operation atomic. Upon reboot, the location of the `next` image is always determined as the counterpart of the `last` image, as shown hereafter, so it is unnecessary to perform a complete swap between `last` and `next` images before power outage. Only the value of `last` matters.

Algorithm 4.1: Imminent power outage interrupt handler.

```
Input: Application state
Output: next, last
Interrupt(imminent-power-outage) begin
    acknowledge interrupt
    switch to kernel stack
    next.stack ← application's stack
    next.data ← application's .data section
    next.bss ← application's .bss section
    next.sp ← application's stack pointer
    last ← next
    shutdown
end
```

The reciprocal operation is the restoration function, illustrated in Algorithm 4.2. Sytare's restoration function is a little more complex than the save counterpart. Indeed, Sytare's `save` function is only responsible for making persistent the computational application state, for the state of peripherals is made persistent in a continuous manner, upon returning from every wrapper exit. On the other hand, Sytare's restoration function is responsible for both restoring the application state and the state of the peripherals. In addition, the restoration function must act differently upon the application being interrupted during application code, *i.e.*, "Run application" state of Figure 4.5 and during driver code, *i.e.*, "Driver call" state. The fact that application code and driver code run on separate states motivates this control-flow distinction.

The wrappers ensure incremental peripheral state persistence. Their behavior, depicted in the previous section, is illustrated in Algorithm 4.3. The wrapper exit mirrors the wrapper entry, the only difference being that the wrapper exit must commit all changes made to peripherals. The commit operation applies to all drivers, for nested driver calls might involve driver routines from other drivers, as shown in Figure 4.3. The example of wrapped driver call further shows how easy and systematic the wrapper is and that it may be automatically generated for programming ease.

Supporting interrupts complexifies the overall picture. Figure 4.6 shows the complete state machine of Sytare, including interrupt-related states. The "Bottom-half" and "Bottom-half driver call" states interact with one another like "Run application" and "Driver call" states do, for they are respectively analogous to them. Power outage detection is enabled in all application- and driver-related states, namely "Run application", "Driver call", "Bottom-half" and "Bottom-half driver call" states. The complete state automaton of Sytare involves a new kernel-side state in comparison to that of Figure 4.5, the "Top-half" state. Whenever an interrupt is fired during the execution of application- and driver-related states, system state transitions to "Top-half" state. Then, upon top-half completion, the next system state is determined by the contents of the bottom-half queue. The transitions are thus fully deterministic, for Sytare's scheduler prohibits nested bottom-halves and ensures that bottom-halves have a higher priority than "Run application" and "Driver call" states. Algorithm 4.4 shows the simple bottom-half scheduler and what an interrupt should look like, apart from the imminent power outage interrupt. Hardware operations are performed first, that is acknowledging the interrupt. A new event is added to the set of pending events. All drivers `on_interrupt` functions, *i.e.*, top-halves, are called for it is their responsibility to accept or discard the event whether they are not concerned. Modified drivers are then saved in order to make persistent the interrupt-bound data. Then, depending on whether the interrupt occurred during application context or driver context, the scheduler is invoked differently. In addition to executing on separate stacks, Sytare's scheduler also applies the worst-case policy. Sytare restarts the interrupted driver call, the same way Sytare would if power failed during the same call, in order to avoid peripheral failures. This is why drivers are restored in that specific case, to preserve peripheral state consistency. Other policies may include opportunistically resume the driver call at the same point it was interrupted, but it would not guarantee resilience against peripheral failures. Driver wrappers could also disable the other interrupts than the imminent power outage to simplify the design, but it would result in a less reactive system. Sytare thus stays quite simple and reactive at the same time.

Algorithm 4.2: Restoration function.

```
Input: last
Function(retry-driver-call) begin
    restore driver call arguments from application stack
    goto last.driverCall
end

Input: image
Data: chkpt0, chkpt1
Function(other) begin
    if image = chkpt0 then
        return chkpt1
    end
    return chkpt0
end

Input: last, drivers
Output: next, application state
Function(restore-checkpoint) begin
    next ← other(last)
    next.driverCall ← last.driverCall
    next.interrupt ← last.interrupt

    application's stack ← last.stack
    application's .data section ← last.data
    application's .bss section ← last.bss
    application's stack pointer ← last.sp

    forall d ∈ drivers do
        d.restore()
    end
    initialize voltage comparator

    if last.driverCall = 0 then
        switch to application stack
        pop CPU registers from application stack
        return from interrupt
    else
        retry-driver-call()
    end
end
```

Algorithm 4.3: Wrapper entry, wrapper exit and a typical example of a wrapped driver routine.

```
Input: Driver call arguments
Output: next
Function(wrapper-entry) begin
    push arguments onto the application stack
    switch to kernel stack
    next.driverCall ← program counter
end

Input: drivers
Output: next
Function(wrapper-exit) begin
    forall d ∈ drivers do
        d.save()
    end
    next.driverCall ← 0
    switch to application stack
    pop arguments from the application stack
end

Input: Driver call address
Function(wrapped-drvcall-example) begin
    wrapper-entry()
    goto Driver call address
    wrapper-exit()
end
```

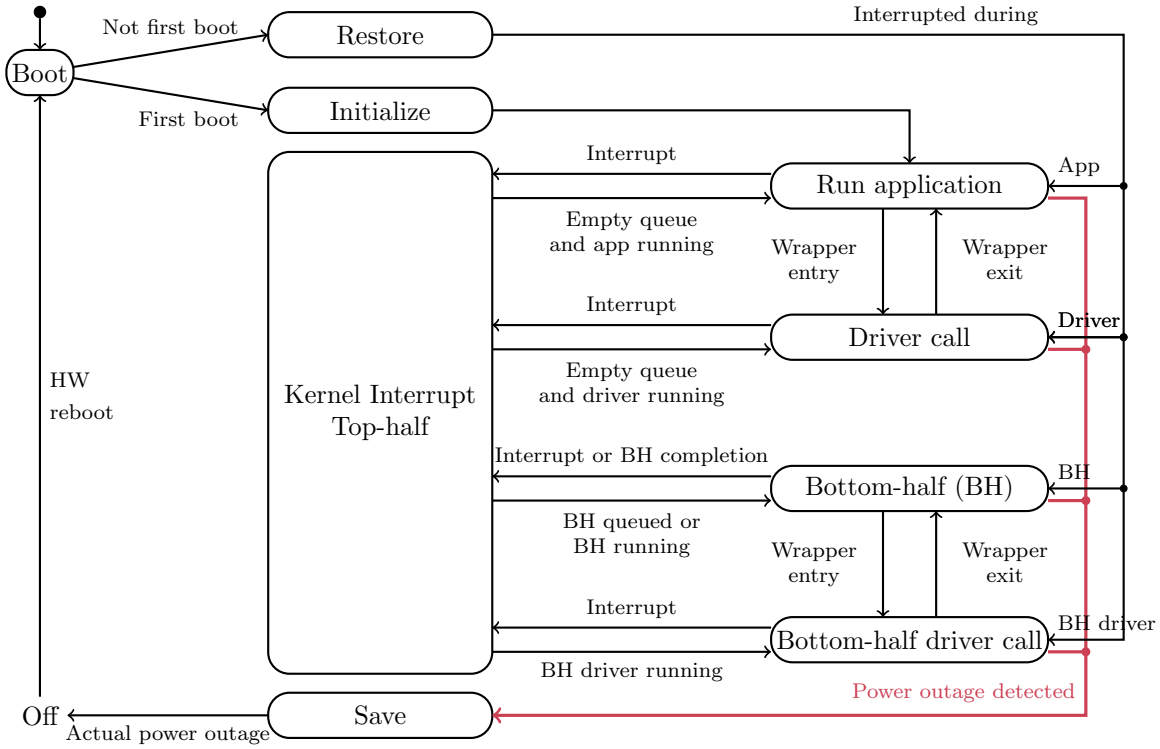


Figure 4.6: Complete run-time state machine of Sytare.

Algorithm 4.4: A typical interrupt handler, distinct from the imminent power outage interrupt, and its interactions with Sytare's scheduler.

```
Input: next
Output: next
Function(run-scheduler) begin
    while event ∈ next.interrupt.events do
        next.interrupt.events ← next.interrupt.events \{ event }
        next.interrupt.bottomHalf ← next.interrupt.bottomHalves[event]
        enable interrupts
        next.interrupt.bottomHalf()
        disable interrupts
        next.interrupt.bottomHalf ← 0
    end
end
Input: newEvent, next, drivers
Output: next, drivers
Function(generic-interrupt-handler) begin
    switch to kernel stack
    acknowledge interrupt
    next.interrupt.events ← next.interrupt.events ∪{newEvent}
    forall d ∈ drivers do
        d.onInterrupt(newEvent)
    end
    forall d ∈ drivers do
        d.save()
    end
    if on kernel stack then
        forall d ∈ drivers do
            d.restore()
        end
        switch to application stack
        if no bottom-half preempted ∧ next.interrupt.events ≠ ∅ then
            next.interrupt.driverCall ← next.driverCall
            next.driverCall ← 0
            run-scheduler()
            next.driverCall ← next.interrupt.driverCall
            next.interrupt.driverCall ← 0
        end
        switch to kernel stack
        retry-driver-call()
    else
        switch to application stack
        if no bottom-half preempted ∧ next.interrupt.events ≠ ∅ then
            run-scheduler()
        end
        return from interrupt
    end
end
```

4.6 Performance Evaluation

The present evaluation of Sytare is performed on the MSP430-EXPFR5739 board. [1] It comprises an evaluation of the time overhead of Sytare's mechanisms with respect to the overall application duration, a fine-grain quantitative time requirements of the boot sequence, discussion about minimal life-cycle duration, memory utilization and energy overhead in a real harvesting scenario.

4.6.1 Power Supply

Power supply is implemented with a programmable power supply for reproducibility. It generates a square signal that can support hundreds of mA for peak activity. It is really important that the power supply is capable of providing the current required by the platform since an unfit power supply would experience high voltage drops that would weaken the results. In the applications depicted hereafter, the current needs never get higher than 25 mA. The output of the programmable power supply is directly connected to the supply pins of the target board. This means that the power supply emulates an energy harvester which energy storage is gated behind a voltage regulator. The power supply can make life-cycles as small as 3 ms.

In addition, the power supply also includes digital pins that support a simple and efficient protocol to instrument software-define breakpoints. This feature enables the power supply to measure and record the execution time between breakpoints.

4.6.2 Benchmark applications

Sytare was tested against three applications involving various levels of interaction with peripheral devices.

LEDs LEDs slowly increases a counter and displays its value using LEDs. This application is rather simple and not actually realistic of what a useful application for transiently-powered systems would be. However, it enables to study the performance overhead of adding persistence to a simple peripheral and driver routine wrappers. In addition, this application uses eight LEDs, which makes its power consumption profile interesting to look at: the power consumption of the platform is directly linked to the value of the counter, and more precisely to the amount of ones in its binary representation.

Sense Sense senses the temperature 80 times using the internal Analog-to-Digital Converter, stores the values and the average value in volatile memory. Between each measurement, the application busy-waits 5 milliseconds.

Wireless Sensor Network Wireless Sensor Network (WSN) senses the temperature using the internal Analog-to-Digital Converter, aggregates ten measurements then sends the information alongside platform statistics, using RF, to a continuously-powered sink. The sink simply checks the contents of the radio packets and is not part of the device under test. The application then puts the radio chipset in sleep mode and waits for some time. The whole process is done 50 times. This application also demonstrates the transparent use of peripherals with inter-peripheral dependencies for more realistic scenarios, as discussed in Section 3.1.

All applications can be compiled on top of Sytare or as bare-metal applications. The evaluation of both versions of the applications serve as a basis for the present evaluation.

4.6.3 Time Efficiency

The most natural metrics, in energy-constrained systems, concerns time efficiency. Specifically, since the studied solution to execute long-running applications in an intermittent power context leverages a lightweight operating system, it is important to assess the time efficiency of the operating system's mechanisms. In other terms, an ideal operating system would have a negligible impact on the overall performance, leaving a fair execution time to the application.

[1] `https://www.ti.com/lit/ug/slau343b/slau343b.pdf`

Ground Truth Execution Time

For a given application, let T_{wired} be the time it takes to run the application to completion under continuous power, without Sytare or any persistence mechanism. T_{wired} is measured between the instant the power is turned on and the instant the application completes. As a consequence, T_{wired} includes hardware boot time as well as program initialization. T_{wired} is used as a ground truth baseline for the evaluation of Sytare, since T_{wired} corresponds to the execution time of a bare-metal version of a given application. T_{wired} is measured using the programmable power supply which was initially parameterized so that the application has enough time to complete without power outage.

Theoretically, a bare-metal scenario would probably take the shape of a super-loop, that is an infinite loop that performs the same sequence of operations over and over. Thus, it was necessary to modify this model so that the super-loop is no longer infinite but iterates a constant number of times, in order to bound T_{wired}.

Execution Time Under Intermittent Power

To establish a comparison with T_{wired}, the evaluation needs a time metrics for the Sytare-aided version of the application. In transiently-powered systems, there are on-time and off-time periods, when respectively the power is on and off. However, the off-time is of little interest, as the platform is completely inactive during those periods. Hence, the evaluation focuses on the on-time. Let T_{on} be a fixed on-time duration for all life-cycles. For each point in Figure 4.7, the power supply is configured to deliver a given T_{on}. The platform is then repeatedly powered for this duration, then powered off. Hence, T_{on} is a parameter of this evaluation.

Let $T_{\text{transient}}(T_{\text{on}})$ be the time needed for the Sytare-aided version of the application to complete, for a given value of T_{on}. $T_{\text{transient}}(T_{\text{on}})$ is defined as the sum of all on-time periods through all life-cycles. Off-time periods are not studied as they do not give any information, nor may the application make progress during off-time periods. However, $T_{\text{transient}}(T_{\text{on}})$ includes the boot time, both hardware and software, and the durations of the checkpoint operations. $T_{\text{transient}}(T_{\text{on}})$ is measured using the programmable power supply, programmed in accordance to T_{on}.

The overhead of Sytare, in terms of execution time, is expressed through the *effective yield* $Y(T_{\text{on}})$, for any value of T_{on}, as defined in Equation (4.1).

$$Y(T_{\text{on}}) = \frac{T_{\text{wired}}}{T_{\text{transient}}(T_{\text{on}})} \tag{4.1}$$

From the definitions of T_{wired} and $T_{\text{transient}}(T_{\text{on}})$, the effective yield is expected to be bounded between 0 and 1.

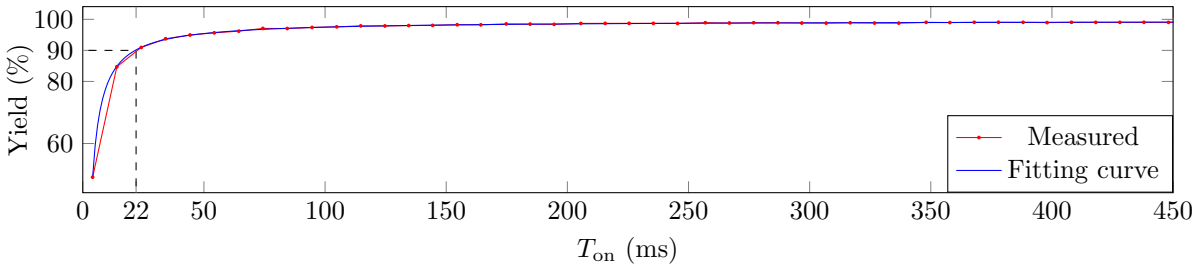
Minimal Life-cycle Duration

For small values of T_{on}, the chances the platform can successfully boot are identical to none and the application will never finish. In other words $T_{\text{transient}}(T_{\text{on}})$ would be infinite and the effective yield would be zero. The minimal life-cycle on-time required for the application to make progress, named $T_{\text{on}}^{\text{min}}$, is defined by Equation (4.2).

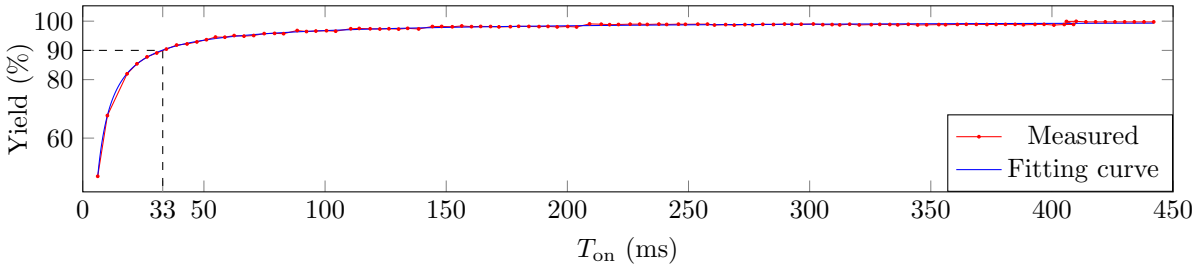
$$Y(T_{\text{on}}^{\text{min}}) = 0 \ \wedge \ \forall T_{\text{on}} > T_{\text{on}}^{\text{min}}, Y(T_{\text{on}}) > 0 \tag{4.2}$$

Maximal Yield

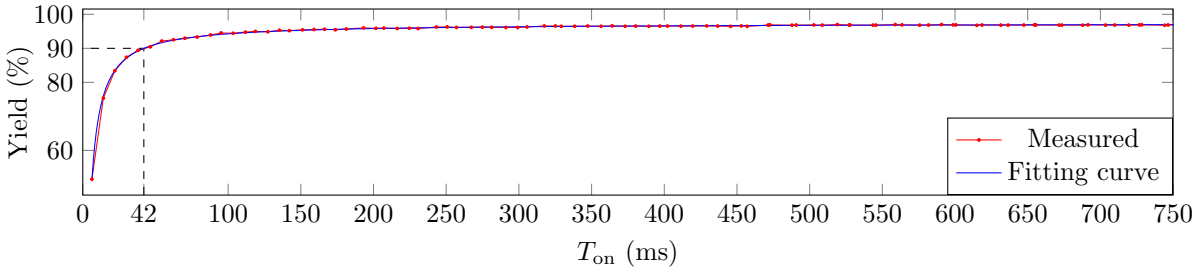
On the other hand, when T_{on} approaches T_{wired}, then the application will run to completion in just one life-cycle with little kernel interaction. In this case, the overhead corresponds to the driver routine wrappers only. Indeed, without power outage, there is no checkpoint save nor restore operation, nor multiple hardware boot times. Thus, even though the kernel boots only once and never has to save or restore checkpoints, execution overhead arising from the system call wrappers still impacts performance and the effective yield will never reach 100%. For this very reason, this particular situation makes Sytare's time overhead minimal, leading to the highest yield values. The maximal yield, named Y^{max}, corresponds to this situation and bounds Sytare's yield.



(a) LED counter application.



(b) Sense application.



(c) WSN application.

Figure 4.7: Temporal yield measurements as a function of life-cycle duration (T_{on}) for all applications.

Measurements and Results

For each application, the monitoring programmable power supply measures T_{wired} and $T_{\text{transient}}(T_{\text{on}})$. This information enables to compute the effective yield $Y(T_{\text{on}})$, shown in Figure 4.7, respectively for the LEDs, Sense and WSN applications. The yield measurements match a hyperbola model, shown in blue in Figure 4.7. A hyperbola has an equation like $\hat{Y}(T_{\text{on}}) = \frac{a}{T_{\text{on}}} + Y^{\max}$. The fact that the hyperbola model fits well the real-life measurements allows to compute the smallest life-cycle duration for which the application can be executed to completion in a transient power context, $T_{\text{on}}^{\min}$. Indeed, since the slope of $Y(T_{\text{on}})$ is almost infinite when T_{on} is small, $T_{\text{on}}^{\min}$ cannot be measured with great accuracy. However, it may be precisely estimated by using the model, notably $T_{\text{on}}^{\min} = -\frac{a}{Y^{\max}}$. The maximal yield, $Y^{\max}$ is, however, measured by reading the asymptotic value of the yield measurements.

The results are given in Table 4.1. The LEDs application uses only LEDs, thus only uses one driver that must be made persistent across power outages. Although the driver is simple, the overhead of Sytare

Table 4.1: T_{wired}, $Y^{\max}$ and $T_{\text{on}}^{\min}$ for each application. T_{wired} *std dev.* is the standard deviation of T_{wired} over a hundred samples.

App.	T_{wired} (ms)	T_{wired} std dev. (µs)	$Y^{\max}$	$T_{\text{on}}^{\min}$ (ms)
LEDs	1003.61	5.06	0.9953	2.04
Sense	410.56	4.61	0.9994	3.27
WSN	891.80	360	0.9729	2.70

is visible as the yield does not exceed $Y^{\max} = 99.53\%$. The Sense application uses multiple peripherals but stays efficient in terms of yield ($Y^{\max} = 99.94\%$) as the different drivers in use do not have a long hardware initialization. The WSN application is a realistic application used for Sytare evaluation. The utilization of a complex active RF transceiver decreases the maximal yield of the application, as RF chip initialization and restoration require active polling and multiple SPI transactions. Besides, the action of sending a message consumes much more current, inducing a voltage drop and increasing the risks of failed checkpoint or of having to retry a code section with timeliness constraints. Despite these points, Sytare is able to run the application on transient power under intermittent supply. The overhead of driver routine wrappers is rather low since the application reaches $Y^{\max} = 97.29\%$ yield efficiency.

In addition, Figure 4.7 emphasizes the life-cycle requirements to perform 90% yield efficiency. The LEDs application only needs life-cycles as short as 22 ms to achieve 90% yield efficiency, the Sense application needs 33 ms life-cycles and the WSN application 42 ms life-cycles. In comparison to the respectively 1003 ms, 410 ms and 891 ms given by T_{wired}, the requirements to ensure a certain quality of service are far less constraining than in the ground truth without checkpointing mechanism.

Except for the first life-cycle, any life-cycle comprises a restoration and a checkpointing phase. If a life-cycle lasts less than restoration time and checkpointing time, application cannot make any progress. In addition, timeliness constraints dictated by the application require some parts of the software to be executed at once or retried from the beginning if power failed. Thus, depending on the application, a life-cycle must be long enough for the longest or most energy-consuming driver routine, in addition to restoration and checkpointing times.

Table 4.1 also shows the minimal durations of a life-cycle, called $T_{\text{on}}^{\min}$, for various applications. Applications that are not peripheral intensive, such as LEDs, can make progress with shorter life-cycles than applications such as Sense and WSN. The minimal life-cycle duration is however different between applications, depending on what peripherals they use. Typically, an application using the radio frequency chip under Sytare is limited by the energy consumption of the restoration function of the RF driver. This sets the lower bound of the viable life-cycle duration to above 3 ms. The other applications do not use this peripheral and their minimal life-cycle duration is around 2 ms. This means that the energy harvester only needs to provide enough energy for the platform to run at least for a few milliseconds. It also demonstrates that Sytare efficiency depends on the application's usage of peripherals and on the complexity of the driver API.

The conclusion about this evaluation is that all applications succeed in running under intermittent supply using Sytare. Applications can run over much shorter life-cycles than required to run them to completion in a credit card model, without losing much yield efficiency.

4.6.4 Kernel

Running the aforementioned applications, the yield did come close to 1 but not actually reach it, because of Sytare's mechanisms. This section focuses on the sources of overhead of Sytare: restoration, save and driver routine wrappers. In addition, these mechanisms also require memory. Non-volatile memory when speaking of checkpoint images, and volatile memory when it comes to Sytare's dedicated kernel stack and application stack. Finally, aside from time concerns and looking directly at the overall energy consumption, the energy requirements of Sytare's restoration and save operations are measured for the WSN application, using a real energy harvester from industry.

System Boot and Restoration

Table 4.2 shows the time spent by the kernel in various phases of the boot sequence, measured for the WSN application. A great portion of boot time is dedicated to booting the platform, which is inherent to the platform itself and not a result of Sytare's doing, and restoring the peripheral state which is the responsibility of Sytare. On the other hand, restoring the state of the application itself and restoring the device context, which is the software part of the restoration, take a total of 72 µs only, being several orders of magnitude faster than the platform hardware boot and the restoration of peripherals. It must be noted that restoring peripheral state does not take the same amount of time as it depends on the amount and type of the peripherals. An active polling phase in the peripheral re-initialization, or the size of the configuration to restore, are factors that impact boot time. More precisely, the majority of

Table 4.2: Booting sequence of WSN application.

Phase	Time (µs)	Ratio (%)
Hardware boot	1240	49.4
Application restoration	45	1.8
Device context restoration	27	1.1
Peripheral state restoration	1170	46.5
Checkpoint initialization	30	1.2

Table 4.3: Temporal impact of driver routine wrappers.

Driver routine	Wrapper time overhead
LED toggle	+1325%
Sense temperature	+27%
Radio sleep	+137%
Radio wake up	+8%
Radio packet send	+1%

the time spent in restoring the state of peripherals is taken by the restoration of the radio device (around 78%), far above the ADC converter (8%), the SPI bus (6%), the clock (4%) and the GPIOs (3%).

System Save

In its current version, Sytare copies the different sections from volatile memory to the `next` checkpoint image, regardless of whether they were modified or not. This is prone to further optimization, for instance with incremental checkpointing seen in Chapter 6. However, without optimization, for a given application, the amount of copied data is always the same, which makes the save execution time deterministic and constant as well. More specifically, for the LEDs and WSN applications, the save time is 26 µs. Note that the checkpointing trigger voltage that generates the imminent power outage interrupt has been set experimentally to a value large enough to allow checkpointing completion in normal conditions. Dynamically adapting this threshold may be important for optimization of transiently-powered systems with just-in-time checkpointing.

Driver Routine Wrappers

The overhead induced by Sytare is not only the sum of the restoration and save times. During application execution, the calls to driver routines via system calls induce time overhead. This overhead is shown in Table 4.3 for several drivers routines. The overhead induced by Sytare for context switch and for saving device contexts depends on the complexity of the peripheral to be accessed and the complexity of hardware actions achieved during a driver routine. The proportion of run-time dedicated to wrappers with respect to the driver routine itself varies from more than 90%, for the LED toggle routine, to an extremely low value, less than 1% for the radio packet sending routine. In practice, as far as the LED toggle routine is concerned, the routine itself consists of a handful of instructions only, and the wrappers only add a dozen microseconds. The relative overhead is high tough, for the LED toggle routine takes less than a microsecond. Nevertheless, the impact of driver routine wrappers on the overall runtime is low, as yield values can reach up to 97%. This may be explained by the fact that driver routine wrappers take about the same duration for all drivers. Hence, short-running driver routines such as toggling an LED are more impacted than already long-running driver routines such as sending a packet over radio.

Memory Occupation

The RAM overhead of Sytare is mostly imputable to the need of a separate kernel stack of size limited to 128 bytes. The MSP430FR5739 micro-controller only embeds 1 kB of volatile RAM and Sytare makes the choice to grant most of it to the application needs. The driver memory occupation in RAM is increased approximately by the size of the mirrored configuration. For example the RF chip driver that comes

Table 4.4: NVRAM requirements, in bytes, of some applications. Checkpoint image size is accounted twice in the total NVRAM use, for checkpoints are double-buffered.

App.	Total NVRAM use	Checkpoint image	Relative checkpoint size (%)
LEDs	6952	385	11.1
Sense	8696	405	9.3
WSN	11994	503	8.4

Table 4.5: Kernel stack utilization, in bytes, for button and radio applications, under several runtime scenarios.

Application	Power outages	IRQ occurrences	Kernel stack utilization
Buttons	0	0	42
		≥ 10	62
	2	0	42
		≥ 10	68
Radio	0	0	58
		≥ 10	100
	2	0	62
		≥ 10	102

with Sytare requires 44 additional bytes in RAM for persistence management. The kernel variables and checkpoint images are located in NVRAM and do not impact kernel RAM occupation.

Table 4.4 shows the amount of NVRAM used by each application. The checkpoint image size is the size of a single checkpoint image. Since Sytare uses a double-buffer mechanism for checkpoint images, the total NVRAM space used solely by the checkpoint images is twice as large. Both images take less than 12% of the total NVRAM utilization. The remaining NVRAM consists of the instruction code itself, which includes the application logic that does not change from the bare-metal baseline and Sytare's code. The WSN application occupies almost 12 kB out of the available 16 kB of NVRAM from the MSP430FR5739 micro-controller, mostly due to code size. NVRAM requirements vary from an application to another because they do not use the same amount of drivers, of variables, *etc.*

Kernel stack utilization

Sytare kernel runs on its own stack, which means that part of the memory is dedicated to the kernel and is not usable from application space. In order to evaluate the amount of used kernel stack, two additional applications were used: a *button* application that counts the amount of times a button is pressed, and a *radio* application that counts the amount of received packets. The button is wired to a hardware interrupt and so is the radio packet reception. Using interrupts enables to measure scenarios with waiting bottom-halves, where the stack usage is generally more important than interrupt-less scenarios. Both applications display their counter on the LEDs.

These two applications were run with two parameters: (*i*) the amount of power outages throughout the experiment and (*ii*) the amount of times the application was interrupted by the button or the radio, excluding power outage detection. Table 4.5 shows how much kernel stack is actually used for both applications under these scenarios. The results show that interrupt handling requires few dozens of bytes at most. Given these results, in the present implementation, the amount of RAM assigned to kernel stack may be safely set to 128 bytes.

However, the nature of the application may change the stack requirements of the kernel, since using different peripherals may result in a different kernel stack utilization. For instance, the GPIO driver performs trivial operations, while the radio driver is built on top of the SPI driver and also uses the GPIO driver. As a result, there are many nested driver routines and kernel stack usage increases. In addition, interrupts add stack utilization overhead as they involve driver calls and manipulate kernel internal functions. Power outages do not substantially change kernel stack usage, as the kernel stack is reset on boot.

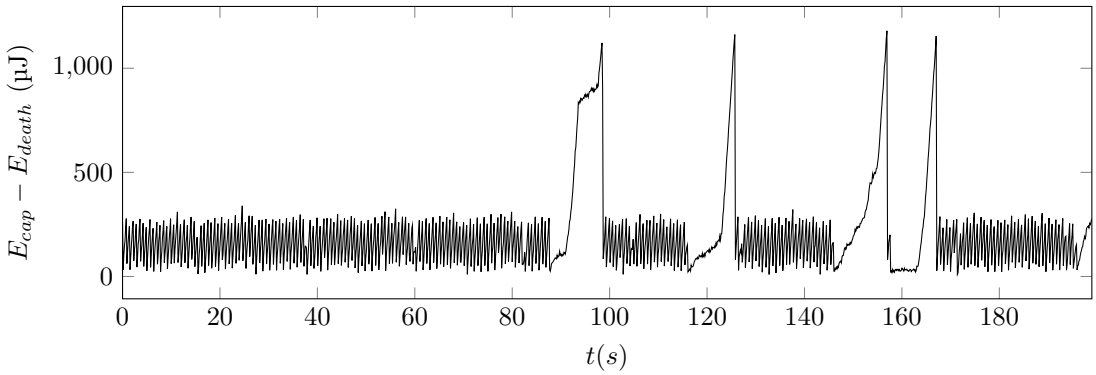


Figure 4.8: Excerpt of the available energy in the storage capacitor, as the platform harvests solar energy. Peaks above 1000 µJ are caused by an internal feature of the harvester. When the solar panel is covered, the storage capacitor is charged too slowly and the harvester initiates a cold-start boot in order to make sure that the next life-cycle will have enough energy for its demands.

Application stack utilization overhead

Application stack utilization mainly depends on the application itself. However, Sytare needs to store some data on the application stack to achieve persistence.

When switching from application stack to kernel stack on driver routine wrapper entry, thirteen 16-bit registers are pushed onto the application stack. As a result, the wrapper mechanism adds a 26-byte overhead to application stack utilization. Since a bottom-half can run when the main application is interrupted during a driver routine, and since bottom-halves also can use peripherals, then in this worst-case scenario the 26-byte overhead can be accumulated, making this overhead reach 52 bytes.

When power outage detection interrupt is generated, the micro-controller pushes two 16-bit registers onto the current stack. Hence, whenever power outage detection occurs while running application code, a 4-byte overhead will be observed in the application stack.

As a result, the typical worst-case scenario as far as application stack utilization overhead is concerned is when the main application is interrupted during a system call and power outage detection occurs when the bottom-half uses a system call. In this specific situation, overhead adds up to 56 bytes. In the experiments, 256 bytes for the application stack were enough in total, for all applications.

Energy overhead

Sytare was tested against the AEM10940 energy harvester from *e-peas* [2] company. It consists of a solar energy harvester and a power manager, and accumulated energy into a capacitor. In the experiments, a large capacitor of 330 µF was used. The embedded power manager powers on and off the MSP-EXP430FR5739 board, respectively whenever the capacitor voltage goes beyond $V_{boot} = 3.21$ V or below $V_{death} = 2.93$ V, which gives an energy budget of 313 µJ. For these applications, lower capacitor values may be used as the longest driver call, namely the radio packet sending driver routine, does not need that much energy. As discussed by the authors of Capybara [15], the storage should be large enough to run the largest atomic section. The largest atomic section is the radio packet sending system call which takes 100 µJ. The solar cell used in the experiments is the CBC-SEH-01 [3], a 32 cm^2 solar cell. A smaller cell can be used and would simply decrease the frequency of the life-cycles, as long as the delivered power is large enough to enable harvesting.

Figure 4.8 shows the amount of energy stored in the capacitor over a period of 200 seconds, under indoor light conditions, while covering the solar cell occasionally. Crescent phases correspond to periods of time when the capacitor is charging and the device under test is unconnected by the power manager. Decrescent phases correspond to periods of time when the device under test is supplied by the power manager. The capacitor is refilled within 730 milliseconds in case of usual office light conditions and

[2] `https://e-peas.com/`
[3] `https://www.ti.com/lit/ug/slau273d/slau273d.pdf`

within 10 seconds when the solar cell is covered. This scenario is thus realistic for a low-power application solely supplied by energy harvesting. Figure 4.8, albeit being an actual measurement, is shown only for hypothesis validation purposes and the measurements themselves are shown as indicative only. The measurements cannot be used as an energy trace, as discussed in Section 3.1.3, for they are dependent on the experimental conditions.

Checkpoint creation and restoration operations respectively consume 0.183 µJ and 2.115 µJ every life-cycle, leaving 310 µJ for the application to run every life-cycle. Hence, Sytare does not impede application progress as it keeps it energy budget small enough and lets the application use the bulk of the energy available.

4.7 ARMorik: Towards new Architectures with NVRAM

The observation can be made that the amount of micro-controllers equipped with non-volatile RAM is limited. The bulk of them is constituted by Texas Instruments' MSP430FR family, using FRAM as a full replacement of Flash. Aside from that specific architecture, other manufacturers and other architectures are timorous towards the usage of non-volatile RAM. This is certainly the consequence of the firms not trusting the economic viability of such a new architecture.

MSP430 has its limitations and, although people like to write portable code, the presence of power outages breaks several assumptions of programming languages such as C, hence some low-level architecture-specific code must be written at some point. It would be interesting to see how operating systems for transiently-powered systems can be written on ARM architectures, for instance. Furthermore, as discussed in Chapter 6, the MPU of ARM-Cortex micro-controllers has more features and is better suited to incremental checkpointing in comparison to that of the MSP430FR micro-controllers.

In addition, as long as the performance gap between volatile and non-volatile RAM is not fixed and as long as the non-volatile RAM has such a limited endurance, platforms are likely to embed both kinds of memory. The MSP430FR micro-controllers with the greatest amount of volatile RAM are the MP430FR604x family, with only 12 kB of volatile RAM and 64 kB non-volatile RAM. The micro-controllers with the greatest amount of non-volatile RAM embed 256 kB of non-volatile RAM but only 8 kB of volatile RAM. More complex applications would be enabled if the platform had more memory. The amount of memory also impacts persistence mechanism design for transiently-powered systems, as discussed in Chapter 6, thus motivating the need for platforms with more volatile RAM and non-volatile RAM.

In the research community, a platform was designed with an ARM Cortex-M3 micro-controller and Flash, but without non-volatile RAM [98]. Another platform, ENGAGE, embeds an ARM Cortex-M4 micro-controller, wired with a 512 kB external FRAM chip [28]. ENGAGE was not yet published at the time of design of the present platform proposal. However, the external FRAM chip used by ENGAGE has a serial SPI interface, which means that reads and writes to non-volatile RAM are even slower than usual.

This section presents ARMorik, an ARM-based alternative to existing platforms equipped with non-volatile RAM, as a subsidiary contribution of this thesis. The design includes an external parallel FRAM chip to map the non-volatile RAM into memory address space, thus mimicking the MSP430FR micro-controllers, in order to keep the performance of FRAM without the overhead of a serial bus.

4.7.1 Hardware Design

The key point of ARMorik is that the non-volatile RAM chip is accessed through a parallel bus. This means that there is a pin for every single bit of address and every single bit of data, in addition to the control pins and power pins.

External NVRAM From the memory's perspective, there is a variety of non-volatile RAM chips that expose a parallel bus, so the choice of the memory chip is only guided by memory capacity and power-related concerns. The Cypress FM28V202A [4] 256 kB FRAM chip was chosen, for its interface similar to that of an SRAM. The constructor indicates that the memory consumes 7 mA in active mode,

[4] `https://www.cypress.com/file/136441/download`

which is high when considering that the MSP430FR micro-controllers consume less than 2 mA including the processor *and* the built-in FRAM. However, a design with an external memory will always be less energy-efficient than a design with every component belonging to the same chip. The ability of the ARMorik platform to demonstrate the interest in transiently-powered systems with more memory and other architectures than MSP430 is not impacted though.

Micro-controller The bottleneck is the micro-controller. Indeed, it is not acceptable to use GPIOs to communicate with the external memory chip and the reason is twofold. First, GPIOs require a certain update latency and cannot fully take advantage of the low latency of memories. For instance, the FM28V202A chip has a latency faster than 100 ns, meaning that the GPIOs must operate at least at 10 MHz. Bit banging, even using a DMA to accelerate the process, is not likely to achieve such frequencies with low-power and low-frequency micro-controllers. Furthermore, there are dozens pins to update, driven by several control registers, which must be carefully performed to avoid bad intermediate states. Second, in order to read or write data from and to the external memory using a GPIO-based bus, the software would have to set the appropriate control registers each time. The external memory chip can thus not be mapped in memory using pins as GPIOs and hence, one cannot simply access the external memory using a single `load` or `store` instruction, which would both generate time overhead and force the software to access the memory through a dedicated API. It is thus necessary that the micro-controller embeds a dedicated logic for external memories. This last property is dependent on the manufacturers. Microchip's ATSAMS70Q19 [5] micro-controller, based on an ARM Cortex-M7, was chosen for its ARM architecture, its Static Memory Controller (SMC) that suits the external memory chip and its parallel address and data buses multiplexed to the pins of the chip. It also has 256 kB SRAM and 512 kB Flash memory. The Low-profile Quad Flat Package (LQFP), a popular surface-mounted integrated circuit package, was elected for its ease of hand-soldering compared to the ball grid array alternatives even though the latter were more commercially available.

External peripherals Besides the built-in peripherals of the micro-controller, the ARMorik platform embeds a handful of toy yet useful peripherals. First, the board has a socket for Texas Instruments' CC2500EMK radio module for compatibility with the MSP430-EXPFR5739 board. Hence, the original driver of the CC2500 module for Sytare can be used almost as is, and the ARMorik platform can even communicate with the MSP430-EXPFR5739 over radio. ARMorik also comprises the ADT7310 [6] digital temperature sensor from Analog Devices. Both the radio module and the temperature sensor use the SPI bus, which is an interesting study case for driver design on a bus with several slaves. Indeed, works that support peripherals (P2), including Sytare as presented here, show applications that use a few peripherals that do not collide with one another. In reality, peripherals are multiplexed and there are often mutually-exclusive configurations. Sytare supports peripheral re-configuration at run-time while many existing works set this concern aside and consider that either the reset-state of the peripherals are sufficient or that a routine written by the application developer is called upon reboot to reset the peripherals to a fixed, known state. Building a platform that has mutually-exclusive configurations is both closer to real systems and also highlights the need of a true peripheral restoration system rather than a hardcoded state setting. ARMorik also embeds a photodiode as an analog peripheral to be sensed using an ADC. Any high-resistance photodiode can fit and the one used in the prototype is the NSL-4962 from Advanced Photonix. Finally, there are eight LEDs apart from the power- and debug-related LEDs.

Power supply The different hardware components of the board operate at different supply voltage ranges, but their intersection is non-null. The temperature sensor requires at least 2.7 V, the radio module 1.8 V and the minimal supply voltage of the micro-controller depends on the required internal peripherals. For instance, the USB module of the micro-controller, albeit not used in ARMorik, requires at least 3.0 V and the Digital-to-Analog Converter controller at least 2.5 V. Also, the resistors in series with the LEDs and the photodiode must be adapted to the supply voltage. There is no other constraint on the minimal supply voltage. Hence, the board proposes a supply module with either a fixed 2.2 V supply or a variable supply between 2.0 V and 3.3 V configurable using a potentiometer. Those supply

[5] `https://ww1.microchip.com/downloads/en/DeviceDoc/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527D.pdf`
[6] `https://www.analog.com/media/en/technical-documentation/data-sheets/ADT7310.pdf`

voltages are provided through an external USB cable, but the low power consumption of the platform allows it to be supplied even with the motherboard of a computer, as opposed to single-board computers that require a 3 A supply.

Circuit design Figure 4.9 shows a photograph and the board design of the ARMorik platform. The dashed lines isolate the power supply and the programming interface from the actual platform proposal. The USBALIM jumper may be unplugged to allow another power supply, such as an energy harvester. The entire schematics are available online. [7]

Energy consumption The energy consumption of the ARMorik platform was not studied. It is considered irrelevant, for the non-volatile RAM is external to the micro-controller and thus cannot be as optimized as an all-in-one micro-controller such as the MSP430FR ones. However, ARMorik is a proof-of-concept platform that encourages other types of architectures to benefit from the properties of non-volatile RAM.

4.7.2 Software

Any ARMv7-M Thumb-2 code can execute on the platform. More specifically to this thesis, Sytare was ported onto the ARMorik platform, thus named sytARM. [8]

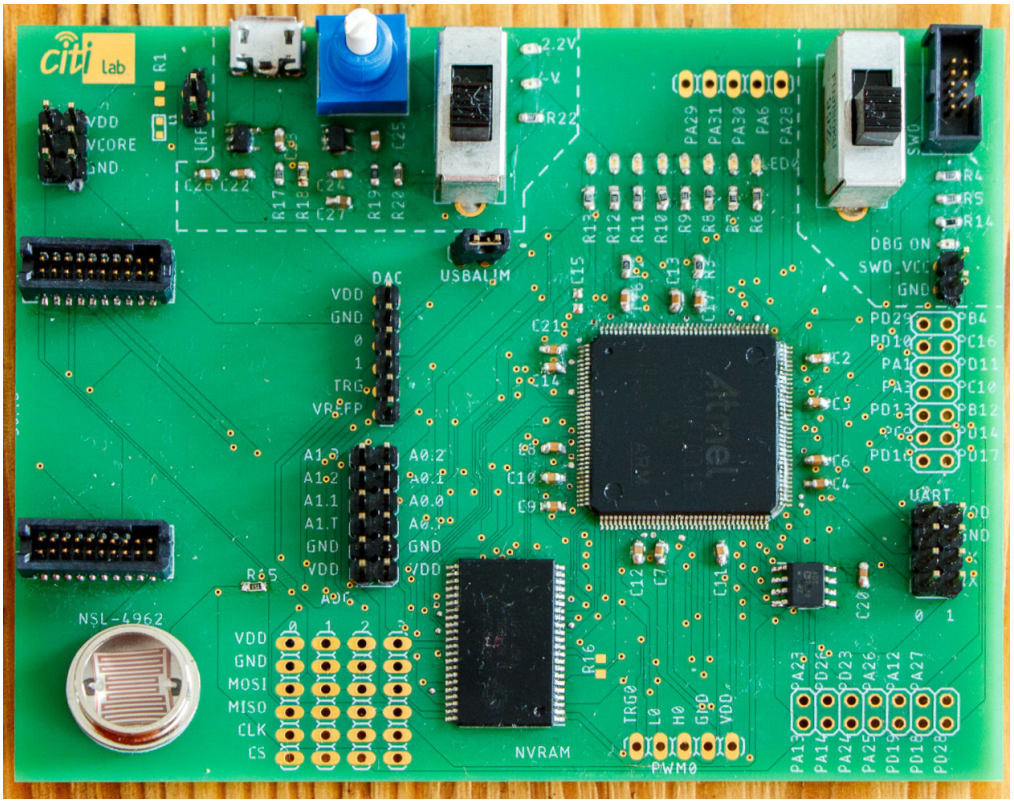
Programming The board provides a Serial Wire Debug (SWD) port on which a programmer can be wired when needed. The programming feature was tested with the Atmel-ICE programmer. Within the context of sytARM, the programming is twofold. A first pass populates the non-volatile RAM with a valid checkpoint image corresponding to the application and peripheral states of the application at the beginning of the `main` function. This first pass is not part of the application software for it is automatically performed before the actual application is flashed. The second pass is the transfer of the actual application to the Flash memory of the micro-controller and from that point onward, the platform is ready to use. Again, the application solely consists of the binary image that resides in Flash memory and of the checkpoint images that reside in non-volatile RAM. Note that the Flash memory is only used as a read-only memory in sytARM, to store instructions and relevant read-only data. Populating the non-volatile RAM chip is automatically performed as a service provided by sytARM toolchain. The main discrepancy between sytARM and Sytare is that a valid checkpoint is provided before the application even starts. This feature simplifies the state-machine from Figure 4.5, thus removing the `Initialize` state. This service does not leverage any ARM-related mechanism and can thus be implemented on MSP430 as well. It simply arises from the "from scratch" approach that was free of legacy code whereas Sytare emerged from successive development sprints.

Broadened application spectrum The 256 kB of volatile RAM allow the execution of more complex applications. For illustrative purposes, MicroPython [9], an implementation of Python 3.x for constrained micro-controllers, has been easily ported onto ARMorik. A MicroPython hello-world application, with its dependencies regarding MicroPython, takes 415.4 kB of instruction memory and that particular implementation required 16 kB of heap. It could not be executed on any MSP430FR micro-controller equipped with non-volatile RAM. In comparison, the LED hello-world of sytARM, similar to the LEDs application of Sytare, requires 4.3 kB of instruction memory and no heap.

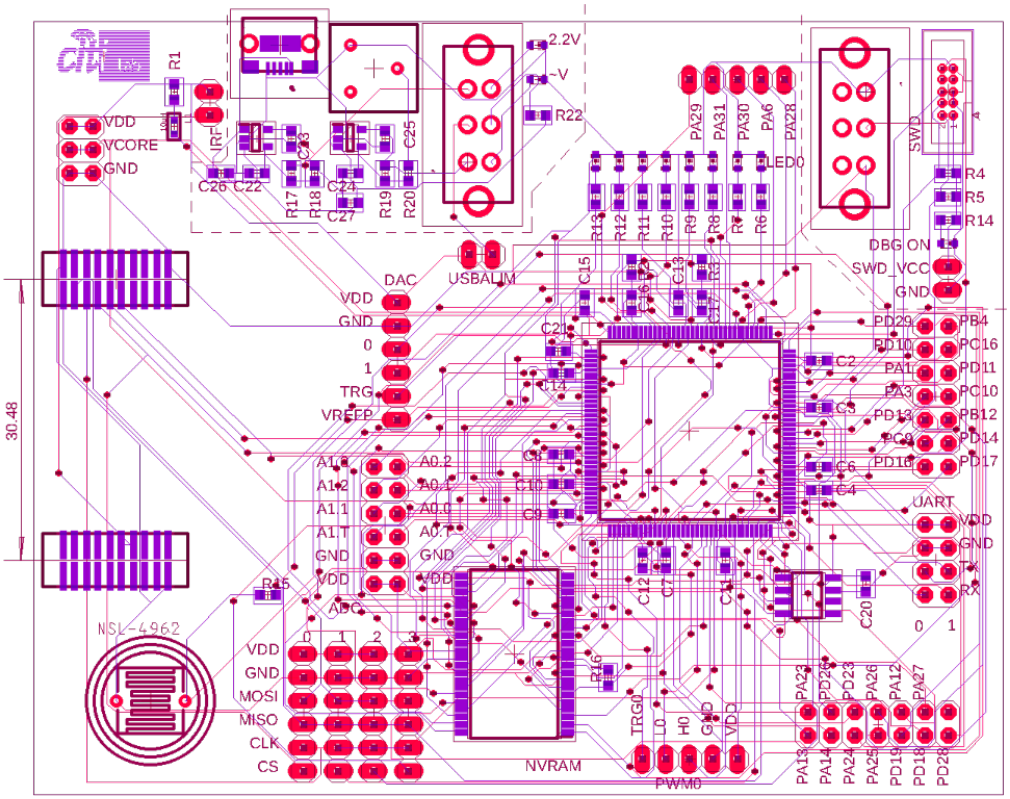
4.7.3 ARM-related mechanisms

SuperVisor Call The Thumb-2 instruction set provides a SuperVisor Call (SVC) instruction that immediately provokes an interrupt. That instruction uses an immediate value as argument and its semantics are left to the software developer. In sytARM, `svc #0` invokes the restoration, `svc #1` is a

[7] `https://github.com/gberthou/armorik`
[8] `https://gitlab.inria.fr/gabertho/sytarm`
[9] `https://github.com/micropython/micropython`



(a) Photograph of the ARMorik board.



(b) Design of the ARMorik board.

Figure 4.9: Photograph and board design of ARMorik.

wrapper entry and `svc #2` a wrapper exit. This makes the kernel code more homogeneous for the SVC execution mode is that of interrupt mode. In the implementation of Sytare on MSP430, the code has to manually fake the interrupt context while sytARM leverages an ARM instruction to execute in a real interrupt context. In addition, the interrupt system of the Application Binary Interface of the Cortex-M architectures alleviates stack management whereas it must be done by hand under the MSP430X architecture.

Memory Protection Unit As discussed in Chapter 6, the Cortex-M7 MPU provides a Memory Protection Unit suitable for memory modification detection at run-time, allowing incremental checkpointing.

Platforms with a greater amount of memory, such as ARMorik, enable to design applications that overcome the simple sensor use-case. They also show the limitations and the non-scalability of some checkpointing techniques currently used for transiently-powered systems with no more than a few kilobytes of volatile memory, as discussed in Chapter 6. Provided that manufacturers will integrate non-volatile RAM to micro-controllers with more memory, the research in transiently-powered systems can be extended beyond the technological limitations of today's platforms.

4.8 Conclusion

Sytare is a lightweight kernel for transiently-powered systems. It brings a real and efficient answer to problems P1, P2 and P3 altogether. To this extent, Sytare was the first kernel to support peripheral state persistence and consistency alongside application state persistence. It also supports user-defined interrupt handlers, making the typical Sytare application closer to a bare-metal baseline, while other state-of-the-art approaches still do not. Bare-metal applications can run on top of Sytare almost as is. The only major modification is that peripheral accesses must be encapsulated into an API, as often encountered in bare-metal development anyway, using Sytare wrappers as an additional layer between application and drivers. The drivers must also implement a restoration routine and a save routine in order to benefit from Sytare's peripheral persistence mechanisms. An optional routine called upon interrupt occurrence may be required if the driver needs to react to some interrupt line, so that the application-side of the interrupt handler can also benefit from state persistence.

While application state checkpointing is not new, Sytare combines classical checkpointing of the application with incremental checkpointing of the peripherals, which was not done before. The peripherals are saved on-the-go, for any driver routine that alters the state of a peripheral notifies Sytare to partially save the newly acquired state. The peripheral-related data to be saved are internal states that enable the drivers to restore peripherals to the same states that were incrementally saved. The so-called driver contexts, that encompass enough data to restore the state of a peripheral, do not necessarily correspond to actual memory-mapped registers. This is less memory-consuming and also allows the usage of peripherals accessed through a communication bus, such as SPI or I^2C. This design choice is important since the energy storage is likely not to afford to dump all the control registers of peripherals accessed through a serial bus, while an imminent power outage threatens the system. Besides, the support for atomic regions, *i.e.*, portions of code that must be consistently run in one life-cycle or retried from the beginning, enables the application to use complex peripherals. Timers, serial interfaces, Analog-to-Digital Converter and radio transceivers are examples of non-trivial peripherals that are guaranteed to correctly run under intermittent power. Actually, the mechanisms of Sytare are generic and hence, any peripheral can be made compatible with intermittent power, as long as the energy storage enables their usage.

A first implementation of Sytare, on the MSP-EXP430FR5739 board from Texas Instruments which includes 16 kB of FRAM together with traditional RAM, shows that the proposal is realistic and suits transiently-powered systems, for its performace impact is reasonable. Applications running on top of Sytare may reach more than 90 % of their temporal performance whith life-cycles shorter than 50 milliseconds. In addition, those applications may run with non-null forward progress as long as the platform is continuously powered on more than 5 milliseconds before power fails. In the case of atomic code sections, the concerned life-cycles must be at least as long as the given atomic code section. It is important to note that the overhead of the persistence mechanisms introduced by Sytare is low, in comparison to the improvement brought by substantially less constraining life-cycle durations. The time overhead due

Sytare wrapping driver calls is quantified and the experiments show that this overhead is less than one percent of sending a radio packet for instance. More generally, applications achieve more than 97 % of their temporal performance when continuously powered, meaning that the wrappers have a low overall overhead, even when the application is peripheral-intensive.

A later implementation on the ARMorik platform, equipped with an ARM Cortex-M7 micro-controller and an external parallel FRAM, consolidates Sytare's suitability for transiently-powered systems and opens the way towards new energy-efficient systems using low-power ARM platforms.

The core contribution of this thesis regarding Sytare is the interrupt support to overcome P2.2. The experiment consolidation, achieved by developing an automated testbench for yield measurements, was also a key task that contributed to the reproducibility and the confidence in the results. The end-to-end design and development of the ARMorik platform and its software as a demonstrator for Sytare and a prospective vision of future features for transiently-powered systems is also brought by this work. Finally, maintaining the software of Sytare was a long engineering task that was yet necessary to sort issues out and increase correctness with regard to the problematics related to intermittent power.

An improved version of Sytare could perform smarter checkpoints, where portions of memory are selected if they were modified between two consecutive checkpoints. This is studied in detail in Chapter 6.

Applications that run under intermittent conditions must face issue P3 at some point and thus, might be exposed to retrying code sections. If they could evaluate the energy requirements of all atomic sections, they, or the kernel, might be able to anticipate power outages and determine whenever trying an atomic section is doomed to fail. A model that includes peripheral state analysis, as well as a cycle-accurate simulator for transiently-powered systems are depicted in Chapter 5. They may serve as a solid basis for energy requirements evaluation and any kernel mechanism that would decide whether to start a given atomic section, considering the remaining energy.

Also, if the non-volatile RAM technologies tend to be more energy efficient and thus have closer performance to volatile RAM technologies, it becomes worthwhile to investigate the memory placement of variables, including application variables. More specifically, it may be interesting to allow the application developer to allocate variables directly in non-volatile RAM. This would expose the entire system to issue P4 and require a thorough support of the identificated issues, *i.e.*, P1, P2, P3 and P4, by the kernel.

Chapter 5

Energy Model of Intermittent Systems and Measurements

The static checkpointing systems discussed in Chapter 3 over-provision the application with checkpoints so that the code sections between consecutive checkpoints are *likely* not to exceed the capabilities of the energy storage. The just-in-time approaches discussed in Chapter 3 and Chapter 4 reduce the amount of checkpoints per life-cycle to the minimal amount of one checkpoint. However, in both checkpointing models, the timeliness and atomicity constraints, namely P3 as defined in Chapter 2, require specific portions of code to run in a consistent time window, and to be retried from the beginning if the corresponding operations were interrupted. In the latter case, the application wastes energy trying to run the code section with timeliness requirements once or multiple times, for the only try that enables forward progress is the last one, *i.e.*, the try that can complete before power fails.

Today's system layers that support intermittent power lack insight of the energy requirements of portions of code. Knowledge about these figures would help, either at compile- or at run-time, the system to make clever decisions, for instance. Being able to tell whether a code section with timeliness constraints may complete within the remaining energy of the current life-cycle enables further optimizations. The system could schedule another, less consuming application task, assuming that the application is divided into tasks. Or the system could simply shutdown early, should the power manager be able to provide such feature, in order for the energy storage to reach the boot energy threshold faster and thus, decrease the off-times. Here, the assumption is made that the system layer provides a reliable, flawless checkpointing mechanism. The knowledge about energy consumption is considered an indication for optimization purposes and does not dismiss the system layer from checkpointing. While the present chapter discusses what resembles an average-case scenario, more comprehensive and pessimistic considerations such as the worst-case energy consumption are totally separate concerns.

This chapter offers an energy-related model of embedded software that takes peripherals into account and echoes a former publication [99]. That model is implemented into a simulator for transiently-powered systems, capable of accurately estimating the energy consumption of any piece of code, whether it involves peripherals or not. The simulator is designed to run unaltered binary images for Texas Instruments' MSP430FR5739, however its modularity enables to extend its usage to any other Instruction Set Architecture (ISA) with low development effort. In order for the simulator to yield adequate results, the model must be populated with actual measurements. A cheap and simple, yet efficient energy measurement device is described in this chapter for that purpose.

Section 5.1 presents the state of the art on energy models for peripherals, software and power supplies. Section 5.2 evokes existing methodologies for energy consumption measurement and Section 5.3 existing simulators that target energy consumption estimation. A new model for software and peripherals is presented in Section 5.4. This model leverages an abstraction of peripherals which is built from the *driver API* and not from the peripheral data-sheet. Section 5.5 proposes a methodology for low-cost energy measurement within adequate power ranges for low-power platforms. All peripheral driver calls and peripheral power states were precisely measured this way. Section 5.6 exposes a simulator for transiently-powered systems using the proposed energy model and energy measurement methodology to populate

the model. The simulator operates at cycle-level for software and abstracts driver calls using the model depicted hereafter. Finally, Section 5.7 shows the performance of the simulator with respect to real-life measurements.

5.1 State-of-the-art Energy Models

Three main approaches can be used to evaluate the energy consumption of embedded systems: theoretical analysis, simulation and hardware measurement. To yield realistic results, simulation must be calibrated with real measurements. One issue is to establish an energy model of all hardware components: CPU, memory and peripherals.

Profiling energy consumption on real hardware has already been studied, but either the peripherals were not specifically studied [100] or the targeted platforms were high-end systems [101].

5.1.1 Hardware Models

The EPIC model [23] tackles the aspect that makes common approaches ineffective in their energy consumption prediction. Many works want to analyze the energy consumption of transiently-powered systems in which energy storage is directly wired to the platform, *i.e.*, the harvest-use model discussed in Section 2.1.3. They often tend to forget that, as a consequence of design choices, the supply voltage of such platforms varies over time and thus, their energy consumption varies. Running the same instruction twice at different supply voltages result in a different energy consumption. Indeed, a change in supply voltage provokes a change in both the current consumption and the actual clock frequency. When the supply voltage decreases, the clock frequency slightly decreases, which means that a given operation would take more time than at higher supply voltage. However, the overall power consumption substantially decreases for the electronic components waste smaller power in Joule effect. At the end of the day, decreasing supply voltage incurs longer execution times but smaller energy requirements. While EPIC is right about changes in energy demands, it assumes that the energy storage directly supplies the platform, *i.e.*, without voltage regulator, which is the opposite assumption of this chapter, as discussed in Section 2.1.3. Nonetheless, EPIC is a key work that should broaden minds about energy calculation: it is *not* as simple as multiplying voltage, current and time altogether at a large scale without inspecting local variations. Despite working on different assumptions, EPIC lays the groundwork for the reasoning that motivates this very work: executing the same piece of code does not always take the same amount of energy.

5.1.2 Software Models

The EH Model [102] grasps the energy consumption of checkpointing mechanisms, notably the energy of creating and restoring checkpoints. The EH Model supports a variety of existing static and just-in-time checkpoint mechanisms. It is an important tool for design space exploration as the authors state. However, it reasons in terms of energy budgets and cannot be used to model the energy consumption of a given, actual piece of code. The electronic aspects are also left apart as a separate concern. The physical quantities that are analyzed are energy and time, and the EH model does not delve deep into the analysis of instantaneous current consumption.

Other works propose power state tracking [103, 104, 105, 106]. Micro-controllers and peripherals are assumed to follow a finite state automaton, where a fixed, constant power consumption is associated to each state. Those states are thus called power states.

SysWCEC [106] proposes a model where the micro-controller may have several power modes while the peripherals may only have two states, on and off. It is focused on the worst case through static analysis and exhaustive path enumeration to handle all cases in a multi-task model. SysWCEC computes all combinatory power states, which is exponential in the number of components taken into account, and thus hardly scalable to actual applications. In reality, peripherals are often more complex than simple on and off state machines. The main difference with the work depicted in the present chapter is that SysWCEC targets Worst-Case Energy Consumption (WCEC), while the present work focuses on an average case. While SysWCEC is a smart WCEC approach that enables to reason about dimensioning platforms or emphasizing harsh run-time conditions, it is still a worst-case approach and it is thus expected

to yield significantly higher energy consumption results with respect to the average case targeted by this work.

Cherifi *et al.* [105] propose to define power states at API-level. The states are extracted automatically by analyzing, at run-time, changes in the instantaneous power consumption, in correlation with the functions called by the software. They use an expensive power tracer with a minimal latency of 20 µs between consecutive samples. The measurements are back-annotated into the generated automata. Their work leverages only high current gaps, which are observed with peripherals such as radio chips, from several milliamperes to dozens milliamperes, but does not apply to less consuming peripherals. Cherifi *et al.* consider stable power states, but shorter states and transitions between states are not modeled. However, their model provides a sound baseline for simulators such as the one exposed in this chapter. It furthermore shows that power-annotated finite state machines are a promising way to be investigated.

Quanto [103] and Powertrace [104] are energy profilers for embedded systems that operate on a single device as well as over networks. Quanto's power state model is populated using measurements performed using the iCount [107] energy meter. An operating system layer dynamically keeps track of changes in power states. This is achieved by modifying driver code: a driver routine that changes the power state of a peripheral must notify the system layer of the change. The energy consumption may be cut down into several energy sinks, each energy sink being associated with a software-initiated activity. This way, Quanto and Powertace are able to provide a detailed energy profile at run-time, befitted to the needs of the application developer. These tags may also be transmitted through network packet tags, hence extending the profiling methodology to an entire operating network.

However, these works only reason about states, but all of them assume that transitions between states are purely symbolical and thus do not study transitions. On the contrary, this chapter extends power state tracking and adds semantics to the transitions in order to enhance the energy model. Reasoning at driver API level, transitions may be driver calls, or sometimes software-initiated automated hardware transitions in the case of asynchronous calls. This chapter proposes to add a duration and an energy cost to every transition, *i.e.*, to every driver call.

5.1.3 Power Supply Models

First, the most natural power supply is the constant and continuous voltage, *e.g.*, the power supply of a desktop computer. In the context of transiently-powered systems, the scheme becomes more complex for the power supply loses its property of continuity. In addition, power supplies for energy harvesting devices are not unified and there exist fundamental differences from one another.

As seen in Section 2.1.3, the platforms may or may not be directly wired to the transient energy storage. In the case where the platforms are directly wired to the transient energy storage, a supplemental measurement and analysis of the instantaneous supply voltage is crucial in order to provide a correct model and a correct estimation of the energy consumption [23]. On the other hand, when a voltage regulator is wired between the energy storage and the platform, the resulting supply voltage is nearly perfectly constant, provided that the energy storage voltage meets the voltage regulator requirements. In other terms, the usage of a voltage regulator enables to model the voltage supply as a square function: its output voltage is either near 0 V or near its nominal voltage. Transitions between those two states may or may not be considered. They may be assumed to be short in terms of time and not significant in terms of ratio between transition times and stabilized times. The model depicted in this chapter opts for a power supply with a voltage regulator, providing a constant supply voltage to the platform under test. This choice is the direct consequence of the discussion from Section 5.1.1, for supply voltage fluctuations uselessly add complexity to the energy model and can disturb electronic components such as clocks and oscillators.

Regardless of the energy model design, a certain amount of energy-related physical quantities are expected to be involved. In order to yield an estimation of the energy consumption of some portion of code, the model needs to be populated with actual values. Such values can be either obtained by scanning the data-sheets of the electronic components, or by measuring the required quantities directly on the real hardware. Next section addresses this problematic and reviews ways of measuring energy consumption from the literature.

5.2 Energy Measurement for Low-Power Systems

Energy consumption measurement for low-power embedded devices is not an easy task. The main issues are cost, design complexity, dynamic range and accuracy. There are numerous ways to measure energy. State-of-the-art approaches are reviewed hereafter.

5.2.1 Temporal Integration of Power

A first approach is to measure and integrate instantaneous power over time. Di Nisio *et al.* [108] identify two categories of circuits for current measurement. The simplest one uses a shunt resistor on the high-end of the platform under test and the other one uses a feedback circuit that enables higher resistor values. The main issue with the circuits using shunt a resistor is that the circuit subtracts from the supply voltage a voltage that is linear to the current drawn by the platform. When the platform under test draws a high current, *e.g.*, when sending a radio packet, the voltage drop due to the usage of a shunt resistor might be significant. The worst-case scenario would be that the voltage drop is so high that the voltage drops below the minimal working supply voltage of the platform, thus preventing any correct functioning every time the platform performs that power-consuming operation. But even without reaching such an extreme extent, variations in supply voltage change the power consumption of the platform, as highlit by the EPIC model [23]. A variation of the traditional shunt resistor circuit is to use a smaller shunt resistor and to amplify the voltage of the shunt resistor using an operational amplifier [109]. A drawback of the amplified variation of the shunt resistor circuit is that noise is amplified and that the operational amplifier introduces an offset. The actual gain of the operational amplifier may also differ from the nominal value from the data-sheet, but remains a constant value when used in a suitable signal frequency range. However, it is possible to calibrate out these values using reference current sinks. The other type of circuit is the one proposed by Di Nisio *et al.* It derives from the widespread voltage-controlled current sink based on an operational amplifier and resembles the design of a feedback ammeter. By adding a resistor in series with the platform under test, the resistor behaves like a shunt resistor. However, their proposal takes advantage of the near short-circuit of the operational amplifier to bypass the resistor and provide a stable supply to the platform under test, thus tackling the main issue of using shunt resistors. In addition, their technique enables the usage of higher resistor values, so that the measured values are less prone to noise. On the other hand, their circuit requires that the supply of the measurement equipment to be greater than the supply voltage of the platform, including the voltage of the potentially big resistor. This might result in the need of an external power supply, which is not necessarily desirable.

Although the proposal of Di Nisio *et al.* is appealing due to the stability of its supply, this chapter uses the simpler amplified shunt resistor circuit. It must be noted that the sole purpose of the energy measurements, in this chapter, is to populate the model with exploitable values, while keeping in mind that a simulator can be built out of it. While the quality of the measurements *de facto* impacts the accuracy of the simulation, the model itself is nonetheless totally independent of numerical values and stands given any energy measurement methodology.

EMPIOT [109] proposes an accurate, low-cost power measurement platform and targets wireless devices. It aims the same kind of platforms as targeted here, although EMPIOT considers that the platform may draw up to 400 mA whereas in transiently-powered systems it is more realistic to consider 30 mA instead. Hence, the platform proposed in this chapter is similar to EMPIOT, but is more suited to the specific dynamic range and resolution needed on low-power devices. In addition to targeting platforms that can draw one order of magnitude higher more current, EMPIOT also has a relatively large resolution of 100 µA. The measurement platform described in this chapter proposes to measure currents up to 26 mA with a 6 µA resolution.

Eprof [101] is an energy profiler that keeps track of the energy consumption and associates it to a software location, despite asynchronous operations. Eprof repeatedly interrupts the micro-processor to capture the stack trace of the running thread. The evaluation of the energy consumption of asynchronous driver calls takes advantage of the existing concept of device requests, provided by operating systems such as Linux. Hence, eprof targets platforms with more capabilities than current transiently-powered systems.

5.2.2 Other Approaches

A completely different approach consists in using a coulomb counter that integrates the current over time directly in hardware. Hahm and Adler [110] use that approach. However, the measurement platform proposed for this chapter does not use a coulomb counter because of the known imprecision in the measurement of the capacitance of a capacitor. In addition, design simplicity is one of the main concerns of the measurement platform.

PEEK [111] provides a physical platform for energy measurements. It leverages a current mirror, to replicate the current drawn by the platform without disturbance. Two capacitors alternately charge and discharge at a rate that is proportional to the current drawn by the platform under test.

The energy meter iCount [107] powers a platform by storing some energy into an inductor. The platform under test draws its energy from the inductor, until the energy level of the inductor drops below the minimal value that enables the platform to be powered. The energy budget of this usage of the inductor is thus constant and iCount counts the amount of times the inductor was depleted to yield its estimation of the energy consumption. The energy meter iCount further provides a constant supply voltage to the platform under test by using a boost voltage regulator. Its energetic resolution is around 0.5 µJ and its highest sampling frequency is 66 kHz.

5.3 State-of-the-art Simulators

Platform simulators that target energy consumption estimation already exist.

Bouhadiba *et al.* propose a simulation at transaction-level [112]. While that solution identifies the power states of the platforms, it rather simulates elapsed time and does not simulate the peripheral operations themselves, nor the execution of the instructions on the micro-controller.

PowerTOSSIM [113] is a simulation environment of TinyOS applications. PowerTOSSIM tracks the evolution of the power state of all peripherals and replaces the software basic blocks by an estimation of their execution times. Transitions in the power state automata are tracked using a trace of driver calls. The software model of TOSSIM, on which PowerTOSSIM is based, requires that the application is re-compiled specifically for the simulator. Similarly to the work of Bouhadiba *et al.*, the micro-controller instructions are not executed, hence TOSSIM cannot yield a precise value of the computational intensive parts of software, such as processing sensed data. PowerTOSSIM bypasses that limitation by counting the amount of times a basic block was executed, having formerly instrumented every basic block in TOSSIM, and by estimating the time required to execute every basic block. The instructions are thus not actually simulated, and the result is still an estimate, albeit closer to reality than what TOSSIM could have yielded. In their experiments, the authors noted a discrepancy of 1.5% to 33.0% cycle count for software execution. The whole simulation could thus be improved with a cycle-accurate simulator that simulates every instruction on an unmodified binary.

Another method consists in modeling the platform on a Field-Programmable Gate Array (FPGA) platform with a dedicated monitor that analyzes memory accesses and elapsed cycles [114]. That method leverages memory energy models in order to yield energy estimations, but the monitor does not record changes of peripheral power state.

5.4 Energy Model of Intermittent Systems at Software Level

This section defines the energy model as a contribution to accurately describe the energetic behavior of transiently-powered systems. The present model extends the concept of power state tracking. Traditionally, peripheral operations initiate transitions between power states and those transitions are purely symbolical, as if such transitions took no time nor energy from the available energy budget. This chapter proposes a novel model that gives meaning to transitions between power states in order to improve model fidelity.

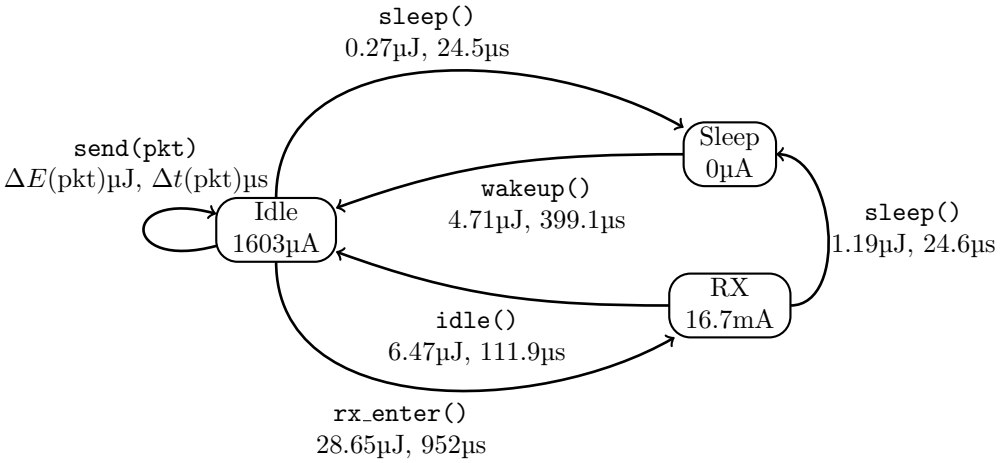


Figure 5.1: Driver state machine of the driver for CC2500EMK radio daughter board. State is `Idle` after driver initialization. The values were measured as shown in Section 5.5.

5.4.1 Peripheral Model

The behavior of any peripheral can be modeled using a finite state machine. It can be as simple as a two-state on/off machine, or it can be much more complex as, for instance, a radio chip state machine with several internal states. These state machines are usually documented in the hardware data-sheet. In general, it is not necessary to model peripherals at a fine grain. Only the state machine of the driver API itself, which is either a subset of the actual peripheral state machine or a higher-level finite state machine, may be considered. The proposal of this chapter is to use these *driver state machines* to model the power consumption of the peripherals. A typical example of such a driver state machine is the one of the radio chip used in this study, that is Texas Instruments' CC2500EMK daughter board, represented in Figure 5.1. The state automaton from Figure 5.1 illustrates the difference between the driver API state machine and the data-sheet state machine. Indeed, the data-sheet exposes all the hardware states of the radio chip whereas the API only exposes higher level services and thus simplifies the state machine. This is desirable for it alleviates the state machines. The resulting driver state machine contains only three states: `Idle`, `Sleep` and `RX`. There is no dedicated state for data transmission. Indeed, the driver call for transmission starts from the `Idle` state, temporarily switches to the hardware transmission state, then switches back to `Idle` state when the radio packet is sent. Hence, from the driver's perspective, the state did not change.

In a driver state machine, each state is considered to have its own power state, *i.e.*, current consumption, assumed to be constant until the peripheral state changes. A strong assumption of the model brought by this chapter is that peripheral power state does not change on its own but always on behalf of the driver routines. In other terms, the present model solely accounts for synchronous APIs. This is a simplifying assumption but, as supported by the results exposed in Section 5.7, it is sufficient to obtain a fair estimation of reality. Strictly speaking, asynchronous calls may allow peripherals to change without the explicit intervention of software code. Timers are examples of such peripherals, for their counter registers do not require any software intervention to update their values, but the counter evolution does not modify the timer power consumption. In order to handle asynchronous calls, the present model assumes that an interrupt is always raised when a peripheral gets into a state that changes its power consumption on its own, so that the interrupt handler becomes one of the transitions of the driver state machine.

In a driver state machine, the transitions have specific semantics. First, all transitions correspond to a driver call. This directly derives from the assumption that driver calls are synchronous. In addition, each driver call, *i.e.*, transition between driver states, has a given cost in execution time and energy. The arguments passed to the driver routines might have an influence on these metrics. Sending a radio packet is a typical example of such a parameterized driver routine because the duration and the energy consumption of the call increase along with the increased size of the packet, as illustrated by `send(pkt)` in Figure 5.1. However, while conducting experiments, most of the durations and energy values associated

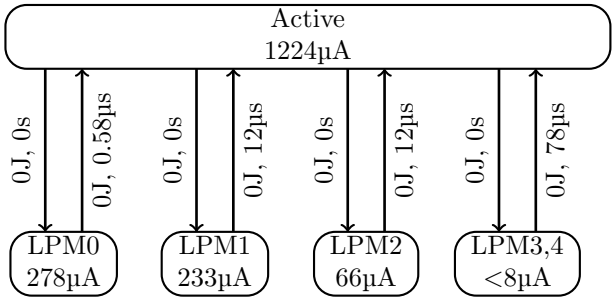


Figure 5.2: Power state machine of the MSP430FR5739 micro-controller. Initial state is `Active` mode. The values of current consumption were measured as shown in Section 5.5. The transitions from `Active` state to `LPMx` are performed by a single instruction, hence the operation lasts a few cycles and is accounted for no additional energy consumption. Conversely, the durations of the transitions from `LPMx` states to `Active` mode vary between 0.58 µs and 120 µs for the MSP430FR5739 micro-controller, depending on the low-power mode.

to driver calls did not depend on the parameters. The last statement however strongly depends on the driver API comprehensiveness. For instance, a driver API that would let the application finely tune the configuration of the peripherals might have a `config` routine with many parameters that would totally change the behavior of the `config` routine, *e.g.*, using a `switch` statement. It may also be noted that the same driver call may consume a different amount of energy, depending on the current state of the peripheral. This is the case of the `sleep` driver call from Figure 5.1, that consumes 0.27 µJ when issued from `Idle` state and 1.19 µJ when issued from `RX` mode. The duration is the same for the code of `sleep` is unique. The difference in energy consumption is solely imputable to the fact that the starting states have different current consumptions.

The states carry information about the peripheral power state. Transitions between states carry information about consumed energy and elapsed time for each transition. Transition information is a major aspect of this contribution, for it is the key information usually ignored by state-of-the-art power state tracking models. Every peripheral operation invocation broadens the gap between the energy estimation of such models and the real conditions, for elapsed time and energy consumed during the execution of the peripheral operation vanish from the overall balance. This motivates the need for a transition model in addition to the existing power state models, which is the main purpose of the present model. To populate this model, numerical values have been obtained by profiling the drivers of Sytare using the methodology described in Section 5.5.

The micro-controller itself also has a power state machine. It has several operating modes: an active mode enabling software to make progress, and the low-power modes disabling several CPU components. All low-power modes share the property that no instruction can be executed. Hence, the semantics of the transitions slightly differ from the transitions of the driver state machine. The micro-controller executes a specific instruction to enter a low-power mode and only the occurrence of an interrupt may achieve the transition back to the active mode. The power state automaton of the micro-controller is shown in Figure 5.2. Transitions from low-power modes to active mode involve re-enabling internal clock, thus it takes a certain amount of time depending on the clocks to re-enable.

Considering separate state machines instead of explicitly enumerating all power state combinations at platform-level, in addition to using a high-level driver API, makes the model proposal of this chapter realistic and scalable to any embedded system.

5.4.2 Software Model

In general, the overall power consumption of a board depends on the power state of the entire platform. This encompasses not only the direct consequences of the piece of software actually being executed, but also the consequences of formerly executed pieces of software, as well as possible asynchronous operations. For instance, the same portion of code may consume more or less energy while an LED is on or off. It may also change without any software intervention. Such a situation occurs when a peripheral internally

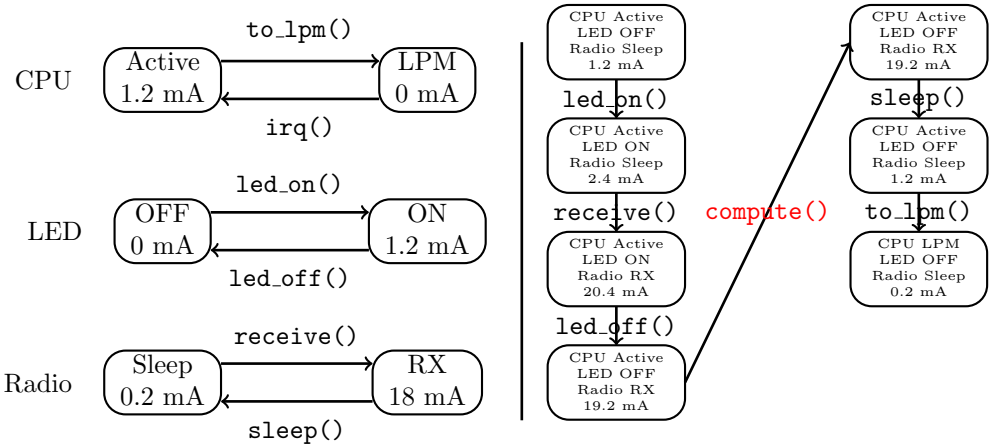


Figure 5.3: Example showing how the platform's power state evolves with driver calls. On the left, the individual power state machines of the CPU, LED and radio chip. The values are displayed for illustrative purposes and do not correspond to actual current measurement. LPM stands for Low-Power Mode. On the right, the evolution of current consumption following a particular software trace. The `compute` call does not alter platform power state.

triggers another as a timer would, or when an external peripheral embeds its own logic and decides to change on its own. Again, the present model assumes driver call synchronicity.

A software program is modeled as a control-flow graph where each node consists of a sequence of either regular assembly instructions or calls to driver functions. Software code that does not call any driver function may change the application state but may not impact the platform power state. On the contrary, only driver code is allowed to modify the peripherals and thus, the platform power state. Driver routines are modeled as atomic function calls, even though in reality they might be interrupted by interrupt service routines. Figure 5.3 illustrates this model, applied to a simplified example platform. The peripheral models, on the left-hand side, are only illustrative here and the real models are detailed in Section 5.4.1. The example follows a linear control-flow graph, where driver calls, such as `led_on`, and computational functions, such as `compute`, are executed. While `compute` is being executed, the platform power state does not change whereas the driver calls change the platform power state.

As for other chapters of this thesis, the model relies on a distinct separation between computational code and peripheral operations through the definition of driver code. As discussed in Section 3.2.1, assuming the existence of a separate driver code is totally realistic for both bare-metal and system-supported views.

5.4.3 Power Supply Model

This chapter supports two power supply models. The first model consists in supplying the platform with continuous supply. It corresponds to battery-powered scenarios where the battery is able to supply steady power during years. The second model consists in harvesting energy and storing it into a capacitor that supplies the platform through a voltage regulator. Power outages are likely to occur often. Hence, the energy harvesting supply model requires a power manager in order to schedule charge and discharge phases. Both these models also assume that the platform is supplied with a constant, steady voltage supply. This assumption is realistic for voltage regulators are used in continuously-powered systems and in some power managers for intermittent systems based on energy harvesting as discussed in Section 2.1.3.

5.5 Energy Consumption Measurement

There is a plethora of ways to measure instantaneous current. All of them have their advantages and drawbacks, as well as their own performance profiles: supply voltage perturbation, amount of noise, dynamic range, *etc.* In the particular case of this work, the measurement platform must provide a high

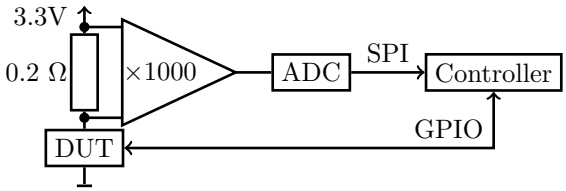


Figure 5.4: Measurement circuit schematics.

sampling frequency because power consumption may change every dozen of instructions. It must also provide a high dynamic range.

The measurement circuit relies on a simple design, similar to that of EMPIOT [109]. The circuit, shown in Figure 5.4, includes a shunt resistor of 0.2 Ω in series on the high-end of the supply voltage of the device under test (DUT). The voltage across the shunt resistor is amplified using Texas Instruments' INA212 operational amplifier with a gain of 1000 V/V. The output of the operational amplifier is fed to Texas Instruments' ADS8661 12-bit Analog-to-Digital Converter. The controller circuit is a Raspberry Pi 3 Model B, executing bare-metal code in order to have a better control over timing constraints than when executing on top of a non-realtime operating system such as Linux. The regulated 3.3V power rail of the Raspberry Pi 3 powers the device under test, while the unregulated 5V power rail powers both the operational amplifier and the ADC. Before measuring, the monitoring device is calibrated using a set of known resistors that emulate fixed, constant current loads. The calibration is a linear regression. In this case, the linear slope coefficient is 200 V/A, which corresponds to the 0.2 Ω resistor amplified by the operational amplifier with a gain of 1000 V/V.

The circuit actually measures the voltage across the shunt resistor, which is linear to the instantaneous current $i(t)$ drawn by the device under test. The energy consumption of executing a piece of code, ΔE, is given in Equation (5.1), where $V_{\mathrm{CC}}(t)$ is the instantaneous supply voltage, t_0 and t_1 the time-stamps corresponding respectively to the beginning and the end of the operation to monitor.

$$\Delta E = \int_{t_0}^{t_1} V_{\mathrm{CC}}(t) \times i(t) \times dt \tag{5.1}$$

As mentioned in Section 5.4.3, the supply voltage V_{CC} is constant over time, either in a continuous power scenario or an energy harvesting scenario with a voltage regulator. Hence, V_{CC} may be moved out of the integral in Equation (5.1). Furthermore, the integral of the current can be substituted by the integral average of the current, multiplied by the elapsed time. Here, the current data is discretized due to sampling so the energy evaluation uses the discrete current average $\overline{I_{[t_0,t_1]}}$ as an approximation of the integral average during the sampling time interval $[t_0, t_1]$. The expression of ΔE may thus be simplified as shown in Equation (5.2).

$$\Delta E = V_{\mathrm{CC}} \times \overline{I_{[t_0,t_1]}} \times (t_1 - t_0) \tag{5.2}$$

The instantaneous current $i(t)$ is obtained from the ADC values: $i(t) = I_{\mathrm{step}} \times X_{\mathrm{ADC}}(t) + I_{\mathrm{offset}}$, where I_{step} is the current increment for each ADC step, X_{ADC} the value returned by the ADC and I_{offset} the measurement offset due to the operational amplifier. The average of the ADC samples over $[t_0, t_1]$ defines $\overline{X_{\mathrm{ADC},[t_0,t_1]}}$, thus echoing the definition of $\overline{I_{[t_0,t_1]}}$. ΔE may be further defined as shown in Equation (5.3), which links energy consumption and sampled ADC values. After calibration, $I_{\mathrm{step}} = 6.4$ µA/step and $I_{\mathrm{offset}} = -4.0$ µA.

$$\Delta E = V_{\mathrm{CC}} \times (I_{\mathrm{step}} \times \overline{X_{\mathrm{ADC},[t_0,t_1]}} + I_{\mathrm{offset}}) \times (t_1 - t_0) \tag{5.3}$$

The value of ΔE given by Equation (5.3) is the energy consumed by the entire platform during $[t_0, t_1]$. This means that the micro-controller, as well as all other powered-on peripherals, are accounted. In order to isolate the energy consumption of a specific subset of components, the experiment operator must subtract, to ΔE, the energy consumption of the rest of the platform. For instance, when sending a radio packet, the micro-controller is in active mode and consumes energy as such. To determine the radio-related energy consumption of sending a radio packet, the energy consumption of the micro-controller's active mode during the entire time interval must be subtracted. Obviously, to reduce the amount of

Table 5.1: Measurements of driver calls energy consumption and duration for the radio, temperature sensor and accelerometer as implemented in Sytare. $\Delta E(x)$ and $\Delta t(x)$ are represented on Figure 5.5.

Driver call	Δt (µs)	ΔE (µJ)	from	to
`rf_init`	451	10.3	uninit	init
`rf_config`	575	4.5	init	idle
`rf_idle`	112	6.0	RX	idle
`rf_wakeup`	399	3.1	sleep	idle
`rf_sleep`	25	0.2	idle	sleep
	25	1.1	RX	sleep
`rf_rx_enter`	952	28	idle	RX
`rf_send(x)`	$\Delta t(x)$	$\Delta E(x)$	idle	idle
`temp_init`	159	0	uninit	init
`temp_sample`	76	0.2	init	init
`accel_init`	55	0	uninit	off
`accel_on`	1025	46	off	on
`accel_off`	10	0	on	off
`accel_sample`	171	0	on	on

other components to subtract, it is recommended to disable as many other components as possible before performing the measurements, leaving only the absolute necessary components on.

The whole monitoring device achieves a dynamic range between 8 µA and 26 mA, samples at 170 kHz and presents a noise that theoretically would correspond to 6 least significant bits. In practice, such a high noise was not observed during the experiments. High-frequency current noise incurs errors in energy measurement. But since integrated high-frequency signals result in small quantities, the overall error is expected to be small as well.

This power monitoring methodology has the advantages of being simple, low-cost and compatible with any platform for the only hardware requirements are basic: a power supply pin on which to plug the shunt resistor, and a couple of GPIOs. The value of the shunt resistor and the gain of the operational amplifier are specific to the targeted platform though.

5.6 Simulation and Energy Consumption Prediction

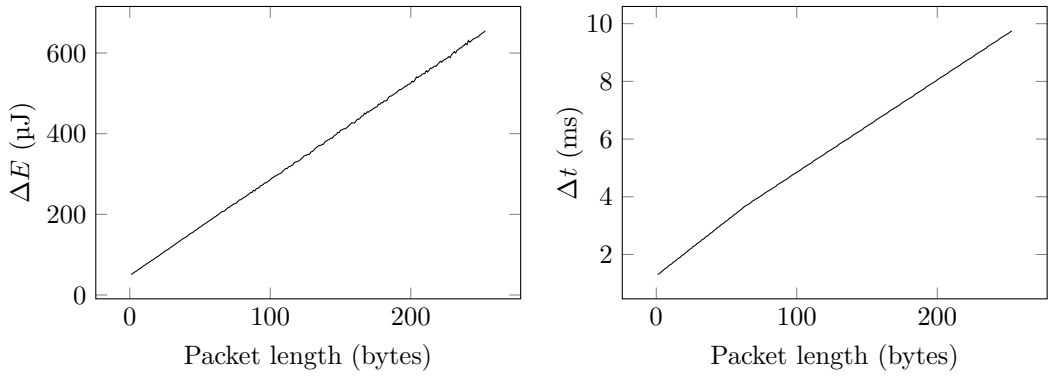
In order to provide an estimation of the energy consumption of a piece of code, the model described in Section 5.4 needs to be populated with numerical values obtained using a measurement device such as the one described in Section 5.5. Notably, the transitions of the driver state automata are tagged with a duration and an energy consumption. It is crucial to get those values right in order to yield an accurate estimation.

5.6.1 Regular Driver Calls

Measurements of duration and energy consumption for driver calls, *i.e.*, state transitions, are given in Table 5.1. Some driver calls *appear* to consume no energy, because their consumption is already accounted in the consumption of the platform over the measured duration. In other terms, when the state of the platform is never changed by the driver routine, no additional energy consumption is accounted. This is the case of `accel_sample` that samples accelerometer values without changing the state of the accelerometer nor the rest of the platform. However, it does not stand for `rf_send` for that driver routine makes internal changes in the state of the radio chip.

Table 5.2: Difference in measured current levels between `on` and `off` states of the Memory Protection Unit (MPU), accelerometer and LED.

Driver state	$i_{\text{on}} - i_{\text{off}}$ (µA)
MPU	0.54
Accelerometer	385
Single LED	1230



(a) Energy consumption (ΔE). (b) Time duration (Δt).

Figure 5.5: Measured (a) energy consumption and (b) duration of radio emission for different packet lengths.

5.6.2 On/off drivers

Some drivers might only present a simple, two-state machine. For instance, LEDs are based on GPIOs, but since it only makes sense to use them as outputs, the state machine of each LED has two states: `on` and `off`. Table 5.2 shows current measurements for some peripherals modeled as on/off drivers. Their driver state automaton could be more complex, however in practice, their set of driver routines currently only achieve transitions between two states. Current is computed by averaging the platform current during a certain amount of time. Computing the difference between two power states of the platform cancels out the consumption of other components that might be powered on.

5.6.3 Driver Routines with Parameters

Driver call energy consumption also depends on the parameters that are passed to the driver routines. For instance, the length of the packet to be sent has a substantial impact on time and energy costs. In order for any prediction to be accurate, the influence of parameter values must be modeled.

A parameterized driver call is exemplified by the `rf_send` driver call of radio peripheral. The measured energy and duration of the packet-sending driver routine are shown in Figure 5.5. For each packet length from 1 byte to 254 bytes, packet emission was measured 64 times. Energy measurements have a standard deviation not higher than 3.6% of the average value. Duration measurements have a standard deviation of at most 11.4 µs in a few cases, and 1.8 µs in average. The relationship between energy and packet length may be modeled as a simple a linear regression: $\Delta E(L) = 2.39 \times L + 47.71$ µJ, where L is the packet length. The relationship between run-time and packet length is better modeled as a two-part linear regression which parameters change at 64 bytes, that is the size of the internal transmission FIFO of the radio peripheral. For packets smaller than 64 bytes, $\Delta t(L) = 38.1 \times L + 1263.3$ µs. For larger packets, $\Delta t(L) = 32.0 \times L + 1646.3$ µs. The discrepancy between energy and time behaviors originates from the energy of populating the FIFO being a few orders of magnitude lower than the energy of actually sending the packet. Hence, the influence of the FIFO limitation is less visible regarding energy in comparison to time.

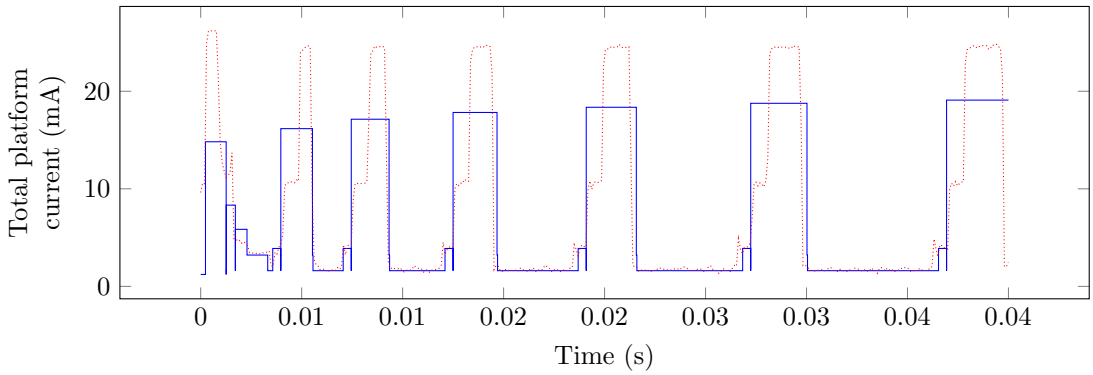


Figure 5.6: Excerpt of simulation record, in solid blue, and actual measurement, in dotted red, of the first seven data batches of the Complete-WSN application. The application uses a radio, an accelerometer and temperature sensor.

5.6.4 Integration in Simulation

The model described in Section 5.4 was implemented on a simulator [1] built on top of ArchC [115]. ArchC is a language for CPU architecture and ISA description. It aims at generating cycle-accurate SystemC code for simulation purposes. For the needs of the simulator, an ArchC model of the MSP430X instruction set was written, in order to execute the binary images compiled for MSP430FR5739 without any modification. This makes the energy estimations relevant for the simulator and the real platform run the same binary image.

Compared to other state-of-the-art simulators, the proposed simulator really executes the genuine software parts, *i.e.*, the computational parts of the code, in a cycle-accurate fashion. The driver part is run symbolically: instead of executing every single instruction of a driver call, the whole routine is bypassed and only its functional effects are simulated. The time duration and energy consumption are directly taken from the measurements presented above. As far as genuine software is concerned, the actual pipeline of the MSP430X architecture not being open-source, the simulator uses a simple heuristic to simulate instruction pipelining: all instructions have their theoretical cycle costs subtracted by 1. Experiments from Section 5.7 show that this estimation gives adequate results.

The simulator comes with two power supply models, as aforementioned: (i) continuous supply and (ii) energy harvester with a power manager that stores energy into a capacitor and powers the device under test through a voltage regulator. In the second scenario, the voltage conversion is considered conservative so far, *i.e.*, there is no energy loss due to the circuitry. When the capacitor voltage drops below a certain threshold, the simulator generates an interrupt and calls the software-defined interrupt handler of the kernel as it would be done in a real scenario of just-in-time systems such as Sytare. Then, when the capacitor voltage drops further to a lower threshold, the simulator virtually switches the device off, refills the capacitor and restarts the virtual platform by running the reset entry of the kernel interrupt vector.

Figure 5.6 illustrates the simulation, under continuous supply, of a specific application which repeatedly senses accelerometer and temperature data and sends the data over radio. The graph also shows actual current values measured on the same application on real hardware. The current peaks correspond to sending radio packets. The main discrepancy between simulation and measurements is that the simulated peaks are smaller yet broader. This observation is a consequence of the usage of the discrete current average $\overline{I}$ of the model depicted in Section 5.5. $\overline{I}$ is computed so that its temporal integral is the same than the temporal integral of the instantaneous current.

[1] `https://github.com/gberthou/archc-msp430x/tree/sytare-syscalls`

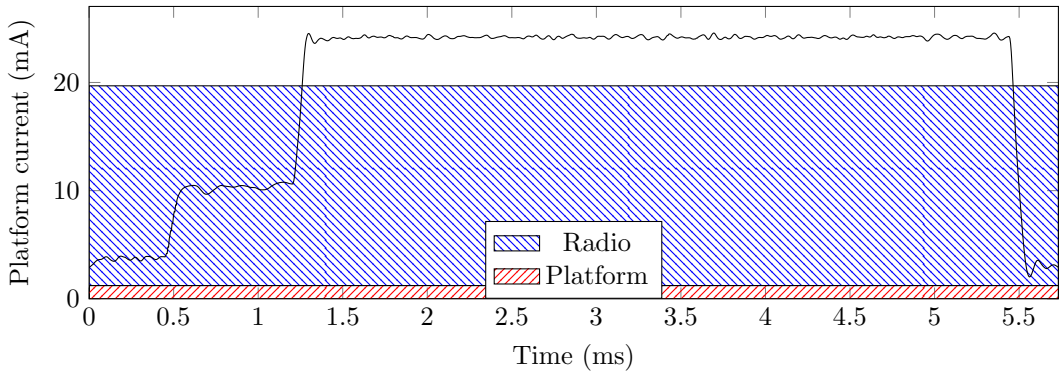


Figure 5.7: Instantaneous current consumption for a transmission of a 128-byte long radio packet. Hatched areas depict the equivalent surface corresponding to the consumption of the radio, in blue, and of the rest of the platform, in red. These averaged values are used in simulating transmissions.

The soundness of this model is asserted, as shown in Section 5.7, if the measured energy consumption and the simulated one match. For instance, in Figure 5.7, the area under the curve of the instantaneous current is equal to the sum of both hatched areas. This principle is what makes the present model both simple and accurate.

5.6.5 Benchmark Applications

The simulation is tested against a benchmark made of the following applications.

LEDs LEDs counts from 0 to 255 while displaying the counter on the LEDs. It is the same application as defined in Section 4.6.2.

Accelerometer Accelerometer turns on the accelerometer, performs ten measurements and turns off the accelerometer, waits one millisecond, repeated 256 times.

Radio Radio puts the radio into sleep mode, waits one millisecond, puts the radio to idle mode and sends a 128-byte long packet, repeated 256 times.

Complete-WSN Complete-WSN uses accelerometer, temperature sensor and radio. It slightly differs from the WSN application from Section 4.6.2. It senses acceleration and temperature several times as data batches and sends the data batches over radio as small packets. The first data batch contains one acceleration and temperature measurement, the second one contains two measurements, and so on, so that data batches grow with time. Data batches are sent over radio through 32 byte-long packets, without any particular network layer that would ensure reception on the other end of the radio channel. The radio is sleeping while sensing data and the accelerometer is always on. The maximal data batch size is statically defined.

5.7 Experimental Results

The results of these simulations are shown in Table 5.3 and Table 5.4. Simulation achieves less than 1% time estimation error and less than 5% energy estimation error, which makes this methodology suitable for a precise estimation of low-power embedded systems with important peripheral usage. The platform depicted in Section 5.5 enables to measure the instantaneous current drawn by the device under test. For instance, Figure 5.7 shows the evolution of the platform current during a radio emission of a 128-byte long packet (`rf_send(pkt)` driver call in Table 5.1). In this specific example, the behavior of the radio transmission is simplified during simulation, but its simulated energy consumption is still accurate, as shown in Table 5.4.

Table 5.3: Comparison, for execution time, between measurements and simulation of the benchmark applications on the MSP-EXP430FR5739 platform.

Application	Measured Δt (ms)	Simulated Δt (ms)	Δt error (%)
LEDs	274	274	0.0
Accelerometer	948	959	1.2
Radio	1815	1836	1.2
Complete-WSN	783	781	0.3

Table 5.4: Comparison, for energy consumption, between measurements and simulation of the benchmark applications on the MSP-EXP430FR5739 platform.

Application	Measured ΔE (µJ)	Simulated ΔE (µJ)	ΔE error (%)
LEDs	5596	5595	0.0
Accelerometer	17113	16309	4.7
Radio	102706	107285	4.5
Complete-WSN	16114	16315	1.2

5.8 Conclusion

The main contribution of this chapter is a new simplified energy model that enables the accurate simulation of low-power embedded systems using peripherals. This is achieved by modeling driver calls at coarse grain, in a parametric way, while the rest of the code is simulated at instruction-level.

The proposed model is validated by implementing a simulator that may be adapted to support any Instruction Set Architecture and any peripheral. In the experiments conducted here, the simulator targets an MSP-EXP430FR5739 platform equipped with an accelerometer, a temperature sensor and an external radio chip. For this platform, the simulator achieves good estimation: less than 1% time estimation error and less than 5% energy estimation error. An important aspect of this work is that it is suited to intermittent system simulation as well.

The energy consumption of the peripherals and driver calls has been precisely obtained using a low-cost measurement platform, hence accessible to any embedded system designer.

The aim of the simulator is average time and energy consumption estimations, while plenty of other works only focus on the worst case. Though it is necessary to evaluate worst case for software and hardware design purposes, it is also important to get a more realistic estimation which order of magnitude better reflects the behavior of the platform. Indeed, average cases are more representative of the probable run-time conditions and transiently-powered systems often use checkpointing.

Using a precise simulator is useful, not only to measure the energy needs of a given portion of code, but also to conduct large experiments on sundry binary images, while taking advantage of the resources of the simulation environment. This approach supersedes the slow traditional procedure that would consist in flashing a binary image and testing it against measurement devices, with a limited amount of hardware platforms. Such a usage of the simulator is made for the generation of the results exposed in Section 6.4, for they involve hundreds of binary images to be tested under several environment scenarios.

In the specific case of transiently-powered systems with heterogeneous memory, such as the MSP430FR micro-controllers equipped with both volatile RAM and non-volatile RAM, the energy consumption model could be further improved by adding a memory model that would support heterogeneity. For instance, the non-volatile RAM of the MSP430FR5739 micro-controller is thrice slower than its volatile RAM. Notably in scenarios where the software evenly shares its memory accesses between a slow and a fast memory, the simulator might benefit from a finer memory model.

In the experiments, the driver interacted with a virtual environment, *e.g.*, by sending radio packets. The simulator could be further improved by adding a scenario description language that would allow the experiment operator to encode some data into the simulation environment. Such data could be the contents of an incoming radio packet, or could be the occurrence of some external event, such as a button being pressed. This perspective would increase the reproducibility of experiments and provide a sound simulation environment to virtually deploy platforms and analyze their consumption.

Chapter 6

MPU-Based Incremental Checkpointing

The checkpointing schemes depicted in Chapter 3 and Chapter 4 do achieve persistence and guarantee consistency between memory and peripherals. However, some of these approaches, notably the ones that do not leverage a static analysis and variable liveness analysis, may save or restore more data than needed. Indeed, under intermittent power, life-cycles may be expected to be fairly short, reducing the likeliness of the application to either modify or read all of its variables before power fails.

This chapter focuses on application state persistence, that is P1. Extending the incremental checkpointing concept to peripheral state is more complex. It is actually already proposed by Sytare, as discussed in Section 4.4.4, for the evolution of the state of peripherals is progressively and incrementally committed to the `next` checkpoint image. Sytare then proposes a two-phase checkpointing system: a just-in-time mechanism for application state persistence and an incremental mechanism for peripheral state persistence. The idea behind this chapter is to make application state persistence incremental as well, albeit still performed once per life-cycle.

While existing approaches already target incremental checkpointing, this one aims at providing incremental checkpointing with minimal overhead, by using a simple component present in most of today's micro-controllers: a Memory Protection Unit (MPU). MPUs are embedded in low-power micro-controllers for they are not as elaborated as Memory Management Units (MMU), for instance. Traditionally used for process isolation, to run untrusted code or to prevent memory-related bugs resulting from a bad memory management, MPUs may also be used for checkpointing purposes. The technique described in this chapter results from a former publication [116] while the benefits of such a technique are studied in greater depth here.

The basics and elementary notions of incremental checkpointing are stated in Section 6.1, including existing software-based and hardware-based approaches. Then, the overall design of this MPU-based incremental checkpointing proposal, evoking earlier works as stated later, is depicted in Section 6.2. A novel generic yet realistic model complements this contribution. The model, described in Section 6.3, includes eleven parameters and enables to reason about the energy consumption of the MPU-based approach in comparison to a full copy. The model also serves as a basis for discussion of transiently-powered systems requirements. The analysis shows that even for short amounts of RAM, *e.g.*, 20 kB, MPU-based incremental checkpointing is much less energy consuming than a full copy of the RAM contents, even when taking advantage of the speedup brought by a Direct Memory Access (DMA) component. The current proposal was implemented and executed on a cycle-accurate simulator of the FRAM-based MSP-EXP430FR5739 demonstration platform. Simulation confirms the reasoning brought by the model, as demonstrated in Section 6.4, and the benefits of this technique for transiently-powered systems that use checkpointing.

6.1 Towards Incremental Checkpointing

Energy available for transiently-powered systems is scarce and thus need to be carefully managed. The worst-case scenario would be that the operating system consumes the entire energy budget for its own needs, leaving nothing for the application, which would substantially annihilate the motivation for using an operating system. Hence, the operating system must provide efficient mechanisms for whatever purposes it is made for. This is true for all systems, however even if a desktop application may suffer from a few slow-downs and give an uncomfortable experience to the human user, the situation would threaten the feasibility of an application under intermittent power assumptions.

In general, such systems use optimized copy operations to populate their checkpoints. The use of a Direct Memory Access, enabling hardware copy operations at a frequency close to that of the memory bus, is very common. Operating systems yet need to optimize how they orchestrate their copy operations. Specifically, two axes of improvement are to be investigated: checkpointing less often and performing smaller copies. Those two axes are orthogonal and the present contribution targets the latter: smaller copies. The first axis, checkpointing less often, is already addressed by just-in-time checkpointing, as described in Section 3.4.2.

Some works propose to decrease the amount of RAM copied to non-volatile RAM, either by considering the role of each region of the RAM [98] or by considering incremental checkpointing [117, 98]. Several works have proposed solutions to improve checkpointing efficiency, *i.e.*, minimizing energy spent for checkpointing.

6.1.1 Software-based Approaches

Aït-Aoudia *et al.* [117] proposes an incremental checkpointing scheme to reduce the amount of writes to non-volatile RAM as much as possible. Their solution leverages a kernel layer, that allocates equally-sized blocks stored in non-volatile RAM. They use a double-buffered checkpoint image in order to prevent permanent crashes. They define a checkpoint image as an ordered array of pointers that point to the address of the non-volatile blocks. If a portion of memory did not change between two consecutive checkpoints, the analogous pointers point to the same block. If the portion of memory did change, then the analogous pointers target separate blocks. The block pool requires garbage collection and a block allocator. Modified blocks are detected by hashing the contents of every block. Thus, their approach presents two sources of overhead: (*i*) the performance of the implementation of the garbage collector and block allocator and (*ii*) the performance of the hash algorithm. No information is given about the hash algorithm, however it may be assumed that its execution time is linear to the block size. In addition, the hash algorithm might introduce redundancy, thus bugs may arise whenever two different block contents yield the same hash value. Depending on the nature of the hash algorithm, this situation is not likely to happen, but since platforms that harvest energy are designed to run almost forever, this may occur in the platform's life-time.

Bhatti and Mottola [98] assign separate roles to different RAM regions, namely the stack, heap, `.bss` and `.data` sections. The main assumption of their work is that the non-volatile storage is Flash which, as discussed in Section 2.5.1, has page constraints. They propose three optimizations. The first one, named Split, simply discards unused stack and does not take into account heap fragmentation: it solely excludes the unused stack from being part of the checkpoint image. The second one, named Heap Tracker, discards unused stack and unused heap. The third one, named Copy-If-Change, proposes an incremental checkpointing scheme. During the first checkpoint operation, the Flash memory needs to be populated, so the entire RAM contents are copied to Flash. Then, only modified portions of RAM need to be updated in the Flash storage. Since their approach assumes a Flash storage, the portions of RAM are studied at the granularity of the Flash page size, for page size is the smallest Flash memory unit that may be updated. The authors did not discuss the block modification detection algorithm of Copy-If-Change, but rather disclose that the Flash-located counterpart of a block is read and compared to the actual RAM block. It can thus be assumed that the comparison is linear to the block size, and that the comparison algorithm early stops when the first discrepancy is found. In addition, it must be noted that the block modification detection algorithm requires potentially numerous Flash reads and that read accesses to Flash may be slower than read accesses to RAM. Flash-related optimizations, such as page alignment and making data contiguous, are out of scope here since the present thesis focuses on the

usage of non-volatile RAM over the usage of other non-volatile memories.

DICE [118] is a checkpointing optimization scheme that is compatible with statically-provisioned checkpoints as well as just-in-time checkpointing techniques, such as the ones depicted in Chapter 3. DICE handles differently `.bss`, `.data` and heap sections on one side and stack on the other side. A compiler pass identifies direct writes to global variables, *i.e.*, `.bss` and `.data` sections, but accesses to global variables through pointers must be instrumented at run-time. The compiler pass records the updates along the natural execution of the application. DICE stores the records as a bitmap where each bit maps a byte of main memory. While this feature enables the actual save operation to already know what variables to save, unlike approaches that require to determine block modification at that critical point [117, 98], it places overhead on every single write to those variables. More specifically, if an application requires writes to the same variable several times in a short time-span, DICE would record the writes several times albeit not necessary in case the next power outage occurs later on. The stack is instrumented by the compiler pass as the latter can evaluate the stack frame size of a given function. Only the current topmost working portion of the stack is saved, which corresponds to the modified stack. The stack pointer is handled with special care for, upon returning from a function, lower portions of stack may also be modified.

eM-map [87] proposes an interesting memory mapping where each function is free to allocate its variables either in volatile RAM or non-volatile RAM. It is not an incremental approach *per se*, however eM-map also aims at, amongst others, reducing the energy dedicated to checkpointing.

Libckpt [119] provides transparent checkpointing for Unix-based systems. As far as incremental checkpointing is concerned, libckpt uses the Unix `mprotect` system call to write-protect all memory pages of the application data, *i.e.*, to make them read-only. An access mismatch generates a Unix SEGV signal and the corresponding signal handler from libckpt flags the page as modified. Upon next checkpoint save operation, only modified pages are included in the checkpoint image.

6.1.2 Hardware-based Approaches

Few works leverage hardware for checkpointing. Bartling *et al.* [120] propose to design a non-volatile micro-controller which automatically saves all CPU and peripheral registers to FRAM, upon detecting a power outage. However, this solution proposal leverages a new kind of architecture, not necessarily accessible to researchers for the time being.

Egger *et al.* [121], whilst studying operating systems for high-power virtual machines, observe that system caches operate at the granularity of disk blocks. They propose to spy on the disk operations in order to flag memory pages that are identical to their disk page counterpart. Using this knowledge, the operating system may safely exclude these identical, *i.e.*, unmodified, memory pages from the checkpoint image. Their run-time assumptions are far from the ones of transiently-powered systems, considering the fact that they target continuously-powered high-power virtual machines, yet the aims are identical. While checkpointing for transiently-powered systems solely tackles power outages, in the world of virtual machines, checkpointing is mostly used for any kind of failure or to stop the virtual machines at any time. Virtual machines issue their disk operations using a hypervisor and Egger *et al.* propose to dynamically instrument the hypervisor. The hypervisor is a piece of software that provides the virtual environment in which the virtual machines run. It notably provides an API to isolate the virtual machines peripheral accesses from the actual operating system and physical machines that support them. Egger *et al.* propose to write-protect the memory pages mapped to the disk blocks, using a power-consuming MMU. An MMU is a hardware component, present in some power-consuming micro-controllers and micro-processors, that provides virtual addresses and access right management to software-defined parts of the memory. MMUs must maintain a translation lookaside buffer that stores the address mapping configuration. In the work of Egger *et al.*, a write access to a read-only address triggers an MMU page fault exception, which grants write-access to the faulting page and restarts the aborted write operation. However, MMUs are absent from low-power embedded designs because of their power requirements.

Freezer [122] leverages a dedicated piece of hardware as a checkpointing helper. It is a checkpoint controller that spies on the address bus and manages a bitmap of modified memory blocks, one bit per block. Upon imminent power outage detection, the bitmap is parsed in order to determine whether to save the corresponding memory block. Freezer is designed to suit the energy demands of transiently-powered systems with a hardware solution that is tailored for checkpointing purposes, without the intervention

Table 6.1: Characteristics of the MPUs from two low-power architectures. A suitable MPU depicts an MPU that can efficiently be used for incremental checkpointing, *i.e.*, an MPU that may operate over the entire volatile memory range and that can replay the faulting instruction upon returning from access mismatch interrupt.

MPU	Address range	Regions	Region size	Suitable
MSP430FR57xx	NVRAM only	3	Customizable	No
ARM Cortex-M	Entire memory	8 with 8 subregions per region	Customizable one region at a time	Yes

of software. It thus supersedes the usage of an MMU. However, Freezer remains a specific hardware component and, consequently, cannot be used on today's off-the-shelf platforms.

These works define incremental checkpointing. In order to selectively save memory regions from volatile to non-volatile memory, the entire volatile RAM must be mirrored in the checkpointing image at least once, or more in case of double-buffering for instance. This stands regardless of the checkpointing mechanism being incremental or not, for the system must always be able to repopulate the RAM upon restoration process.

This chapter proposes a similar scheme to the one of libckpt [119] and Egger *et al.* [121]. Instead of using Unix system calls that would draw heavy dependencies and would hardly fit on today's transiently-powered systems and instead of leveraging a high-power MMU, the present solution features a low-power and less sophisticated MPU. This key difference enables to extend the usage of this type of incremental checkpointing to more modest micro-controllers and harsher run-time conditions. In addition, this work exposes a parameterizable model of the energy consumption of the save operations in case of a full copy or in case of an MPU-based incremental approach. This novel model is totally distinct from the implementation and is a contribution by itself, for it greatly helps to design platforms and to choose kernel policies. Note that the chapter is written as if assuming that the incremental checkpointing mechanism was implemented on top of Sytare. It is nevertheless independent of the underlying system layer and can be easily adapted to any other approach.

6.2 Incremental Checkpointing Design using an MPU

An ideally efficient incremental checkpointing mechanism would only save data that have been modified during the current life-cycle. This work heads towards this goal, through the utilization of an MPU in order to reduce the quantity of data saved into non-volatile RAM, *i.e.*, to get as close as possible to the ideal case. This section briefly recalls the principle of MPUs, then the present proposal of MPU-based incremental checkpointing mechanism is detailed, as well as the MPU requirements to make the proposal feasible.

6.2.1 Memory Protection Units

Low-power embedded systems do not have hardware support for memory virtualization through MMUs. An MMU is energy-expensive, mainly because of the presence of the Translation Lookaside Buffers, made of associative memory. However, many embedded systems include hardware support for memory isolation thanks to MPUs which enable to *manage access rights* to some memory region.

The characteristics of an MPU substantially vary from one platform to another but their goal is the same: guarantee memory integrity and fire interrupts upon access rights mismatch. Table 6.1 gives the characteristics of MPUs present on the MSP430FR57xx and ARM Cortex-M platforms. The criterion "Suitable" indicates whether a given MPU may easily support this technique, as explained hereafter.

6.2.2 Leveraging an MPU for Incremental Checkpointing

The main objective of this work is to reduce time elapsed in operating system context, to allow the application to execute longer with respect to the operating system. Here, the operating system cannot

afford to compute checksums on memory regions as in previous works [117]. The present proposal consists in using MPU as hardware support for incremental checkpointing, while the software remains oblivious to the detection of modified variables. The idea is to keep track of whether an MPU region has been modified since the last checkpoint or not.

Set of dirty regions

Let D be the set of *dirty* regions, *i.e.*, MPU regions that have been modified since the last checkpoint. When a write occurs in the region R_i of RAM, the MPU triggers a kernel-managed interrupt. In practice, D may be implemented as a bitfield. MPUs being simple, the amount of regions or sub-regions rarely exceeds 64, which makes any implementation of D remain a simple data structure.

Boot sequence

Right after boot, D is empty ($D = \emptyset$) and all regions of volatile RAM are write-protected by the MPU. This work does not provide a lazy policy for data restoration, meaning that the restoration routine is not modified further than the initialization of D and of the MPU. In the very specific case of Sytare, Algorithm 4.2 still stands. This is complementary to any checkpointing mechanism.

MPU access mismatch interrupt

An MPU interrupt is fired whenever an unauthorized access to some memory location is attempted. Within the context of this proposal, an MPU interrupt is fired only when the application tries to write to a read-only location. The interrupt handler marks dirty the region that possesses the faulting location: $D \leftarrow D \cup \{R_i\}$. The interrupt handler then unprotects the region, so that further write accesses to the region are allowed. As a consequence, the set of protected regions is always the complementary set to D; *i.e.*, locked regions $= \{R_i \forall i\} \setminus D$. The interrupt handler finally returns to the *faulting instruction*, *i.e.*, the instruction that caused the write mismatch access. The notion of faulting instruction is crucial to this approach, for the system needs to re-execute the instruction with the updated access rights. Now that the region is unprotected, the write access can complete without further interrupt and the execution resumes as usual.

Checkpoint save operation

When it is time to perform the save operation, the system reads the information held inside D. For each region registered inside D, the system performs a DMA copy from the original volatile RAM location into the `next` checkpoint image.

```
1  static int a = 0; /* Region R0, .bss  */
2  static int b = 1; /* Region R4, .data */
3
4  void main(void)
5  {
6      compute_1();
7      a = 42;
8      compute_2();
9      b = 24;
10     compute_3();
11 }
```

(a) C code.

Line	D
6-7	$\emptyset$
7-8	$\{R_0\}$
8-9	$\{R_0\}$
9-10	$\{R_0, R_4\}$

(b) Evolution of the set of dirty regions, without power outage.

Figure 6.1: Toy code that illustrates memory accesses to different MPU regions.

Scenario Example

The code sequence in Figure 6.1 illustrates this mechanism, assuming an 8 kB volatile RAM and an MPU capable of protecting eight equally-sized regions R_0 to R_7 of 1 kB each. First, the system initializes its intrinsic variables and mechanisms. At the end of that operation, $D = \emptyset$. The application runs `compute_1`, which does not change variables outside the CPU registers or the application stack, thus D stays empty. The application then writes to variable `a`, located in region R_0. Since, at that point, R_0 is write-protected, an MPU interrupt is triggered. The interrupt handler is executed. Upon interrupt handler completion, $D = \{R_0\}$ and the MPU is configured to stop protecting R_0. Line 7 is executed again, this time with R_0 no longer protected. The write to variable `a` that originally failed, is now executed correctly and the value of variable `a` is updated. The program continues its normal execution, runs `compute_2` that, like `compute_1`, does not raise any MPU interrupt. Then, when reaching line 9, the application tries to write a value to variable `b`. Given that variable `b` is not located in the same region that the formerly unlocked region of variable `a`, another MPU interrupt is raised. Upon interrupt handler completion, $D = \{R_0, R_4\}$ and the MPU is configured to stop protecting both R_0 and R_4. The write to variable `b` is successfully re-played and the application continues its execution. Later on, the system eventually wants to perform a checkpoint save operation. When that moment comes, only regions R_0 and R_4 are copied to non-volatile RAM.

MPU requirements

This design of MPU-based incremental checkpointing requires the MPU to provide a certain set of features. The MPU must (*i*) operate at least on the entire RAM address range and (*ii*) raise interrupts that are able to return to the faulting instruction. The "Suitable" criterion of Table 6.1 thus asserts whether a given MPU may be compatible with both aforementioned requirements.

The MPU of the MSP430FR57xx micro-controller family only maps non-volatile RAM, it is sufficient to discard that kind of micro-controller according to the criterion. In addition, its interrupt system would return to the instruction following the faulting instruction. Workarounds may be engineered in order to deduce the address of the faulting instruction by embedding a lightweight MSP430X instruction decoder, which would substantially hamper the performance of an MPU-based incremental checkpointing. Hence, the MPU of the MSP430FR57xx micro-controller family is not suitable for this. However, for practical purposes, since Sytare and its applications were written for the MSP430FR5739 micro-controller, an adapted version of the MPU was integrated to the platform simulator described in Section 5.6 in order to keep Sytare's code-base.

On the other hand, the MPU of ARM Cortex-M micro-controllers meet the MPU requirements and are thus perfectly befitted for this work. This is the purpose of the ARMorik platform detailed in Section 4.7, that comprises an ARM Cortex-M7 micro-controller. The MPUs from the Cortex-M micro-controllers constrain memory placement and require a more elaborate linker script, for the MPU page addresses must be aligned with their sizes, which are powers of two. Apart from that constraint, the MPU is flexible and enables efficient implementations of the present contribution.

The energy gain, compared to saving the entire RAM, depends on the characteristics of the executed application, as well as platform-related parameters. These characteristics and parameters are detailed in the next section.

6.3 Analysis of the Energetic Benefits of Incremental Checkpointing

Before proceeding to the implementation and its results, this work provides a model of the energy consumption of the save operation, for the application state only, with and without the MPU-based incremental approach. Any additional operation, such as saving the state of peripherals, is out of scope here as this approach does not monitor accesses to peripheral control registers. The baseline is a non-incremental approach, that is a full copy of the `.bss` and `.data` sections regardless of them being modified or not. This baseline is referred to as the *full-copy* approach.

Table 6.2: Model parameters and their default values, used in the models of the full copy and the MPU-based incremental approaches.

Symbol	Description	Unit	Typical value
S_{words}	Amount of RAM used by the application	Word	2^{13}
f_{DMA}	DMA bandwidth	Word/second	8×10^6
P_{plat}	Power drawn by the whole platform, without CPU, DMA nor MPU	Watt	1.65×10^{-2}
P_{DMA}	Power drawn by the enabled DMA	Watt	ϵ
P_{MPU}	Power drawn by the enabled MPU	Watt	1×10^{-5}
P_{CPU}	Power drawn by the unhalted CPU	Watt	3.96×10^{-3}
α	Average ratio of dirty regions within one life-cycle	-	0.1
N_{region}	Number of regions handled by the MPU	-	16
t_{overhead}	Time to check if a region must be copied	Second	3×10^{-6}
t_{int}	Execution time of the MPU interrupt handler	Second	5×10^{-6}
E_{critical}	Average amount of energy wasted due to re-executing code when the MPU interrupt occurs during a critical section	Joule	ϵ

The energy consumed by this process depends on several parameters. The model depicted in this chapter, as well as the results that come out of it, may be used as a base for design space exploration, in terms of hardware and software specifications.

6.3.1 Modeling MPU-based Incremental Checkpoint

Model parameters

The performance of this proposal heavily depends on some parameters, listed in Table 6.2. The amount of RAM used S_{words} is important for the bigger the memory is, the more vital it becomes to save energy by selectively checkpointing fractions of memory to non-volatile RAM. S_{words} is accounted word-wise, since the DMA is considered to be used at word granularity to use the best performance from the DMA. Hence, the actual RAM size is obtained by multiplying S_{words} by the size of a word on a given platform. For instance, the MSP430FR57xx has 16bit-long words, meaning that the RAM size in bytes would be twice as large. The operating frequency of the DMA f_{DMA} directly impacts the time needed to perform a checkpoint.

The average dirtiness ratio, α, is the average proportion of regions that have been modified since last checkpoint when a new checkpoint arrives. This parameter α is important: a small value gives this proposal better results. In many transiently-powered systems, little energy is available between consecutive checkpoints, allowing the execution of a few thousands or millions of instructions each time. Hence, α is indeed expected to be low. Note that α is a complex parameter, for it depends on the energy budget of the platform and on the application behavior, since the application may dynamically change the power consumption of the platform.

This proposal relies on a standard micro-controller equipped with an MPU. The amount of regions N_{region} that the MPU can handle is also crucial. If the regions are too numerous, the platform spends a long time handling interrupts. But on the other hand, with fewer regions, the checkpointing is less incremental and closer to a full copy of the RAM contents. The characteristics of the MPU define the upper bound of N_{region}, however the system is free to use fewer regions. Although the MPU is a hardware component, it is driven by software, which means that there is some time overhead due to the CPU running instructions. This overhead is specific to incremental checkpointing. There are two sources of software-related time overhead. First, this mechanism is interrupt-based which introduces an overhead, named here t_{int}, that is the time to handle the MPU interrupt. The interrupt handler must simply flag the concerned memory region as dirty, and unlock that region to allow further modifications from the software until the memory is saved in the checkpointing process. Second, during the checkpointing process, the software must check every region dirtiness flag to determine whether they must be copied or not, this overhead is called t_{overhead}. Both overheads are expected to be small, but not negligible, within the order

of a few microseconds for each.

This model also needs insight about some electronics aspects of the platform. Four distinct power sinks are considered: the micro-controller itself that consumes P_{CPU}, the DMA that consumes P_{DMA} apart from the micro-controller, the MPU that consumes P_{MPU}, and the rest of the platform, including peripherals, that consumes P_{plat}. The power consumption of the MPU, P_{MPU}, is only accounted in the incremental checkpointing since the full copy does not need the MPU and thus it can be turned off. The different power consumptions are platform-dependent and furthermore, P_{plat} also depends on the application since the amount and the nature of enabled peripherals depend on the state of the application at a given point in time. The default values of the power consumptions were chosen arbitrarily for the analysis. P_{CPU} is set to 3.96 mW, which corresponds to 1.2 mA, the measured consumption of the active mode of the MSP430FR5739, under a 3.3 V supply. P_{plat} is set to 16.5 mW which corresponds to a consumption of 5 mA under a 3.3 V supply. The value of P_{plat} can be interpreted as the time-related average of the platform, excluding the micro-controller, when the radio is on for reception or transmission during 25% of the time. P_{DMA} is considered negligible as low-power part of the micro-controller. P_{MPU}, although also part of the micro-controller, is considered to symbolically consume 10 µW. In practice, any increase in power consumption was not detected using the methodology depicted in Section 5.5. However, the analysis aims at being fair, so a symbolic non-zero value was given to P_{MPU} in order to encompass more power-consuming MPUs, while not reaching the power consumption of a power-consuming MMU neither.

Once acquainted with these parameters, the next section describes the equations that analytically compute the energy spent by performing a full copy of the entire RAM, named $E_{\text{full-copy}}$, by this MPU-based incremental checkpointing solution, referred to as $E_{\text{MPU-incremental}}$, as well as state-of-the-art approaches named E_{hash} and E_{DICE}.

Power levels

The model supports four power states. Indeed, the CPU and the DMA are considered mutually exclusive and the MPU is accounted only in the MPU-based approach unlike the other approaches that make no use of MPU. In the full copy approach, the copy is performed by the DMA, so the CPU is off, hence the power level of that approach is $P_{\text{DMA}} + P_{\text{plat}}$. In the rest of this chapter, for clarity purposes, combined power levels are depicted using a set notation, *e.g.*, $P_{\{\text{DMA, plat}\}} = P_{\text{DMA}} + P_{\text{plat}}$.

In the MPU-based approach, the MPU is always on. The copy is performed by the DMA, while the dirtiness detection and the interrupt handlers are run on the CPU. Hence, the MPU-based approach needs two power levels: $(i)\, P_{\{\text{DMA, MPU, plat}\}}$ and $(ii)\, P_{\{\text{CPU, MPU, plat}\}}$.

Finally, the software-based approaches studied here [117, 98, 118] present time overhead due to CPU-based computations. These specific phases use no MPU nor DMA, hence their power level is $P_{\{\text{CPU, plat}\}}$. These power levels are used in the following energy requirement analysis.

Full copy energy consumption

As the save operation of the checkpointing scheme is performed using a DMA, *i.e.*, without using the CPU, the energy required to checkpoint the entire RAM is simply given by Equation (6.1):

$$E_{\text{full-copy}} = \frac{S_{\text{words}}}{f_{\text{DMA}}} \times P_{\{\text{DMA, plat}\}} \tag{6.1}$$

MPU-based incremental checkpointing energy consumption

To compute the energy used by the MPU-based incremental mechanism, the expression of $E_{\text{MPU-incremental}}$ is more complex and the definition of intermediate symbols helps. To save a single region using DMA, with the CPU halted, the required energy is expressed as:

$$E_{\text{region}} = \frac{R_{\text{words}}}{f_{\text{DMA}}} \times P_{\{\text{DMA, MPU, plat}\}}$$

With $R_{\text{words}} = \lceil \frac{S_{\text{words}}}{N_{\text{region}}} \rceil$ the amount of words in a region. This model assumes, for simplification purposes, that the MPU regions are equally-sized, hence the definition of R_{words}. However, model aside, this

proposal of incremental checkpointing does not impose that the MPU regions should be equally-sized. On the contrary, a clever organization strategy of the MPU regions and of the variable allocation may improve the performance of the incremental checkpointing.

The energy consumed by MPU interrupt handling corresponds to the energy spent by the MPU interrupt handler plus the energy needed to re-run a portion of code, if for instance the interrupt occurred during a code section with timeliness constraints, *cf.* problem P3 and its solutions depicted in Chapter 3 and Chapter 4. The energy spent to re-run a timely portion of code is abstracted by E_{critical}. In reality, each timely portion of code has its own needs, meaning that there would be as many values of E_{critical} as concerned portions of code. However, for the sake of model simplicity, a unique value of E_{critical} is considered. In some cases, frequent interrupts might prevent the application from efficiently making progress. Hence, the energy of dirtiness detection, E_{detect}, is expressed as:

$$E_{\text{detect}} = t_{\text{int}} \times P_{\{\text{CPU, MPU, plat}\}} + E_{\text{critical}}$$

In the analysis presented in Section 6.3.2, the energy overhead due to re-executing atomic section is ignored, *i.e.*, $E_{\text{critical}} = \epsilon$ in Table 6.2, but in the cycle-accurate simulation in Section 6.4, it is simulated. In practice, it depends on the application requirements in terms of timeliness constraints, and on the way the system handles timeliness.

The energy to checkpoint only dirty regions, E_{dirty}, corresponds to:

$$E_{\text{dirty}} = N_{\text{dirty}} \times E_{\text{region}} + t_{\text{overhead}} \times P_{\{\text{CPU, MPU, plat}\}}$$

With $N_{\text{dirty}} = \lceil \alpha \times N_{\text{region}} \rceil$, the average amount of dirty regions per checkpoint.

Finally, the energy dedicated to incremental checkpointing, named $E_{\text{MPU-incremental}}$, during an entire life-cycle, is given by Equation (6.2):

$$E_{\text{MPU-incremental}} = E_{\text{dirty}} + N_{\text{dirty}} \times E_{\text{detect}} \tag{6.2}$$

It should also be noted that the MPU must be on at all times, which adds a background power consumption that constrains the energy budget over the entire life-cycle.

Hash-based energy consumption

The proposals of Aït-Aoudia *et al.* [117] and of Bhatti and Mottola [98] determine, at checkpoint time, which portions of memory are modified using a software computation that reads all memory regions. They are alike, albeit Aït-Aoudia *et al.* use a hash function whereas Bhatti and Mottola make a word-to-word comparison. The two proposals are referred to as, respectively, Hash and Sweep. The model of the energy consumption can be derived from the model of the MPU-based incremental checkpointing.

These solutions do not use the MPU, resulting in $P_{\text{MPU}} = 0$, $t_{\text{int}} = 0$ and $E_{\text{critical}} = 0$. In addition, the time overhead due to hashing or sweeping memory is not the same as t_{overhead}. As stated before, the time overhead of the hash- or sweep-based approaches is linear to S_{words}. A specific parameter, $t_{\text{hash-word}}$, is defined and represents the time taken to hash or sweep over a single word. Hence, the time overhead of those solutions is expressed as $S_{\text{words}} \times t_{\text{hash-word}}$. The energy requirements of saving a checkpoint for those solutions, named E_{hash}, is then expressed by Equation (6.3), derived from Equation (6.2):

$$E_{\text{hash}} = N_{\text{dirty}} \times \frac{R_{\text{words}}}{f_{\text{DMA}}} \times P_{\{\text{DMA, plat}\}} + S_{\text{words}} \times t_{\text{hash-word}} \times P_{\{\text{CPU, plat}\}} \tag{6.3}$$

Note that the amount of regions is no longer limited by hardware, however this study considers it equal to the formerly defined N_{region} for comparison purposes. In practice, the $t_{\text{hash-word}}$ of Aït-Aoudia *et al.* is expected to be larger than that of Bhatti and Mottola, for a hash operation over the entire memory is likelier to last longer than comparing words and early-stopping upon the first discrepancy. However, they are accounted the same within this study, and a default value of $t_{\text{hash-word}} = \frac{1}{24}$ µs is considered. This corresponds to a single machine cycle for an MSP430FR5739 micro-controller operating at 24 MHz. This value is chosen in order to show a fair comparison of the MPU-based approach with those approaches, taken under their sunniest side, *i.e.*, minimizing E_{hash}.

DICE energy consumption

DICE [118] records every write to a global variable through a `record` system call. That system call is purely software and may be expected to be fast, yet it is called every time a global variable is written to. Let W be the average amount of writes to global variables between two consecutive checkpoints. Let t_{record} be the completion time of the `record` system call. Then, the energy of recording every write to global variables, E_{record} is defined as follows:

$$E_{\text{record}} = W \times t_{\text{record}} \times P_{\{\text{CPU, plat}\}}$$

DICE saves the topmost portion of the stack. Let $S_{\text{topmost-stack}}$ be the average size, in words, of the topmost portion of stack. Then, the energy of copying it, $E_{\text{topmost-stack}}$, is defined as follows:

$$E_{\text{topmost-stack}} = \frac{S_{\text{topmost-stack}}}{f_{\text{DMA}}} \times P_{\{\text{DMA, plat}\}}$$

Finally, DICE copies the dirty bytes. The copy is performed at the granularity of the bytes, unlike the other incremental approaches that operate at the granularity of a block. Hence, the ratio of dirty bytes must be defined, named α^* to show the analogy with α. The amount of dirty bytes is thus $\lceil \alpha^* \times S_{\text{words}} \rceil$. Furthermore, DICE reads an internal bitmap in order to determine which bytes must be copied. This is performed in software and has a temporal cost for each bit, named t_{bit}. Since a bit corresponds to a byte in main memory, there are $\lceil \frac{S_{\text{words}} \times N_{\text{bytes-per-word}}}{8} \rceil$ bits, with $N_{\text{bytes-per-word}}$ the word width on a given platform. On an MSP430FR57xx platform, $N_{\text{bytes-per-word}} = 2$. The energy of copying dirty bytes, $E_{\text{dirty-bytes}}$, is then:

$$E_{\text{dirty-bytes}} = \frac{\lceil \alpha^* \times S_{\text{words}} \rceil}{f_{\text{DMA}}} \times P_{\{\text{DMA, plat}\}} + \lceil \frac{S_{\text{words}} \times N_{\text{bytes-per-word}}}{8} \rceil \times t_{\text{bit}} \times P_{\{\text{CPU, plat}\}}$$

Note that the definition of $E_{\text{dirty-bytes}}$ considers word-wise DMA copies, whereas in practice DICE checks dirtiness byte-wise and the dirty bytes are not necessarily contiguous in memory, meaning that many copies would be needed with software overhead between copies. Again, this study wants to demonstrate a fair comparison with state-of-the-art approaches and thus presents DICE under optimal conditions, *i.e.*, minimizing E_{DICE} as defined hereafter.

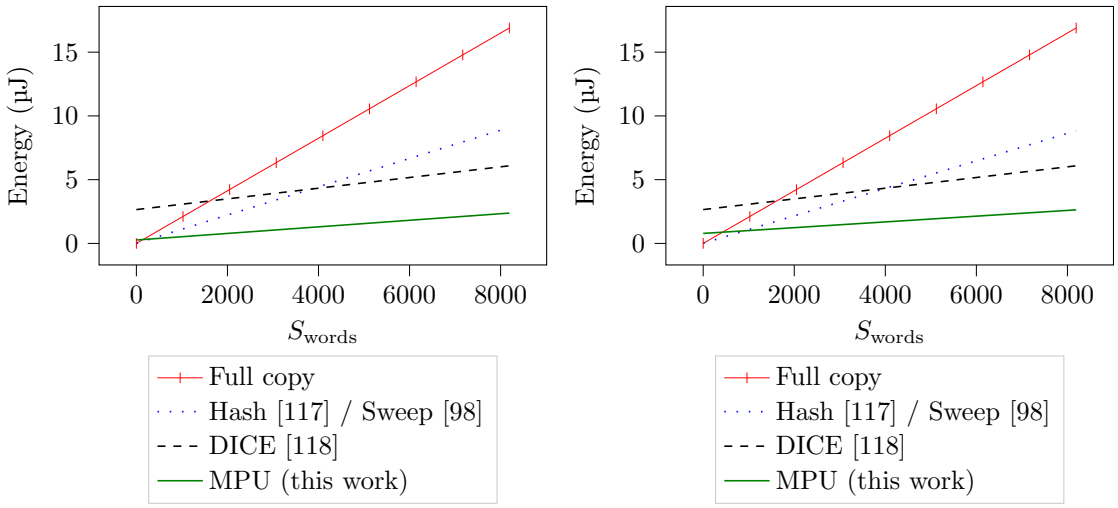
As a consequence, the overall energy of the save operation using DICE, E_{DICE} is summed in Equation (6.4):

$$E_{\text{DICE}} = E_{\text{record}} + E_{\text{topmost-stack}} + E_{\text{dirty-bytes}} \tag{6.4}$$

For the remaining part of this study, the default value of W is 128, meaning that there are 128 write accesses to global variables between consecutive checkpoints in average. Assuming that DICE's topmost stack management is efficient, $S_{\text{topmost-stack}}$ is set to 16 words. Again assuming DICE's efficiency, t_{record} and t_{bit} are respectively set to 1 µs and $\frac{1}{24}$ µs, exposing DICE's best performance. Also, for comparison purposes, α^* is set equal to α.

6.3.2 Comparison with Other Approaches

In this section, the benefits of the present proposal are evaluated using the analytical model provided in the previous section. Specifically, Equation (6.2) is studied against Equation (6.1), Equation (6.3) and Equation (6.4), by exploring their behaviors when parameters change. The parameters that are not indicated in the following figures have default values mentioned in Table 6.2. These values have been obtained when measuring the properties of the MSP430FR5739 micro-controller, as described in Section 5.5. The precision of these values is not very important here. What is important is the aspect of the evolution of energy consumption when some parameter, *e.g.*, RAM size, changes.



(a) $N_{\text{region}} = 16$, $S_{\text{min}} = 312$ (b) $N_{\text{region}} = 64$, $S_{\text{min}} = 912$

Figure 6.2: Energy consumption of the checkpointing mechanisms with respect to size of RAM for different values of N_{region}.

Impact of RAM size Figure 6.2 shows the energy consumption with respect to the RAM size for different values of N_{region}. The first intuition is confirmed, the larger the amount of RAM used is, the better is this proposal. Another observation from Figure 6.2 is that the MPU-based incremental checkpointing is not always the most efficient with an increase of the amount of regions. This phenomenon is detailed and discussed in the next section. The plots from Figure 6.2 keep the same features when making S_{words} vary. This gives a hint about the conditions that make the MPU-based incremental checkpointing consume less energy than the other approaches, as studied hereafter.

When comparing to the full copy baseline, the MPU-based approach is less energy-consuming when the RAM size is greater than a certain value $S_{\text{min/full-copy}}$, . The definition of $S_{\text{min/full-copy}}$ stands only when the relationship of Equation (6.5) is true:

$$P_{\text{MPU}} \times N_{\text{dirty}} < P_{\{\text{DMA, plat}\}} \times (N_{\text{region}} - N_{\text{dirty}}) \tag{6.5}$$

Equation (6.5) illustrates the fact that the energy overhead due to using the MPU must be lower than the energy saved by *not* copying the $N_{\text{region}} - N_{\text{dirty}}$ unmodified regions. Provided that Equation (6.5) stands, then:

$$\forall S_{\text{words}} > S_{\text{min/full-copy}}, E_{\text{MPU-incremental}}(S_{\text{words}}) < E_{\text{full-copy}}(S_{\text{words}})$$

In other terms, $S_{\text{min/full-copy}}$ is the minimal amount of RAM words that makes the incremental checkpointing worth using in comparison to the classical full copy. Note that $S_{\text{min/full-copy}}$ actually depends on the other parameters as listed in Table 6.2, as explicitly stated in Equation (6.6).

$$S_{\text{min/full-copy}} = \frac{f_{\text{DMA}} \times [(\frac{t_{\text{overhead}}}{N_{\text{dirty}}} + t_{\text{int}}) \times P_{\{\text{CPU, MPU, plat}\}} + E_{\text{critical}}] + \Delta_R \times P_{\{\text{DMA, MPU, plat}\}}}{\frac{P_{\{\text{DMA, plat}\}}}{N_{\text{dirty}}} - \frac{P_{\{\text{DMA, MPU, plat}\}}}{N_{\text{region}}}} \tag{6.6}$$

Where $\Delta_R = R_{\text{words}} - \frac{S_{\text{words}}}{N_{\text{region}}}$, *i.e.*, the remaining part of word due to the ceil operation that defines R_{words}. As a matter of fact, the value of Δ_R lies between 0 and 1 and a fair estimation of $S_{\text{min/full-copy}}$ may consider Δ_R being 0 or 1, with a low impact on the result, compared to the memory demands of realistic applications that are orders of magnitude higher. In practice, $S_{\text{min/full-copy}}$ may also be determined using a dichotomic reasoning as well, between the values given by Equation (6.6) with $\Delta_R = 0$ and $\Delta_R = 1$.

When comparing the MPU-based approach to the hash- or sweep-based methods, the observations are the same. However, since the hash- and sweep-based methods are more optimized than the full-copy, the MPU-based approach is best only when the RAM size exceeds a greater value, named $S_{\text{min/hash}}$. The

definition of $S_{\text{min/hash}}$ stands only if the relationship stated in Equation (6.7) is true:

$$N_{\text{dirty}} \times \frac{P_{\text{MPU}}}{f_{\text{DMA}}} < N_{\text{region}} \times t_{\text{hash-word}} \times P_{\{\text{CPU, plat}\}} \tag{6.7}$$

Equation (6.7) materializes the fact that the energy overhead of enabling the MPU must be lower than the energy consumed by hashing the entire memory. Then, analytically, $S_{\text{min/hash}}$ follows Equation (6.8).

$$S_{\text{min/hash}} = N_{\text{dirty}} \times \frac{(t_{\text{int}} + \frac{t_{\text{overhead}}}{N_{\text{dirty}}}) \times P_{\{\text{CPU, MPU, plat}\}} + E_{\text{critical}} + \Delta_R \times \frac{P_{\text{MPU}}}{f_{\text{DMA}}}}{t_{\text{hash-word}} \times P_{\{\text{CPU, plat}\}} - \frac{N_{\text{dirty}} \times P_{\text{MPU}}}{N_{\text{reg}} \times f_{\text{DMA}}}} \tag{6.8}$$

As with $S_{\text{min/full-copy}}$, $S_{\text{min/hash}}$ may be obtained using a dichotomic approach between the boundaries set by setting Δ_R to 0 and to 1.

In Figure 6.2, the MPU-based approach is always less energy-consuming than DICE. Actually, their graphs cross at a large negative value of S_{words}. The condition for $E_{\text{MPU-incremental}}$ to be lesser than E_{DICE}, is given, considering that $\alpha^* = \alpha$, in Equation (6.9).

$$\frac{N_{\text{dirty}}}{N_{\text{region}}} \times \frac{P_{\{\text{CPU, MPU, plat}\}}}{f_{\text{DMA}}} < \alpha \times \frac{P_{\{\text{DMA, plat}\}}}{f_{\text{DMA}}} + \frac{N_{\text{bytes-per-word}}}{8} \times t_{\text{bit}} \times P_{\{\text{CPU, plat}\}} \tag{6.9}$$

Equation (6.9) illustrates the trade-off between copying dirty regions with the MPU on one hand, and copying dirty bytes and managing the bitmap on the other hand. The minimal RAM size that makes the MPU-based approach beneficial over DICE, $S_{\text{min/DICE}}$, could be defined, however in the case of Figure 6.2, the MPU-based approach is always better using the default values of the other parameters.

Finally, a global threshold, named S_{min} can be defined as the least amount of RAM required to make the MPU-based approach better than all the other studied approaches. As a consequence :

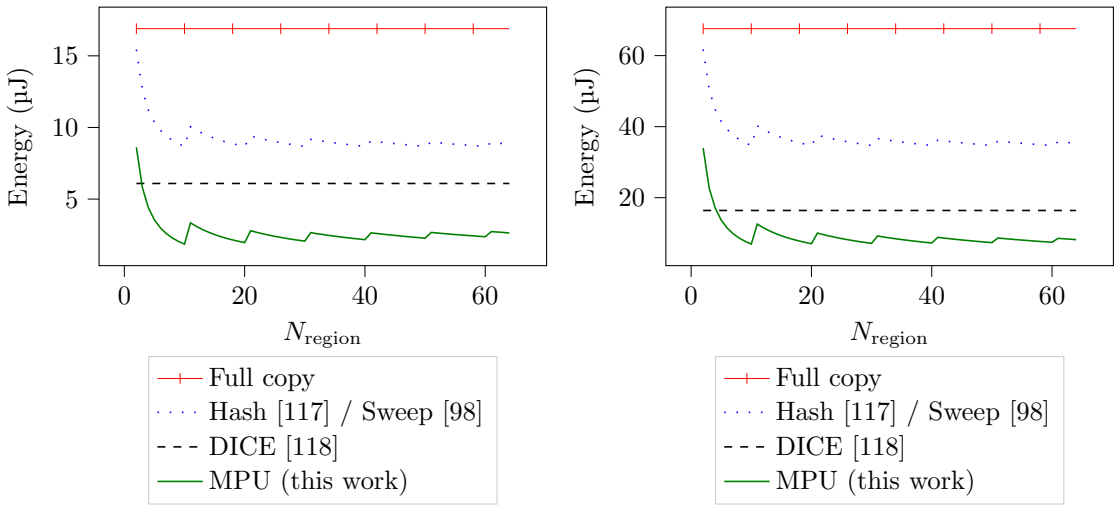
$$S_{\text{min}} = \max\{S_{\text{min/full-copy}}, S_{\text{min/hash}}\}$$

In Figure 6.2, the values of S_{min} are fairly low, less than 2 kB, which implies that the MPU-based approach consumes less energy than the other approaches for the needs of realistic applications. The slopes of the graphs also show that DICE and the MPU-based approach are more scalable than the full-copy and the hash-based approaches, *i.e.*, increasing RAM demands increases the checkpointing energy, but to a lower extent. However, it is necessary to mitigate these results by taking into account the influence of other parameters, as discussed further below.

Impact of the amount of regions If the regions are too numerous, the platform spends a long time handling MPU interrupts. But if N_{region} is too small, the checkpointing is less incremental and resembles more the classical full-copy with detrimental overhead. This is illustrated by Figure 6.3: for the considered amounts of RAM, the optimal amount of regions is below ten. In the 16 kB RAM scenario, the MPU-based approach needs at least three regions to be beneficial over DICE and in the 64 kB scenario, it requires five regions. With N_{region} increasing, the oscillations get weaker but the overall energy consumption of the MPU-based mechanisms increases as well, for the run-time is likelier to be interrupted more often. The worst case scenario would be that the application attempts to write to variables that are located in different regions, so the MPU interrupt handlers prevent the application from making progress.

Impact of other parameters The model provided here does involve eleven parameters that interact with each other in a complex manner. Figure 6.4 proposes to look at other parameters while exploring away from the default values mentioned in Table 6.2.

Figure 6.4a confirms that a high α decreases the efficiency of incremental checkpointing. Indeed, when α grows towards 1, the system has to copy an amount of data that becomes closer to the amount of data required by the classical full copy. In that case, the overhead due to MPU interrupt handling makes it a challenge, for the MPU-based incremental checkpointing, to keep being beneficial. However, even with stable energy harvesters that are able to run the platform longer and thus to achieve a greater α such as solar-based harvesters, the incremental checkpointing requires only a few thousands of words to show better performance.



(a) 16 kB RAM. (b) 64 kB RAM.

Figure 6.3: Energy consumption of the checkpointing mechanisms with respect to the amount of MPU regions for different RAM sizes.

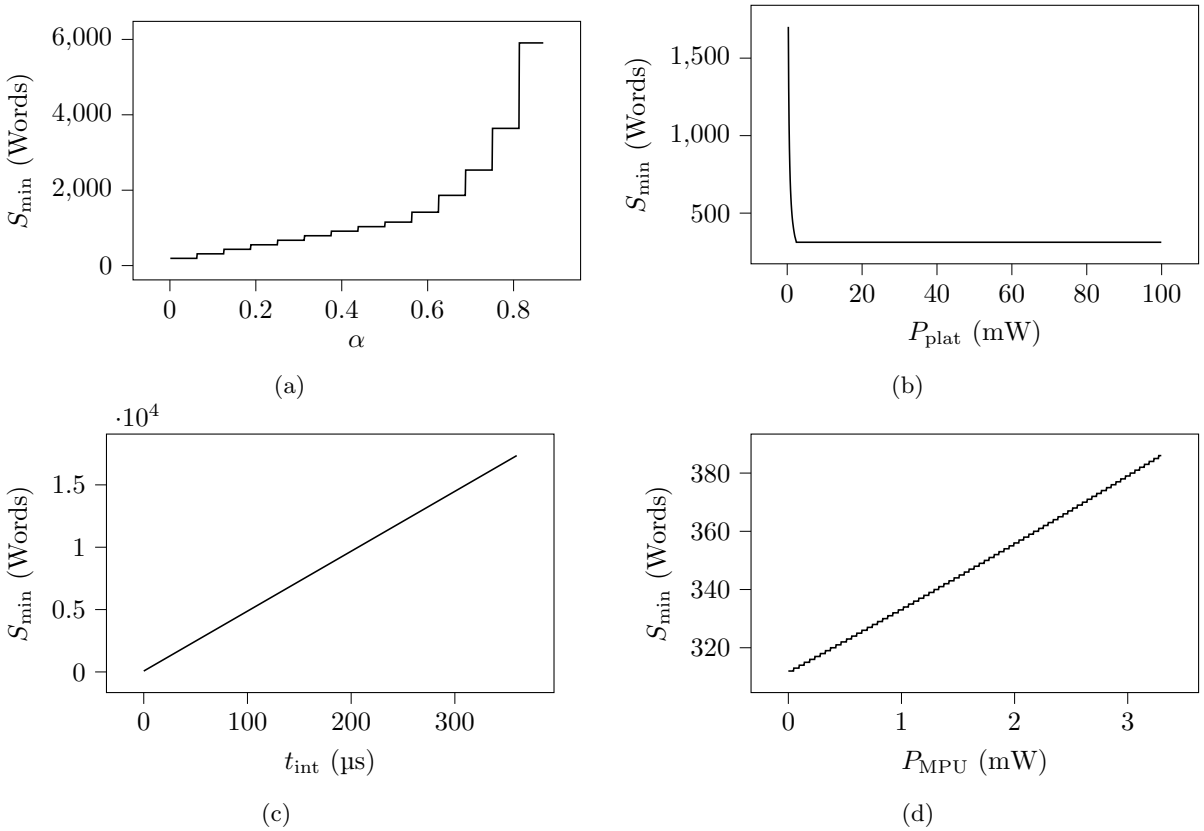


Figure 6.4: Impact of (a) α, (b) P_{plat}, (c) t_{int} and (d) P_{MPU} on S_{min}, the minimal amount of RAM words to make the MPU-based incremental checkpointing worthwhile over the other approaches.

Figure 6.4b shows that the lesser P_{plat} is, the less efficient the incremental checkpointing is. This observation can be directly derived from Equation (6.5) and Equation (6.7). Decreasing P_{plat} applies a greater pressure on P_{MPU}, which is likelier to be less efficient. In practice, this means that the time saved by using the incremental checkpointing saves a greater amount of energy if the platform is consuming a lot of power when performing the save operation, as one could expect.

Figure 6.4c confirms the fact that the execution time of the MPU interrupt handler is crucial. A longer execution time means a higher energy budget allocated for checkpointing, since the micro-controller and the peripherals are consuming power in the meantime.

The electronic properties of the MPU can differ from a micro-controller to another and its configuration may have an impact on its consumption. This motivates the need to study the influence of P_{MPU} on the performance of the incremental checkpointing. Figure 6.4d shows that, if the MPU consumes more, $S_{\min}$ increases accordingly, meaning that there must be a greater amount of memory to be made persistent in order to keep the incremental checkpointing beneficial. However, $S_{\min}$ values are still low, even when P_{MPU} is high. Thus, even a complex, power-consuming MPU would not hinder the benefits of the MPU-based incremental checkpointing.

Some extreme values were shown on purpose in Figure 6.4, in order to emphasize the low yet realistic requirements that make this proposal efficient. For instance, t_{int} is not expected to be greater than a few microseconds, however Figure 6.4c shows that greater values would require a greater $S_{\min}$ but still realistic from the application's perspective. Another example is when α is very high in Figure 6.4a. In practice, the application is not expected to modify most of the memory contents before a power outage occurs, yet greater values of α would require a realistic value of $S_{\min}$. In any case, $S_{\min}$ is always low, making the incremental checkpointing often worth using over a classical full copy.

6.4 Validation through Simulation

In this section, the analytical results are validated thanks to a simulation platform. One limitation of the analytical approach is that α is always the same whereas it may change from one life-cycle to another. In cycle-accurate simulation, as in reality, α changes every life-cycle. Another limitation is that P_{plat} is not constant throughout the life-time of the application. In order to simplify the model, an average P_{plat} was considered, but the simulator computes the exact instantaneous power consumption, depending on the state of the platform and peripherals at any time. This section focuses on a comparison between the MPU-based approach and the full-copy baseline.

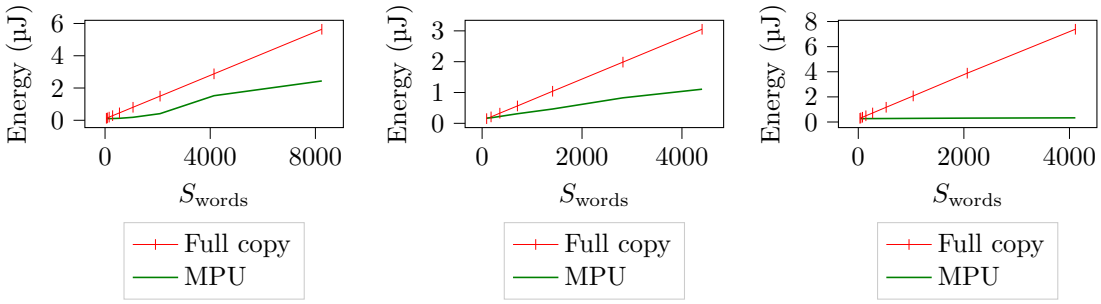
6.4.1 Simulation Platform

The simulator used to yield the experimental results is the one depicted in Section 5.6. It thus simulates an MSP430FR5739 micro-controller with some peripherals. For the needs of this very work, the implementation of the MPU in the simulator was modified to handle up to sixteen regions instead of three, as well as to cover volatile RAM address range instead of solely the non-volatile RAM address range. This new MPU is then compliant with the MPU requirements listed in Section 6.2.2 and mimics the features of ARM Cortex-M MPUs. The memory capacity of the platform was also virtually modified in order to propose 20 kB volatile RAM and 40 kB non-volatile RAM. [1] The rest of the virtual platform is left untouched.

6.4.2 Benchmark Applications

The simulation is tested against home-made benchmarks for no unified benchmark suite for transiently-powered systems using peripherals exists yet. The benchmark consists of a few applications, some of them being more computational while the rest is more focused on peripheral usage. Computational applications are investigated here as well, in order to truly evaluate this proposal against sundry application demands, to see the behavior of this proposal with computational and memory-intensive applications and peripheral-intensive applications.

[1] `https://github.com/gberthou/archc-msp430x/tree/wider-memory`



(a) Quicksort
$\alpha = 0.152$, $N_{\text{region}} = 16$

(b) RSA
$\alpha = 0.272$, $N_{\text{region}} = 16$

(c) Complete WSN
$\alpha = 0.012$, $N_{\text{region}} = 16$

Figure 6.5: Checkpointing energy consumption with respect to size of RAM for different applications, when the energy budget is 120 µJ.

Quicksort Quicksort initializes and sorts an array of pseudo-random data. The array size is statically defined. The initialization phase modifies the entire array in descending address order, meaning that α is meant to be rather high at the beginning of the application, before the actual sort algorithm runs.

RSA RSA initializes and performs an RSA encryption onto an array of data. The array size is statically defined.

Complete-WSN Complete-WSN uses accelerometer, temperature sensor and radio. It is the same benchmark application that is described in Section 5.6.5.

All applications are declined into several instances, one instance per combination of S_{words} and N_{region} values if the MPU is enabled, one instance per value of S_{words} for the full copy version without MPU. S_{words} is made to vary by changing, at compile-time, the static array sizes in order to obtain the desired volatile RAM utilization. Each instance has its own memory access signature, that impacts α. However, α is also impacted by the available energy within a single life-cycle, *i.e.*, the weaker the life-cycle, the lower is expected to be α. The experiments made also vary the available energy. This experimental methodology generates hundreds of instances, *i.e.*, hundreds of executable binary images, as well as dozens run-time scenarios dependent on the available energy. Combined altogether, there are thousands experiments to conduct, hence another motivation for simulation over actual measurements. In all the results presented here, the available life-cycle energy is 120 µJ.

6.4.3 Checkpointing Layer

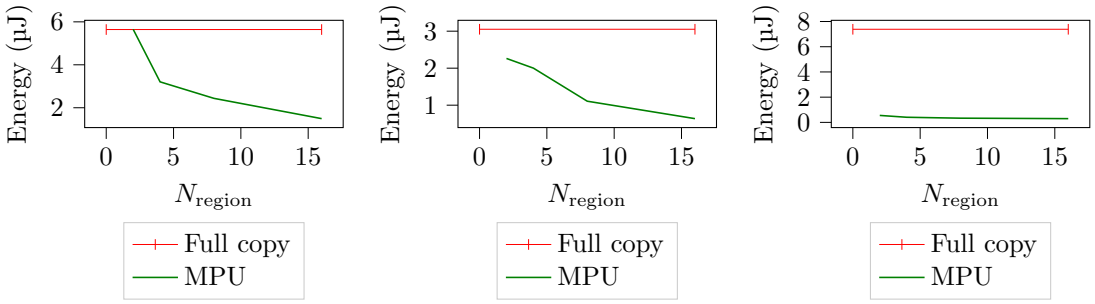
The aforementioned applications run on top of Sytare, as depicted in Chapter 4. Within the context of this work, the mechanism that makes the volatile RAM persist in non-volatile RAM was slightly modified, by integrating MPU information and selectively copying from volatile RAM to non-volatile RAM based on that information. [2] The results shown for the full copy correspond to the original version of Sytare.

6.4.4 Validation of Analytical Results

Given the experimental setup described in the previous section, the actual performance of this MPU-based incremental checkpointing proposal is evaluated here against the simulation platform. The displayed α values are computed as the average α of the first 32 life-cycles for all executions involved in a given graph.

Impact of RAM size Figure 6.5 shows how the needs in RAM impact the energy required for checkpointing for the benchmark applications. The results are similar to the anticipated values given by the model and shown in Figure 6.2. The major discrepancy is the fact that, in practice, α is not a constant.

[2] `https://gitlab.inria.fr/citi-lab/sytare/-/tree/ex-mpu`



(a) Quicksort, 16.1 kB (b) RSA, 8.6 kB (c) Complete WSN, 8 kB

Figure 6.6: Checkpointing energy consumption with respect to the amount of MPU regions for different applications, when the energy budget is 120 µJ.

Indeed, depending on where the application resumes and on the energy budget of the next life-cycle, the application does not require the same regions, nor the same amount of regions, to be made persistent for the next life-cycle. The results show that, when the application uses more than a hundred RAM words ($S_{\text{min/full-copy}}$ is around 100), it is always better to use the incremental checkpointing rather than full RAM copy.

Impact of the amount of regions Figure 6.6 shows how the amount of MPU regions impacts the energy required for checkpointing for the benchmark applications. Analytical results showed that incremental checkpointing is always better than a full RAM copy, when α is low enough. To this extent, simulation results are alike. The visible steps of Figure 6.3 do not appear in Figure 6.6 for the amount of regions was made to vary at fine grain in Figure 6.3, unlike along powers of two in Figure 6.6.

6.5 Validation on a Real Platform

The MPU-based incremental checkpointing proposal was implemented, not only in simulation, but also on a real platform, namely ARMorik. For today's ARM micro-controllers are not yet equipped with non-volatile RAM, ARMorik uses an external parallel non-volatile memory chip, hence it consumes significantly more energy than if the memory is embedded inside the micro-controller's chip. Energy measurements on the ARMorik platform would thus not be realistic nor meaningful and, as a result, the implementation on the ARMorik platform is genuinely for qualitative appreciation. The MPU of the ARM Cortex-M7 fits perfectly the requirements of this solution, which makes the proposal viable on low-power micro-controllers, although more elaborated than an MSP430FR5739 micro-controller.

6.6 Conclusion

The MPU-based incremental checkpointing uses an MPU to make portions of memory read-only on boot and keep track of their modifications, throughout a life-cycle, using the access mismatch exception. The model, as well as the experimental results in simulation, give the MPU-based approach the lowest energy requirements in comparison to a classical full copy, to a hash performed in software and to a systematic instrumentation of the write accesses. The present proposal is realistic in terms of feasibility and energy requirements, even for transiently-powered systems, although the MPU needs to be elaborated enough. It was implemented on the ARMorik platform and lately, a similar approach leveraging MPU exception was adopted on a battery-free Game Boy [28], thus corroborating the adequation of such a solution on transiently-powered systems. The conditions under which MPU-based incremental checkpointing is better than an unselective copy are realistic today, and they will be even more realistic in the future as the amount of embedded RAM is likely to increase. However, the checkpointing time and energy depend on the amount of dirty regions, and thus the checkpointing system loses part of its determinism.

The novel model exposed in this chapter enables reasoning on the performance of any checkpointing scheme, using a handful of parameters. Its limitations include the fact that, in reality, the dirtiness ratio and the power drawn by the platform are not constants but values that change over time.

Also, it may be noticed that the duration of a life-cycle, which depends on the energy storage, is not important to appreciate the benefits of incremental checkpointing. It only has an impact on the amount of dirty regions. With shorter life-cycles, there would be fewer dirty regions as fewer write instructions would be executed. This work targets rather small systems, that cannot be powered by solar panels, nor embed a big capacitor, which is the case for many applications. It would be interesting to extensively study the amount of dirty regions with regard to the power source, the energy storage and the application needs, but this is left for future work.

This work currently proposes to selectively save memory regions based on whether they were modified or not. However, the restoration scheme is the same: all memory regions are restored. It is possible to extend the concept of this work to restoration as well. A lazy restoration policy would not restore the RAM on boot, apart from the required stack to make the application resumable. Instead, the memory regions can be read-protected in addition to write-protected. On the first read attempt to a given region, the MPU handles the exception, restores the matching region and grants read access to the region. The write accesses would be addressed independently of the read accesses, in the very same manner than depicted in this chapter.

Chapter 7

Formal Proof of Checkpointing-Based Intermittent Systems

The literature about operating systems for transiently-powered systems contains many solutions that propose interesting mechanisms to enable the execution of long-running applications despite unpredicted power outages, however their properties are not formally demonstrated. The papers informally state that their proposals work, at best providing insights of proofs or using some application benchmarks [84, 66, 67, 77]. Some works model their solutions more formally and expose a central theorem [58]. Yet it is a single block in the logic and the proof must be integrated into a larger picture.

A formal proof of such systems would decouple the proposal from its implementation. Indeed, an implementation bug should not make a valid proposal fail and conversely, observing an expected execution of several benchmark applications does not prove the concept and the mechanisms behind under any situation.

The present chapter briefly summarizes the first work that proposes a comprehensive and peripheral-aware model of transiently-powered systems equipped with non-volatile memory [123]. The published work provides a high-level model of application specification and run-time execution. It then truly proves that the principles some operating systems rely on allow the correct execution of long-running applications across power outages. Ironically enough, the objective of this chapter is not, in its essence, to exhaustively formalize the problem and its solutions, but is rather a bare introduction to the published work that soundly defines each entity and conducts all formal proofs.

Memory and CPU volatility (P1) is assumed to be correctly handled by existing operating systems, for it is achievable through basic copies from and to non-volatile memory. Consistency between saved memory and CPU states is to be carefully looked over, however it is the matter of a few picked assembly instructions. The objective of this work is not to prove assembly code nor to mistrust the persistence properties of non-volatile technologies. As a result, this work leaves P1 aside and focuses on the correctness of an application with respect to problems P2 and P3, that involve peripherals and roll-backs required for correct execution of peripheral operations under intermittent power. A later publication of Surbatovich *et al.* [124] targets a formal proof of the correct execution of non-idempotent accesses to non-volatile memory (P4). The contribution of this thesis within the context of this chapter, apart from bringing the topic to the table, was the informal specification of the intermittent run-time and of an operating system through the example of Sytare. The remainder of the contribution, including the formal model, Coq development [1] and proof design, was performed by the co-authors for their expertise made them the most competent to perform these tasks.

Section 7.1 exposes a glimpse of formal models of systems and parts of systems, as well as recent advances in models and proofs that integrate crash into their designs. Section 7.2 is a quick recapitulation of the problems brought by intermittent power as initially studied in Section 2.6. Then, Section 7.3 defines the model of the specification and of a run-time environment with power outages. Section 7.4

[1] `https://gabertho.gitlabpages.inria.fr/icp-model/`

informally initiates the reasoning behind the proof that execution traces under intermittent power are refinements of the specification as long as there exists a checkpointing mechanism that tackles P1, P2 and P3. Finally, Section 7.5 discusses the adequation of existing system layers for transiently-powered systems, including Sytare developed in this work and presented in Chapter 4, with respect to the present model and Section 7.6 concludes that topic.

7.1 Proving Systems

Proving properties of operating systems is not an easy task. Multi-threaded systems complicate the exercise [125, 126] but fortunately enough, transiently-powered systems are not as sophisticated. Non-volatile memories are common examples of proven systems for their states are, by definition, persistent thus any error can be propagated over time and is likely to cause an error in the future.

Hoare logic Hoare logic leverages reasoning about what predicates hold before and after a program is executed. The notation {precondition} S {postcondition} states that, if precondition holds, the program S can run and upon completion, postcondition then holds. Chen *et al.* use Hoare logic to certify a Unix-like file-system with logging [127]. They further extend that work by creating an actual certified file-system, FSCQ [128], that is machine-checkable using the Coq proof assistant. It is based on Crash Hoare Logic, a variant from the Hoare logic. The motivation behind the extension of Hoare logic is that it natively does not support crash. Within Hoare logic, a crash would interrupt a program, which thus may no longer guarantee that the postcondition is settled. In addition, it does not support recovery that would involve a specific recovery procedure. Crash Hoare Logic, on the other hand, proposes semantics for crash conditions and recovery execution.

Linearizability Linearizability involves to reason about a trace of invocations and their response events [129]. Distinction can be made between sequential and concurrent traces. Sequential traces require their invocations to be directly followed by their response, whereas concurrent traces do not follow that rule. Straightforward concurrency can be observed in a multi-threaded environment, however it is also present in single-threaded embedded systems when initiating asynchronous peripheral operations. Linearizability properties can be extended to integrate crashes, for instance with the introduction of the concept of durable linearizability [130, 126]. Crash events complement the set of available events in a trace. The notion of durable linearizability, as defined in [130], does account for operations that leave an observable mark before the crash occurred. Buffered durable linearizability expresses that the state after the crash must be consistent, but not necessarily up-to-date. However, that logic does not formalize the timeliness constraints. The application resumes, after the crash has been handled, and the trace follows its intended flow.

Domain-Specific Language Domain-Specific Languages (DSLs) are designed to model a high-level interface of a system. The system is described using a state machine and each of the possible operations is modeled regarding its impacts on the system state. The aim of solutions based on DSLs is to establish a bisimulation between two programs, one being a transformed version of the other. The reference program is the specification while the one to be proven a simulation of the former is the actual implementation or any execution of that implementation. The transitions between states come along with a sequence of observable effects and the sequence of observable effects must be rigorously the same between the specification and the implementation. CompCert [131] is a formally-verified compiler for a subset of the C programming language that shines by using the Coq proof assistant both for compiler programming and correctness proof. CertiKOS [132] enables the construction of certified concurrent operating systems. An example of such kernels, mC2, is mostly compiled with CompCert to that extent. Perennial [125] verifies crash-safe systems in a concurrent multi-threaded context. It defines correctness as a refinement between the code of an application and its specification. Perennial defines a DSL that, alongside a proof of linearizability of concurrent accesses to a shared resource, enables the refinement to be established. It is complemented with Hoare logic for each operation. Surbatovich *et al.* [124] also rely on a DSL. They center their analysis on architectures such as A2 and A4 with a memory organization M2 where variables may be allocated directly in non-volatile memory. More specifically, they target issue P4 that arises when

non-idempotent accesses to non-volatile variables are interrupted by a power outage, including changes in control-flow due to peripheral operations not necessarily returning the same value before and after the power outage occurred. They assume a static checkpointing scheme that is complementary to the assumption of just-in-time checkpointing of this chapter, as stated hereafter. ELEVEN82 [133] analyzes C code with the intent to detect recoverability bugs or prove their absence. The key idea behind that work is that, after a program has recovered from crash, its state must simulate a valid state that was observed before the crash. Considering traces, the recovery operation must thus be able to reset the system state to a prior point in the trace. ELEVEN82 illustrates the concept of crash with an external event that nullifies volatile state but preserves non-volatile state, which is precisely the kind of crash targeted by this thesis. The main objective behind ELEVEN82 is to show whether a bisimulation between the trace of the program with crashes and the trace of the program without crash can be established.

As seen in the former works, these techniques are not disjoint but quite combinable. For instance, Hoare logic can be integrated into DSLs [125]. It is also common to use DSLs with linear trace analysis, which is leveraged by the model described in this chapter. Peripheral operations are inherent parts of that language for they may modify the state of the platform. A refinement between the specification and the crash-aware model is proven in [123].

7.2 Challenges of Intermittence

A comprehensive study of challenges brought by intermittent power is exposed in Section 2.6. The volatility of the memory and of computational parts of the platform, P1, is left aside for it is assumed to be well handled by operating systems. The volatility of peripheral states, P2, is at stake for they are reset upon reboot, thus breaking the assumptions of the software, initially designed to be run sequentially with former peripheral calls having effective consequences on run-time. The timeliness constraint, P3, requires sections of the application to be run in a continuous time-span. If the remaining energy does not enable the platform to complete a section of code with timeliness constraint, the entire section is intended to be attempted again from the beginning, when the energy storage permits it. The usual solutions to P3 in the literature either maintain a finite or infinite log of peripheral operations or roll-back the control-flow of the application to the beginning of the section with timeliness constraint. Either way, the control-flow is rewound in time, which also breaks the assumptions of the software, since those transitions do not correspond to explicit control-flow operations allowed by programming languages such as calling a function, loop statements, *etc.* The unexpected rewinding potentially creates a failure in non-idempotent environments, which is the core of P4. A non-idempotent environment includes actions that cannot be undone, such as physically moving a mechanical device, or writing to non-volatile memory.

7.3 Model and Definitions

The present model leverages a certain amount of assumptions. The main assumption is that the non-volatile memory is only used as a persistent storage and cannot be used as a working memory. It corresponds to the architecture A1 and the memory organization M1a, that are leveraged by systems such as Mementos [21], Hibernus [22, 86], HarvOS [85], RESTOP [84], KARMA [67], MPatch [28], and Sytare [66]. Variables are only stored in SRAM. This assumption wipes out P4, but the specific proof of operating systems that cope with P4 has been studied [124]. Furthermore, P1 is considered solved. This chapter can thus plainly focus on correctness regarding peripherals. In addition, the environment in which evolve the peripherals is assumed to be idempotent. Network layers handle repeated packets, so a network-related peripheral does fall into that category. But a motor does not, for its movement is not automatically undone. A just-in-time checkpointing, with an imminent power outage interrupt, is assumed, to only observe one checkpoint per life-cycle, but the model and proof can be easily adapted to encompass the entire spectrum of checkpoint mechanisms. What matters is that a checkpointing mechanism is assumed. However, this model does not assume that the checkpointing operation always succeeds. For instance, if the imminent power outage energy threshold is dynamically updated, this model encompasses the cases where the remaining energy is so low that the checkpointing operation cannot complete before the platform actually runs out of power.

In the context of intermittent computing, correctness of a particular execution can be defined as the execution effectively following a given specification. That specification is the behavior of an application under continuous supply. Since P1 is assumed to be solved and P4 is not considered in this chapter, the present specification is tailored to encompass both P2 and P3, *i.e.*, peripheral state volatility and timeliness constraints. Hence, the specification is entirely correlated with peripherals but not with software concerns such as the state of the volatile memory. As a consequence, the specification is a trace of peripheral operations, an ordered sequence. It is to be noted that a matching trace also testifies that P1 is properly handled for later peripheral operations can be reached only if the software logic is properly made both persistent and consistent with peripherals. Aside from the trace itself, correctness also implies that timely operations are executed within a single life-cycle, or retried from the beginning otherwise.

7.3.1 Entity Definitions

Micro-controller The notion of micro-controller here encompasses the register file and volatile RAM. Peripherals are defined after. Non-volatile memory is simply modeled as a piece of memory with data retention properties when the platform is no longer powered. Non-volatile memory as working memory is not supported by this model and is left for future work, as addressed in [124].

Peripherals The peripherals are considered to have one or several states. Their states may be modified only upon the execution of *peripheral operations*. This model roughly resembles the one depicted in Chapter 5, without the energy-related concerns and valid at any granularity, from a high-level API to a low-level per control register grain.

7.3.2 Specification Under Continuous Power

The specification has two distinct modes: *user* mode and *driver* mode. User mode depicts computational code that may be interrupted and resumed at any point within its inner control-flow graph, *i.e.*, without timeliness constraint. Driver mode is the complementary state, *i.e.*, code that must be time-consistent. Although it is called driver mode, the specification stands for any time-consistent code, even code that would mix peripheral operations and computations to guarantee a certain time-consistency between sampled sensor values and the transmission of some data related to the samples. In the specific case of correctness, *i.e.*, ensuring that the run-time environment is not broken, such timeliness constraints are rather *soft* while peripheral timeliness constraints such as waiting for an oscillator to stabilize are *hard* for failing the latter would impede any further execution while the former would only consider outdated data but not threaten the application logic.

Hence, the present model imposes a certain amount of rules or axioms. Code in user mode cannot interact with peripherals and only code in driver mode can. This corresponds to the common separation between application code and driver code, through a well-designed API for instance as discussed in Section 3.2.1. Changing from either mode to the other is performed using an *enter* operation and a *leave* operation that respectively enters and leaves driver mode. Unlike the wrapper entry and exit defined in Chapter 4, the enter and leave operations are purely symbolical and do not correspond to an actual change in system state. Indeed, the instructions involved in the wrapper entry and exit may be merged with the driver call code so that the system state effectively transitions from user to driver state abruptly. Figure 7.1 illustrates the concept of specification under continuous power, using a two-state automaton that fully describes the run-time under continuous power. The two states are `User` and `Driver`, and four types of operations, *i.e.*, transitions, exist. A `USR` operation works in `User` state and does not change the state. It corresponds to some computation, making the application progress but without accessing

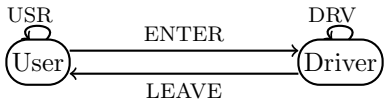


Figure 7.1: State machine of the system under continuous power. The `Driver` operations are the only operations to be accounted into the specification.

the peripherals. Its driver counterpart is the `DRV` operation that works in `Driver` state. It corresponds to a peripheral operation, such as the modification of a control register. Finally, the `ENTER` and `LEAVE` operations, as already defined, are the only operations capable of changing the system state.

Recalling that `ENTER` and `LEAVE` are symbolical and that `USR` operations cannot access peripherals, the only kind of operations that appear in the specification trace are the `DRV` operations, for the specification only accounts for the changes in peripheral state. For clarity purposes, an instance of specification is noted as a semicolon-separated sequence of peripheral operations, *i.e.*, $\mathrm{op}_1; \mathrm{op}_2; ...; \mathrm{op}_n$ where each op_i is a peripheral operation.

7.3.3 Execution Trace Under Intermittent Power

The introduction of intermittent power brings its challenges, discussed in Chapter 2, and their solutions, discussed in Chapter 3 and Chapter 4. This chapter assumes a just-in-time checkpointing system, that triggers an explicit checkpointing operation upon the detection of an imminent power outage at run-time. The state machine of the system under intermittent power is based on the one of the continuous power, originally depicted in Figure 7.1. One difference, that actually does not change the state automaton, is that now the `ENTER` and `LEAVE` transitions are respectively recorded into the execution trace as E and L events, while they do not appear in the specification. This being stated, new states and transitions complement the state automaton.

The checkpointing operation is performed in a new system state called `Power`. The imminent power outage interrupt is enabled during the `User` and `Driver` states, meaning that there is a transition from those states to `Power`. In a sunnybright scenario, the checkpoint succeeds and the system goes from state `Power` to state `Off-OK`. If the checkpointing operation fails, for instance because the energy level threshold of the imminent power outage interrupt has been dynamically set to an insufficient value, the system state enters the `Off-KO` state. The distinction between the `Off-OK` and `Off-KO` states is made here to ease the understanding, whereas both states were merged in the original publication. In terms of hardware state, there is a single off state where supply voltage is not high enough to power on the components. In addition, the imminent power outage interrupt may simply not be fired at all, for instance if the interrupts were disabled by the application for some reason or if the interrupt threshold was set too low. In that case, the platform eventually runs out of power without even attempting a checkpointing operation. There thus exist transitions from the `User` and `Driver` states to the `Off-KO` states. Upon reboot, the last valid checkpoint is always restored. If the system stopped in `Off-OK` state, the last valid checkpoint corresponds to the latest forward progress. On the other hand, if the system stopped in `Off-KO` state, the operating system layer restores the same checkpoint as in the previous life-cycle, hence cancelling all partial progress that has been done during the previous life-cycle. This can be easily implemented as a double-buffer, as some existing solutions already do. Finally, an optional `Init` state materializes all operations needed by the operating system during the very first boot of the platform. These operations may include the initialization of non-volatile memory and of internal variables of the operating system. However, as demonstrated in Section 4.7, the existence of that `Init` state is implementation-related and is not actually needed if the programming toolchain is able to properly populate the non-volatile memory with a valid checkpoint. Figure 7.2 shows the state machine of the system under intermittent power, as an augmentation of the automaton from Figure 7.1.

The *execution trace*, as opposed to the *specification*, is not only limited to the peripheral operations. An execution trace also records the events of successful and failed checkpoints. A checkpointing operation is successful if and only if the system successively takes an energy interrupt transition and a checkpoint success transition, as shown in Figure 7.2. In other terms, a checkpointing operation is successful when the system reaches the `Off-OK` state. On the contrary, a checkpointing operation is not successful when the system reaches the `Off-KO` state. Several paths lead to that failed state. The platform can either fail to trigger the imminent power outage interrupt, which automatically leads to the `Off-KO` state. Or the imminent power outage interrupt handler can lack energy to complete. Regardless of the path taken to reach either of the `Off-OK` and `Off-KO` states, the definition of execution trace requires the definition of two events: (*i*) a successful checkpoint denoted Chkpt✓ and (*ii*) a failed checkpoint denoted Chkpt✗. An execution trace thus consists of sequences of peripheral operations bounded by E and L events, plus Chkpt✓ events and Chkpt✗ events. It may also contain unfinished sequences of peripheral operations, starting by an E event and ending with either a Chkpt✓ event or a Chkpt✗ event, in case they could not

complete until a power outage occurred.

7.3.4 Differences With the Original Model

The model presented in this chapter differs from that of [123]. First, the state automaton of the system under intermittent power makes a clear distinction between the `Off-OK` and `Off-KO` states whereas they were originally grouped together into a single state. Here the choice was made to separate them so that the states themselves carry the information on whether the checkpointing operations could or could not be performed before power outage. Also, to limit the amount of separate transitions, a single energy interrupt transitions from both `User` and `Driver` states to `Power` state. Similarly, a single actual power outage transitions from both `User` and `Driver` states to `Off-KO` state. The resulting model is simpler, but the transitions do no longer carry the information of the initial state. Note that the model does not lose its ability to grasp the non-determinism of the success of the checkpointing operation, for the transitions available from `Power` state are taken non-deterministically.

Another main difference with the original model is that the execution trace does no longer include succeeded or failed logging operations. Instead, it proposes to include the events E and L, respectively corresponding to the `ENTER` and `LEAVE` transitions. This proposal appears to be more natural after reading Chapter 4. This modified execution trace definition is shown hereafter to be equivalent to that of the original model, through the usage of the second filter pass.

The proof elaborated by the co-authors in [123] is tailored for the model depicted in that very publication. The proof would need some adjustments to work on the present model. Next section hence shows an insight of reasoning about the present model.

7.4 Proof Sketch

The correctness of the system mechanisms addressing problems related to intermittent power is established by showing that any execution trace *matches* a specification. The subtlety of the semantics of *matching* is that code re-execution is allowed, as long as the repeated code is under a timeliness-related constraint. For instance, if the application needs to perform several `DRV` operations in a row, *i.e.*, in the `Driver` state, the application has to re-execute the entire sequence of `DRV` operations had the platform run out of power during the sequence execution.

To show, in a more straightforward manner, the correspondence between the execution trace and the specification, a three-pass filtering of the execution trace is proposed. The present description of the filtering passes can resemble regular expressions. The filtering itself is however conceptual and not dependent on the present expressions.

The first pass consists in removing the sub-traces of the life-cycles that could not successfully checkpoint, *i.e.*, that led to a Chkpt✗ event. As a consequence, the sub-trace between a Chkpt✓ event (or the beginning of the trace) and the following Chkpt✗ event can be discarded. For instance, the execution trace $E; \mathrm{op}_1; L;$Chkpt✓$; E; \mathrm{op}_2;$Chkpt✗ becomes $E; \mathrm{op}_1; L;$Chkpt✓. The motivation for this filter comes

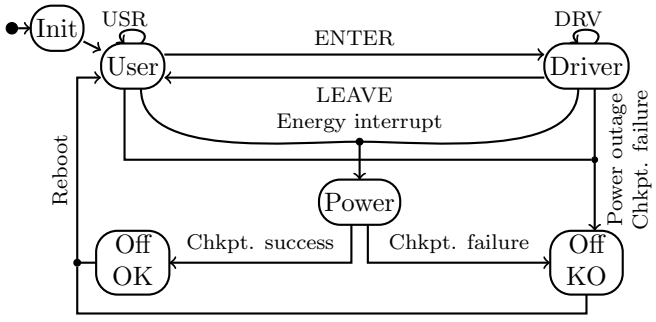


Figure 7.2: State machine of the system under intermittent power. The energy interrupt depicts the imminent power outage interrupt, while the power outage event depicts the actual power outage.

```
1  sram int x;
2  timely { // ENTER
3       temp_config();
4       x = temp_sample();
5  } // LEAVE
6  sram int y = compute(x);
7  timely { // ENTER
8       spi_config(RADIO);
9       radio_config(SEND);
10      radio_send(y);
11 } // LEAVE
```

(a) Example of application code. The `timely` blocks specify the timeliness constraints of the application. The keyword `timely` is conceptual and its expression, within an actual programming language or using a particular library, is prone to vary.

$$
\begin{aligned}
\text{SPEC} &= 3;4;8;9;10 \\
\text{EXEC} &= E;3;4;L;E;8;9;\text{Chkpt}\checkmark; \\
&\quad\ E;8;\text{Chkpt}\text{✗}; \\
&\quad\ E;8;9;10;L;\text{Chkpt}\checkmark \\
f(\text{EXEC}) &= E;3;4;L;E;8;9;\text{Chkpt}\checkmark; \\
&\quad\ E;8;9;10;L;\text{Chkpt}\checkmark \\
(g \circ f)(\text{EXEC}) &= E;3;4;L;\text{Chkpt}\checkmark; \\
&\quad\ E;8;9;10;L;\text{Chkpt}\checkmark \\
(h \circ g \circ f)(\text{EXEC}) &= 3;4;8;9;10
\end{aligned}
$$

(b) Specification (SPEC) and a possible execution trace (EXEC) under intermittent power of the application given in (a). The execution trace is refined after the first pass (f), the second pass (g) and the third pass (h). SPEC and $(h \circ g \circ f)$(EXEC) match. The peripheral operations are denoted by their line number.

Figure 7.3: Example of application code with timeliness constraints, alongside its specification and a possible execution trace, showing that they match.

from the operating system layer complying with the ability to always keep a valid checkpoint in memory, *e.g.*, using checkpoint double-buffering.

The second pass removes partial progress that was made in `Driver` state but that could not complete until the energy depleted. The sub-trace between an E event and a consecutive Chkpt✓ event is discarded, provided that there is no L event in between. The Chkpt✗ events were formerly removed by the first pass. For instance, the execution trace $E;\text{op}_1;L;\text{Chkpt}\checkmark;E;\text{op}_2;\text{Chkpt}\checkmark$ becomes $E;\text{op}_1;L;\text{Chkpt}\checkmark$. The motivation for this filter arises from the assumption of idempotent environment, as well as the software itself being idempotent due to the memory hierarchy assumption that volatile RAM is used as working memory and non-volatile memory for storage purpose only.

The third pass removes all E, L and Chkpt✓ events from the execution trace, to map the execution trace alphabet onto the specification alphabet. For instance, the execution trace $E;\text{op}_1;\text{op}_2;L;\text{Chkpt}\checkmark$ becomes $\text{op}_1;\text{op}_2$. This pass solely serves the purpose of comparing an execution trace to a specification.

Figure 7.3 illustrates the mapping between execution trace and specification, through a typical example of application with timeliness constraint. The considered execution trace, EXEC, involves three life-cycles. The first one is able to complete the first `timely` block (lines 3 and 4), to perform some computation on sampled data (line 6) and starts executing the second `timely` block. An imminent power outage interrupt occurs and the system successfully performs a checkpointing operation. On reboot, the application resumes at line 7 for the timeliness constraint imposes the second `timely` block to be re-tried from the beginning. During the second life-cycle, the `timely` block cannot complete and the following checkpointing operation fails. At the beginning of the third life-cycle, the system again resumes at line 7 thanks to the former valid checkpoint that guarantees non-negative progress despite checkpoints occasionally failing. The second `timely` block completes within the third life-cycle and, later on, a checkpointing operation is successfully performed. The idempotent properties of the operating system layer, in addition to the assumptions of this chapter, allow the aforementioned execution trace filter to operate. As a result, the execution trace matches the specification and thus, that particular execution is correct.

The definition of the specification and of the execution trace, combined with the filters proposed above, give an insight on how to reason about correctness of execution under intermittent power. This chapter however does not delve deeper into formalism and proof, for it is already done, alongside Coq development, in the original publication [123] by the co-authors.

7.5 Applications of the Model to Actual Systems

In order for the execution trace to match the specification, the software, or more probably the system layer, must have certain properties, namely solve P1, P2.1 and P3. Indeed, without P1 solved, no forward progress can be saved so the application would have to restart from the very beginning after every power outage. That situation is precisely that of the credit card model and, while it works using beamed energy supply, it would be completely useless when harvesting natural energy from the environment as discussed in Section 2.2. In addition, without P2.1 solved, the peripherals reboot in reset-state and one cannot expect the observable trace to be the same with the peripherals in different configurations that the ones expected by the software. The only valid trace that would exist under intermittent power would be a trace where the application restarts from the very beginning of the application every time a power outage occurs, similarly to P1 not being solved, hence also falling back into the credit card model. Without P3 solved, timeliness assumptions are broken, which can impede compliance with functional specifications from the data-sheets of the peripherals. The software is initially written assuming that these time-related concerns are handled consistently and power outages break the assumption of time-consistency.

According to table 3.1, the only two systems that comply with the requirements of solving P1, P2.1 and P3 are KARMA [67] and Sytare [66]. RESTOP [84] does not support timeliness constraints, for its driver layer operates at the granularity of an access to a single peripheral control register, while timeliness constraints may spread onto several accesses, *e.g.*, when calibrating a sensor. After modifying RESTOP to support timeliness constraints, which should not be difficult given the fact that RESTOP records all peripheral accesses, it could become a candidate to fit in the present model and its proof of correctness. KARMA and RESTOP both record a log of peripheral operations, which directly maps onto the traces of peripheral operations that are core to the specification and execution traces. Sytare only records the last defined states of peripherals and the restore function of every driver is in charge of finding a path in the drivers state automata to reach those states again, which can be considered an aggressive optimization of logging. Anyway, peripheral operation logging is simply a helper and makes the mapping with the present model more straightforward. In practice, it is simply an implementation detail. The properties to be respected being that P1, P2.1 and P3 are correctly treated. Sytare further employs function wrappers for driver routines, which correspond, to some extent, to the `ENTER` and `LEAVE` events of the execution trace.

Myriads of systems do address P1 and P3, but the peripheral state volatility is not handled: either peripherals are simply ignored or the system assumes that the peripherals are to be restored at a predefined hardcoded state [21, 58, 86, 61, 56, 55, 65, 87, 64, 78, 15, 57, 77, 89]. They could fit in the present model if they did manage to support peripherals, however it is not an easy task.

7.6 Conclusion

This chapter echoes a former publication [123] that proposes a specification of intermittent computing with peripherals. It highlights the required properties of a just-in-time checkpointing mechanism or operating system layer that must be complied with in order to guarantee correct execution under intermittent power. This model focuses on peripherals as they are the most challenging part of such checkpointing mechanisms, given the problems P2 and P3. A clear distinction between user mode and driver mode is stated as a basis of the model. A so-called specification records the sequence of peripheral operations that should be observed when running a given application under continuous power. The actual trace also records peripheral operations, as well as completed and failed checkpoints, plus the transitions between user mode and driver mode. If the checkpointing system enables to discard partial progress in the execution of driver mode without non-idempotent behavior, it can be shown that the execution trace effectively corresponds to the specification. Thus are stated the minimal conditions that a checkpointing system should satisfy to correctly handle peripherals. The correctness of the model was formally proven in [123]. Although the assumptions of the model limit the extent of its applications, it can nonetheless support KARMA and Sytare, plus RESTOP modulo the support for timeliness constraints. The most constraining assumption is the non-volatile memory solely used for storage purposes and not as working memory. Relaxing that very assumption would extend the usage of the resulting model to architectures A2 and A4 and thus to the checkpointing systems that leverage those architectures.

Chapter 8

Conclusion

The present thesis proposes a handful of contributions in the area of transiently-powered systems at operating system level. A certain amount of domains are involved in this work, including but not limited to software programming, electronics, architecture, simulation and formal proofs. Similarly, a large spectrum of tasks was undertaken during the realization of this work, from high-level modeling to embedded software development and maintenance, plus electronic design and soldering and correctness proving along the path. The following sections bring out the essence of this work in a nutshell and open the thinking to new and interesting perspectives.

8.1 A Quick Glance at the Contributions

An intermittent supply impedes the proper execution of traditional applications. Credit cards assume that the energy will be eventually back in amounts that allow the execution of an application to completion. More constrained systems, like transiently-powered systems, do not make that assumption and have to cope with power outages. Many parts of the system state are volatile and thus lost upon power outage, hence platforms embed non-volatile memory, not only to store instructions but also to save application progress. The micro-controller, the memory contents and the peripherals must be kept consistent with one another regarding the application as specified for a continuously-powered scenario. Special care must be given to peripherals which are not as simple to restore as memory, for they may involve specific sequences of operations, atomic waits, any form of operation that must be time-consistent and restarted from the beginning should power fail. Timeliness also adds time-related semantics to the application. The application developer knows the needs of the application and can tell whether a given piece of data needs to be time-consistent with another one, by for instance reasoning about the expiration time of a sensed value, considering that off-times can be as long as hours or days. Common solutions use checkpointing to spread the execution of applications across power outages, which can induce unspecified roll-backs in the control-flow and threaten non-volatile memory consistency if not properly handled.

Today's solutions to spreading execution across power outages are numerous and leverage different architectures. The most prominent ones consist of a mix of volatile and non-volatile memory, or of a single non-volatile memory. In every depicted solution, the processors are volatile for non-volatile processors are still experimental to this day. Checkpoints can be inserted statically, dynamically upon timer expiration or energy storage monitoring, or by design with multi-task languages that consider task boundaries checkpoints. Peripheral handling is not very popular, but works that consider peripherals tend to use more or less compressed logs of peripheral accesses.

Sytare was the first kernel to support peripheral state persistence and consistency, interrupts and application state persistence altogether. Its motto is to have the lightest impact on the application and driver programming, to enable support of almost bare-metal applications with the least amount of modifications. By using a wrapper mechanism around driver routines, Sytare achieves that objective: the application only prefixes driver calls to benefit from Sytare's atomicity mechanism. On the other hand, Sytare however requires the driver developer to provide an initialization routine, an interrupt routine, a save routine and a restoration routine. While the initialization and the interrupt routines exist anyway in

common driver development best practices, the save and restoration routines arise from the intermittent nature of the power supply. Peripherals are saved incrementally upon returning from driver routine and the state of the application and memory are saved in bulk when the energy storage runs low. Sytare was initially developed for MSP430 and, during the present thesis, acquired interrupt support. Sytare has then been ported to ARM Cortex-M micro-controllers and runs on the ARMorik platform, which was designed and assembled to propose an ARM-based alternative to the traditional MSP430-based platforms with more volatile memory as future platforms would tend to embed more volatile memory.

A novel and accurate peripheral-aware energy model for embedded systems is proposed in this thesis. Using separate basic power state machines, the cost of combinatorial automata can be avoided. Even a low-cost energy meter can provide figures that are precise enough for the model, thus populated by the measurements, to yield accurate results. The model is implemented inside a simulator that reproduces the behavior of the platform at instruction-level for genuine computational software sections, while the software sections involving peripherals are symbolically executed at the granularity of a driver routine. The symbolical execution consists in changing the state of the platform according to the specification of the driver routine, plus advancing time and augmenting energy consumption by amounts that were measured offline for the given driver routine. Hence, the model includes a time and an energy model of each relevant driver routine. The bulk of the driver routines can be represented as constants if the driver API is well-designed, or may be linear with respect to a parameter in the specific case of sending a radio packet, for instance, as the execution time and energy depend on packet length. Such a simulator thus predicts not only time, but also energy requirements, as well as the locations, in the code, of power outages. In addition, it enables large-scale testing of application benchmarks under repeatable virtual environment conditions. While the simulator is calibrated for an average case estimation, it is useful for dimensioning albeit not considering absolute worst-case concerns and gives a realistic hint about an application's needs and behavior.

As an improvement of classical full checkpointing, this thesis proposes to focus on modified volatile memory regions, as far as application state is concerned. Within the proposed alternative to Sytare checkpointing, peripherals are still managed as Sytare does. A Memory Protection Unit (MPU), present in all micro-controllers, is used to set all volatile memory regions as read-only. Upon access mismatch exception, the memory region is marked to be saved during next checkpointing operation, and the region is unlocked to enable further modifications for the kernel already knows that any change made to the region must be part of the next checkpoint. This is called MPU-based incremental checkpointing. The proposal comes along with a novel model of checkpointing systems with eleven parameters to play around with. The model enables the comparison between different checkpointing schemes and is furthermore a handy tool for a system designer willing to quickly explore design space based on their needs. When the application uses more than a few kilobytes of volatile memory, MPU-based incremental checkpointing is always better than a full copy or manually-managed section modification detection alternatives. With forthcoming platforms that are likely to embed more memory, incremental checkpointing will become even more beneficial and could thus integrate new systems for transiently-powered systems.

Finally, the present thesis formulates a formal proof of checkpointing systems for transiently-powered systems that was never attempted before. By considering the sequence of peripheral operations under continuous power as a specification, it is possible to show that the actual trace of peripheral operations that are observed under intermittent power does or does not comply to the specification, using trace refinement. The model leveraged by the proof encompasses just-in-time checkpointing systems and supports actual systems such as KARMA and Sytare. The proof unsurprisingly shows that a just-in-time checkpointing system must, besides making application state persistent, enable control-flow roll-backs for timeliness purposes and prevent non-idempotent operations. Discarding partial progress between the roll-back location and the location where the control-flow halted before the last power outage is a common way to prevent non-idempotent operations and thus fits the proof.

8.2 Perspectives

Each contribution of this thesis, as well as the thesis at a global scope, leads to questioning about transiently-powered systems, in terms of services to be provided by an operating system layer, of energy consumption estimation, of embedded systems simulation, of correctness of such systems under

intermittent power and of communication protocols that involve transiently-powered systems.

8.2.1 Operating System

Energy-aware scheduler Current operating systems for transiently-powered systems, depicted in Chapter 3 and Chapter 4, opportunistically place checkpoints, either statically or dynamically, to ensure forward progress. Recent advances target the evaluation of the energy consumption of an application or part of an application, as discussed in Chapter 6. That information may be used to dimension the hardware platform, such as the adequate capacity of the energy storage. It could be further used to build an energy-aware scheduler that can preempt operations that cannot complete due to the remaining energy and knowledge about the energetic properties of the operation to perform. It would be interesting to see priority inversion between tasks due to dynamic energetic concerns, when a less energy-consuming task preempts another more consuming and initially with higher priority, while complying to precedence constraints such as ensuring that a sensing task did produce a sufficient amount of data for a processing task to operate. Ultimately, such a scheduler would decrease the likeliness of atomic sections with timeliness constraints to be started without the guarantee that they can be run to completion before any power outage occurs. That scheduler would enable to save failed attempts and use the remaining energy either to make forward progress in another non-atomic task, or to shutdown the platform early so that the required energy level is reached sooner than if the energy storage was completely depleted trying to run an atomic section that could not complete. The latter point requires close cooperation with the power manager module, for it is yet not common for a power manager for low-power energy-harvesting devices to provide a logic pin to force an early shutdown.

Incremental checkpointing The MPU-based incremental checkpointing contribution from Chapter 6 can be further extended to not only have a selective save operation, but also a lazy restoration. By setting memory regions with no read nor write access on boot, the kernel is able to catch any attempt to read a memory region and restore that region only on that occasion. Then, the memory region is granted with read permission and the next write to it will flag the region as to be part of the next checkpoint, following the scheme depicted in Chapter 6.

Memory placement Within the optimistic scenario where the performance gap between volatile memories and non-volatile memories shrinks, the memory organization of transiently-powered systems should tend to resemble a flat organization rather than a hierarchy. While some existing works currently place all data into non-volatile memory, many others use volatile memory as a fast main memory and propose a commit policy to make changes to volatile memory persistent, by copying the adequate pieces of data into non-volatile memory. Regardless of the solution, today's assumptions weigh down the usage of memory from the application developer's point of view. They can be restricted to the sole use of volatile memory whereas the non-volatile memory is only managed by the operating system layer, as done in Sytare. Or they can be restricted to use a specific non-volatile memory access API, hardly allowing pointers, in order to tackle non-idempotence issues arising from letting the application developer place their variables in non-volatile memory. Efforts, either in software or directly in architectural support, could further simplify variable management by allowing the application developer to use their variables as it pleases them, with a smart underlying mechanism that would address non-idempotence concerns if a given variable resides in non-volatile memory.

8.2.2 Worst-Case Energy Consumption

While the contribution model of Chapter 5 studies an average case, worst cases are more popular due to their bounding properties, useful for dimensioning hardware and conducting feasibility studies. Static analyses do already exist.

0g [134] computes the Worst-Case Energy Consumption (WCEC) of a given program, either in absolute terms whenever the consumption of all micro-controller instructions is properly characterized, or in relative terms on the contrary. 0g specifically targets energy-constrained embedded systems, however its analysis does not include peripherals, which are the most versatile components and have the greatest impact on energy consumption. However, the authors of 0g rightfully state that computing the WCEC is

not as simple as computing the Worst-Case Execution Time (WCET) and multiplying it with the power consumption, for simply multiplying WCET with the power consumption is a common misconception of the domain. Using that misconception, the worst-case power consumption must be considered, *i.e.*, the power consumption of the platform with all peripherals in their most consuming power state. Embedded software developers know that their software must be energy-efficient and thus always propose a better peripheral management policy than setting all peripherals to their most consuming power state. Hence, a poorly-built WCEC may be orders of magnitude higher than realistic scenarios.

SysWCEC [106] considers several power states for each peripheral. The peripheral accesses are studied at driver API level, which does not encompass subtle state variations such as when configuring several control registers in a row. In addition, short busy-waits are often present in embedded software, *e.g.*, when waiting for an oscillator to settle or for a hardware buffer to consume its data. It is thus not unfrequent to see control structures such as **while**(REGISTER & mask);. Any WCEC or WCET tool would yield either an infinite energy consumption or execution time, or would assume an arbitrary amount of iterations and yield an unrealistic result. It would be interesting to incorporate data-sheet knowledge and measurements as annotations for the WCEC tool, in order to yield realistic upper-bounds of the energy consumption.

Also, fine instruction study should be performed. Not all stores and loads are equivalent and the addresses give useful information about what parts of the platform are involved in a given instruction. Address analysis would enable the WCEC tool to distinguish between volatile memories, non-volatile memories and each peripheral, knowing that the wait state, *i.e.*, the duration of the memory transaction, depends on the nature of the underlying memory and that peripherals further alter the state of the platform. Such alterations include power consumption and clock frequency, that have a strong impact on energy estimation. In addition, serial buses require reads and writes to the same address. For instance, **while**(n−−) {FIFO = ∗ptr++;} writes n words to a First In, First Out (FIFO) memory. Technically, there are n writes to the same address. However, the result is different from only writing the last value for the FIFO actually accumulates data, while a variable would not. On the contrary, **while**(n−−) {a = ∗ptr++;} could be coarsely rewritten a = ptr[n−1]; n = 0; provided that a and ptr are variables without side effect.

At the end of the day, peripheral-aware WCEC analysis with fine grain instruction and address analysis would be both an interesting and challenging topic to work on. In the context of this thesis, a joint work with another research team was initiated towards that direction. The idea was to modify the Heptane WCET estimation tool [135] to build a new WCEC tool capable of accurately estimating the energy consumption of busy-waits, of memory accesses and of peripheral-related operations using data-sheet figures and actual measurements as inputs. However, it remained stalled due to a lack of time.

8.2.3 Simulation

The simulator depicted in Chapter 5 provides accurate results, but that accuracy can be further improved. Instructions that use memory-based operands are not yet checked against the target addresses. This results in some discrepancy with respect to the reality for the simulator primarily targets a MSP-EXP430FR5739 with heterogeneous memory. Accesses to volatile memory and non-volatile memory do not require the same duration and these properties must be taken into account by the simulator. Furthermore, to enable extensive benchmarking, it would be interesting to integrate a scenario description language to the simulator. Hence, it would become possible to place events at some points in time, such as the reception of a radio packet. Scenarios could incorporate pre-programmed data exchange to simulate a distant node for instance, as well as events related to energy harvesting such as an energy surge or specific charging patterns. With both improvements, the resulting simulator would have increased accuracy and become a basis for sandboxing transiently-powered systems and wireless sensor networks.

8.2.4 Formal Proof

The proof exposed in Chapter 7 reasons about an architecture where volatile memory is used as main memory and non-volatile memory is solely used for persistent storage purposes. While that assumption encompasses a large amount of today's systems for transiently-powered systems, many others do not

make use of volatile memory and prefer working directly in non-volatile memory. The proof could thus be further extended to support those systems.

8.2.5 Communication Protocol

Off-the-shelf communication protocols often assume a stable power supply. Common ones allow failures to occur, for instance when a network packet is poorly sent or not received at all. However, communication failures are often addressed by redundancy and checksums, which implicitly assume that the communicating nodes can provide both computational abilities and energy to reissue a packet or to hash its contents. Within transiently-powered systems, one has to consider both sides of the communication. A first scenario would be that a remote operator, with steady power, queries one or more transiently-powered systems and they answer opportunistically if they received the query within an on-time. The communication would hence be asymmetrical, for on one hand there would be an operator with great amounts of energy and on the other hand an energy-harvesting device. It would remove half of the difficulty, for only the power supply of the energy-harvesting device would be an issue. The applications of such a scenario are rather limited, but it is a natural approach at building wireless sensor networks where the sensor nodes are solely responsible for sensing and transmitting the data when requested. Another, more complex, scenario might only involve transiently-powered systems or allow sensor nodes to communicate with one another, *e.g.*, to propagate the data from distant sensor nodes towards an operator. This second scenario is far more interesting, scientifically speaking, than the first one, for in a local communication both ends may shutdown at any time. An energy-related rendezvous may be agreed between both nodes, as part of a handshake sequence, to initiate a data exchange when both nodes can afford it, in terms of energy. For instance, the first node that harvested enough energy can relay part of its energy to the other node in order to make both nodes available for communication.

8.3 Transiently-Powered Systems of Tomorrow

Today, micro-controller manufacturers are fainthearted regarding the integration of non-volatile RAM, let alone the replacement of volatile RAM by non-volatile RAM, in new micro-controller designs. Concretely, only Texas Instruments provide such micro-controllers, and a chat with other manufacturers revealed their untrust in the economic viability of such architectures. The first step to overcome is thus to convince industrials to dig into the direction of platforms equipped with non-volatile RAM, aside from data-centers. Recent MSP430 platforms from Texas Instruments tend to corroborate that newer designs incorporate more memory, both volatile and non-volatile. It is thus expectable that, provided that these micro-controllers perdure, they would embed more memory and even more computation capabilities and more complex peripherals.

Advances in platform energy efficiency and in energy harvester design may lengthen life-cycle duration or enable form-factor reduction to unlock new kinds of applications. However, power outages are still foreseen to be part of tomorrow's low-power platforms, hence transiently-powered systems will continue to exist. The usage of lightweight operating systems that provide a failure-resilient run-time environment to applications may become more popular as software engineering for these platforms would be eased. Such operating systems would provide efficient answers to application and peripheral state volatility, timeliness constraints and issues related to non-idempotent operations altogether.

Bibliography

[1] M. Prauzek, J. Konecny, M. Borova, K. Janosova, J. Hlavica, and P. Musilek, "Energy Harvesting Sources, Storage Devices and System Topologies for Environmental Wireless Sensor Networks: A Review," *Sensors*, vol. 18, no. 8, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/8/2446

[2] F. Yildiz, "Potential Ambient Energy-Harvesting Sources and Techniques," *The Journal of Technology Studies*, vol. 35, no. 1, sep 2009.

[3] S. Sudevalayam and P. Kulkarni, "Energy Harvesting Sensor Nodes: Survey and Implications," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 3, pp. 443–461, 2011.

[4] A. Duque, "Bidirectional Visible Light Communications for the Internet of Things," Theses, Université de Lyon - INSA Lyon, Oct. 2018. [Online]. Available: https://hal.inria.fr/tel-01940002

[5] G. Moayeri Pour and W. D. Leon-Salas, "Solar energy harvesting with light emitting diodes," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 1981–1984.

[6] I. Sari, T. Balkan, and H. Külah, "An Electromagnetic Micro Power Generator for Low-Frequency Environmental Vibrations Based on the Frequency Upconversion Technique," *Journal of Microelectromechanical Systems*, vol. 19, no. 1, pp. 14–27, 2010.

[7] X. Lu, P. Wang, D. Niyato, D. I. Kim, and Z. Han, "Wireless Networks With RF Energy Harvesting: A Contemporary Survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 757–789, 2015.

[8] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-Based Battery-Free Programmable Sensing Platform," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 11, pp. 2608–2615, 2008.

[9] B. H. Calhoun, D. C. Daly, Naveen Verma, D. F. Finchelstein, D. D. Wentzloff, A. Wang, Seong-Hwan Cho, and A. P. Chandrakasan, "Design considerations for ultra-low energy wireless microsensor nodes," *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 727–740, 2005.

[10] Z. Zhou, M. Peng, Z. Zhao, W. Wang, and R. S. Blum, "Wireless-Powered Cooperative Communications: Power-Splitting Relaying With Energy Accumulation," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 4, pp. 969–982, 2016.

[11] I. Krikidis, G. Zheng, and B. Ottersten, "Harvest-use cooperative networks with half/full-duplex relaying," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, 2013, pp. 4256–4260.

[12] A. M. Siddiqui, L. Musavian, and Q. Ni, "Energy efficiency optimization with energy harvesting using harvest-use approach," in *2015 IEEE International Conference on Communication Workshop (ICCW)*, 2015, pp. 1982–1987.

[13] T. Li, P. Fan, Z. Chen, and K. B. Letaief, "Optimum Transmission Policies for Energy Harvesting Sensor Networks Powered by a Mobile Control Center," *IEEE Transactions on Wireless Communications*, vol. 15, no. 9, pp. 6132–6145, 2016.

[14] I. T. Castro, L. Landesa, and A. Serna, "Modeling the Energy Harvested by an RF Energy Harvesting System Using Gamma Processes," *Mathematical Problems in Engineering*, vol. 2019, pp. 1–12, apr 2019.

[15] A. Colin, E. Ruppel, and B. Lucia, "A Reconfigurable Energy Storage Architecture for Energy-Harvesting Devices," *SIGPLAN Not.*, vol. 53, no. 2, pp. 767–781, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173210

[16] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 5–16.

[17] S. Peng and C. Low, "Prediction free energy neutral power management for energy harvesting wireless sensor nodes," *Ad Hoc Networks*, vol. 13, pp. 351–367, 2014.

[18] F. A. Aoudia, M. Gautier, and O. Berder, "Fuzzy power management for energy harvesting Wireless Sensor Nodes," in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[19] S. Peng and C. P. Low, "Throughput optimal energy neutral management for energy harvesting wireless sensor networks," in *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, 2012, pp. 2347–2351.

[20] F. Yuan, Q. T. Zhang, S. Jin, and H. Zhu, "Optimal Harvest-Use-Store Strategy for Energy Harvesting Wireless Systems," *IEEE Transactions on Wireless Communications*, vol. 14, no. 2, pp. 698–710, 2015.

[21] B. Ransford, J. Sorber, and K. Fu, "Mementos: System Support for Long-Running Computation on RFID-Scale Devices," *SIGPLAN Not.*, vol. 46, no. 3, pp. 159–170, Mar. 2011.

[22] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, 2015.

[23] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-Powered Computers," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 97–109.

[24] Y. K. Tan and S. K. Panda, "A Novel Piezoelectric Based Wind Energy Harvester for Low-power Autonomous Wind Speed Sensor," in *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, 2007, pp. 2175–2180.

[25] A. Ibrahim, M. Jain, E. Salman, R. Willing, and S. Towfighian, "A smart knee implant using triboelectric energy harvesters," *Smart Materials and Structures*, vol. 28, no. 2, p. 025040, jan 2019.

[26] G. Saini and M. Shojaei Baghini, "A Generic Power Management Circuit for Energy Harvesters With Shared Components Between the MPPT and Regulator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 3, pp. 535–548, 2019.

[27] *LTC3588-1 Nanopower Energy Harvesting Power Supply*, Linear Technology, 2010.

[28] J. de Winkel, V. Kortbeek, J. Hester, and P. Pawełczak, "Battery-Free Game Boy," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 4, no. 3, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3411839

[29] M. Dini, A. Romani, M. Filippi, V. Bottarel, G. Ricotti, and M. Tartagni, "A Nanocurrent Power Management IC for Multiple Heterogeneous Energy Harvesting Sources," *IEEE Transactions on Power Electronics*, vol. 30, no. 10, pp. 5665–5680, 2015.

[30] G. C. Martins and W. A. Serdijn, "An RF Energy Harvester with MPPT Operating Across a Wide Range of Available Input Power," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[31] A. Lazaro, R. Villarino, and D. Girbau, "A Survey of NFC Sensors Based on Energy Harvesting for IoT Applications," *Sensors*, vol. 18, no. 11, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/11/3746

[32] V. Talla, B. Kellogg, B. Ransford, S. Naderiparizi, S. Gollakota, and J. R. Smith, "Powering the next Billion Devices with Wi-Fi," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2716281.2836089

[33] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith, "Ambient Backscatter: Wireless Communication out of Thin Air," vol. 43, no. 4. New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 39–50.

[34] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power Management in Energy Harvesting Sensor Networks," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, pp. 32–es, Sep. 2007.

[35] M. Magno, D. Porcarelli, D. Brunelli, and L. Benini, "InfiniTime: A multi-sensor energy neutral wearable bracelet," in *International Green Computing Conference*, 2014, pp. 1–8.

[36] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith, "Battery-Free Cellphone," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 1, no. 2, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3090090

[37] T. Li and X. Zhou, "Battery-Free Eye Tracker on Glasses," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 67–82.

[38] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith, "Towards Battery-Free HD Video Streaming," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. USA: USENIX Association, 2018, pp. 233–247.

[39] H. Nakamura, T. Nakada, and S. Miwa, "Normally-off computing project: Challenges and opportunities," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 1–5.

[40] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, Apr. 2013. [Online]. Available: https://hal.inria.fr/hal-00945122

[41] J. Labrosse, *MicroC/OS-II: The Real Time Kernel*. CRC Press, 2002.

[42] J. J. Labrosse, *UC/OS-III: The Real-Time Kernel and the Texas Instruments Stellaris MCUs*. Weston, FL, USA: Micrium Press, 2010.

[43] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Springer-Verlag, 01 2005, vol. 00, pp. 115–148.

[44] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, 2004, pp. 455–462.

[45] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 29–42.

[46] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An Energy-Aware Runtime for Computational RFID," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, pp. 197–210.

[47] P. Zhang, D. Ganesan, and B. Lu, "QuarkOS: Pushing the Operating Limits of Micro-Powered Sensors," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. USA: USENIX Association, 2013, p. 7.

[48] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating System Implications of Fast, Cheap, Non-Volatile Memory," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. USA: USENIX Association, 2011, p. 2.

[49] J. Meena, S. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale Research Letters*, vol. 9, no. 1, p. 526, 2014.

[50] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging NVM: A Survey on Architectural Integration and Research Challenges," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 2, nov 2017.

[51] G. Di Pendina, K. Torki, G. Prenat, Y. Guillemenet, and L. Torres, "Ultra Compact Non-volatile Flip-Flop for Low Power Digital Circuits Based on Hybrid CMOS/Magnetic Technology," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, J. L. Ayala, B. García-Cámara, M. Prieto, M. Ruggiero, and G. Sicard, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 83–91.

[52] K. Garello, F. Yasin, and G. S. Kar, "Spin-Orbit Torque MRAM for ultrafast embedded memories: from fundamentals to large scale technology integration," in *2019 IEEE 11th International Memory Workshop (IMW)*, 2019, pp. 1–4.

[53] D. C. Gilmer, T. Rueckes, and L. Cleveland, "NRAM: a disruptive carbon-nanotube resistance-change memory," *Nanotechnology*, vol. 29, no. 13, p. 134003, feb 2018. [Online]. Available: https://doi.org/10.1088%2F1361-6528%2Faaaacb

[54] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu, "TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 221–236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati

[55] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent Execution without Checkpoints," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017.

[56] A. Colin and B. Lucia, "Chain: Tasks and Channels for Reliable Intermittent Programs," *SIGPLAN Not.*, vol. 51, no. 10, pp. 514–530, Oct. 2016. [Online]. Available: https://doi.org/10.1145/3022671.2983995

[57] E. Ruppel and B. Lucia, "Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1085–1100. [Online]. Available: https://doi.org/10.1145/3314221.3314583

[58] B. Lucia and B. Ransford, "A Simpler, Safer Programming and Execution Model for Intermittent Systems," *SIGPLAN Not.*, vol. 50, no. 6, pp. 575–585, Jun. 2015.

[59] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan, "QuickRecall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers," *J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 1, Aug. 2015.

[60] R. Smith and S. Rixner, "Surviving Peripheral Failures in Embedded Systems," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. USA: USENIX Association, 2015, pp. 125–137.

[61] J. Van Der Woude and M. Hicks, "Intermittent Computation without Hardware Support or Programmer Intervention," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, pp. 17–32.

[62] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui, "On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 203–207.

[63] M. Surbatovich, L. Jia, and B. Lucia, "I/O Dependent Idempotence Bugs in Intermittent Systems," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360609

[64] K. Maeng and B. Lucia, "Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, pp. 129–144.

[65] J. Hester, K. Storer, and J. Sorber, "Timely Execution on Intermittently Powered Batteryless Sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131672.3131673

[66] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1390–1403, sep 2019.

[67] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui, "Intermittent Asynchronous Peripheral Operations," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, ser. SenSys '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 55–67.

[68] T. Li, R. Ragel, and S. Parameswaran, "Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 875–880.

[69] M. Hicks, "Clank: Architectural Support for Intermittent Computation," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 228–240, Jun. 2017.

[70] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs," in *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2013, pp. 216–224.

[71] A. Mirhoseini, B. D. Rouhani, E. Songhori, and F. Koushanfar, "Chime: Checkpointing Long Computations on Intermittently Energized IoT Devices," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 4, pp. 277–290, oct 2016.

[72] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, J. Shu, and H. Yang, "Ambient Energy Harvesting Nonvolatile Processors: From Circuit to System," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[73] Y. Liu, Z. Wang, A. Lee, F. Su, C. Lo, Z. Yuan, C. Lin, Q. Wei, Y. Wang, Y. King, C. Lin, P. Khalili, K. Wang, M. Chang, and H. Yang, "4.7 A 65nm ReRAM-enabled nonvolatile processor with 6× reduction in restore time and 4× higher clock frequency using adaptive data retention and self-write-termination nonvolatile logic," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016, pp. 84–86.

[74] F. Su, W. Chen, L. Xia, C. Lo, T. Tang, Z. Wang, K. Hsu, M. Cheng, J. Li, Y. Xie, Y. Wang, M. Chang, H. Yang, and Y. Liu, "A 462GOPs/J RRAM-based nonvolatile intelligent processor for energy harvesting IoE system featuring nonvolatile logics and processing-in-memory," in *2017 Symposium on VLSI Technology*, 2017, pp. T260–T261.

[75] A. Lee, C. Lo, C. Lin, W. Chen, K. Hsu, Z. Wang, F. Su, Z. Yuan, Q. Wei, Y. King, C. Lin, H. Lee, P. Khalili Amiri, K. Wang, Y. Wang, H. Yang, Y. Liu, and M. Chang, "A ReRAM-Based Nonvolatile Flip-Flop With Self-Write-Termination Scheme for Frequent-OFF Fast-Wake-Up Nonvolatile Processors," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 8, pp. 2194–2207, 2017.

[76] K. Monga, A. Malhotra, N. Chaturvedi, and S. Gurunayaranan, "A Novel Low Power Non-Volatile SRAM Cell with Self Write Termination," in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, pp. 1–4.

[77] K. Maeng and B. Lucia, "Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1101–1116. [Online]. Available: https://doi.org/10.1145/3314221.3314613

[78] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive Kernel for Tiny Batteryless Sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 41–53.

[79] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[80] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," *arXiv e-prints*, p. arXiv:1308.5174, Aug. 2013.

[81] H. Zhang, M. Salajegheh, K. Fu, and J. Sorber, "Ekho: Bridging the Gap between Simulation and Reality in Tiny Energy-Harvesting Sensors," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, ser. HotPower '11. New York, NY, USA: Association for Computing Machinery, 2011.

[82] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: A Language and Runtime System for Perpetual Systems," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 161–174.

[83] H. Zhang and B. Ransford, "Moo : A Batteryless Computational RFID and Sensing Platform," University of Massachusetts, Amherst, Tech. Rep., 2011.

[84] A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell, "RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems," *Sensors*, vol. 18, no. 1, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/1/172

[85] N. A. Bhatti and L. 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*, ser. IPSN '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 209–219.

[86] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[87] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, Apr. 2017. [Online]. Available: https://doi.org/10.1145/2983628

[88] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawełczak, "Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices," *ACM Trans. Sen. Netw.*, vol. 16, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3360285

[89] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-Sensitive Intermittent Computing Meets Legacy Software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 85–99.

[90] K. Maeng and B. Lucia, "Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1005–1021. [Online]. Available: https://doi.org/10.1145/3385412.3385998

[91] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016, pp. 1–6.

[92] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, and J. Sorber, "Persistent Clocks for Batteryless Sensing Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 4, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2903140

[93] H. Williams, X. Jian, and M. Hicks, "Forget Failure: Exploiting SRAM Data Remanence for Low-Overhead Intermittent Computation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 69–84.

[94] B. Cassens, A. Martens, and R. Kapitza, "The Neverending Runtime: Using New Technologies for Ultra-Low Power Applications with an Unlimited Runtime," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16. USA: Junction Publishing, 2016, pp. 325–330.

[95] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 199–213. [Online]. Available: https://doi.org/10.1145/3297858.3304011

[96] S. S. Baghsorkhi and C. Margiolas, "Automating Efficient Variable-Grained Resiliency for Low-Power IoT Systems," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 38–49.

[97] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence for Transiently-Powered Systems," in *2017 Global Internet of Things Summit (GIoTS)*. IEEE, jun 2017.

[98] N. A. Bhatti and L. Mottola, "Efficient State Retention for Transiently-Powered Embedded Sensing," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16. USA: Junction Publishing, 2016, pp. 137–148.

[99] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power Embedded Systems," in *2020 Global Internet of Things Summit (GIoTS)*. IEEE, jun 2020.

[100] A. Acquaviva, L. Benini, and B. Riccó, "Energy Characterization of Embedded Real-Time Operating Systems," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 13–18, Dec. 2001.

[101] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin, "Profiling Software for Energy Consumption," in *2012 IEEE International Conference on Green Computing and Communications*, 2012, pp. 515–522.

[102] J. S. Miguel, K. Ganesan, M. Badr, C. Xia, R. Li, H. Hsiao, and N. E. Jerger, "The EH Model: Early Design Space Exploration of Intermittent Processor Architectures," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, oct 2018.

[103] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking Energy in Networked Embedded Systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, pp. 323–338.

[104] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes, "Powertrace: Network-level Power Profiling for Low-power Wireless Networks," SICS, Tech. Rep. 2011:05, 2011.

[105] N. Cherifi, T. Vantroys, A. Boe, C. Herault, and G. Grimaud, "Automatic Inference of Energy Models for Peripheral Components in Embedded Systems," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, aug 2017.

[106] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 24:1–24:25. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8979

[107] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. IEEE, apr 2008.

[108] A. Di Nisio, T. Di Noia, C. G. C. Carducci, and M. Spadavecchia, "High Dynamic Range Power Consumption Measurement in Microcontroller-Based Applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 65, no. 9, pp. 1968–1976, 2016.

[109] B. Dezfouli, I. Amirtharaj, and C.-C. C. Li, "EMPIOT: An energy measurement platform for wireless IoT devices," *Journal of Network and Computer Applications*, vol. 121, pp. 135–148, 2018.

[110] O. Hahm and S. Adler, "Profiling energy consumption of Wireless Sensor Nodes with almost zero effort," in *2012 IEEE International Conference on Communications (ICC)*. IEEE, jun 2012.

[111] T. Hönig, H. Janker, C. Eibel, W. Schröder-Preikschat, O. Mihelic, and R. Kapitza, "Proactive Energy-Aware Programming with PEEK," in *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, ser. TRIOS'14. USA: USENIX Association, 2014, p. 6.

[112] T. Bouhadiba, M. Moy, F. Maraninchi, J. Cornet, L. Maillet-Contoz, and I. Materic, "Co-simulation of Functional SystemC TLM Models with Power/Thermal Solvers," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, may 2013.

[113] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh, "Simulating the Power Consumption of Large-Scale Sensor Network Applications," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 188–200.

[114] G. Patrigeon, P. Benoit, and L. Torres, "FPGA-Based Platform for Fast Accurate Evaluation of Ultra Low Power SoC," in *2018 28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2018, pp. 123–128.

[115] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "ArchC: a systemC-based architecture description language," in *16th Symposium on Computer Architecture and High Performance Computing*, 2004, pp. 66–73.

[116] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "MPU-based incremental checkpointing for transiently-powered systems," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 89–96.

[117] F. A. Aoudia, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to NVRAM for transiently-powered systems," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, may 2014.

[118] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Efficient Intermittent Computing with Differential Checkpointing," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 70–81.

[119] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. USA: USENIX Association, 1995, p. 18.

[120] S. C. Bartling, S. Khanna, M. P. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams, "An 8MHz 75µA/MHz zero-leakage non-volatile logic-based Cortex-M0 MCU SoC exhibiting 100% digital state retention at VDD=0V with <400ns wakeup and sleep transitions," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2013, pp. 432–433.

[121] B. Egger, Y. Cho, C. Jo, E. Park, and J. Lee, "Efficient Checkpointing of Live Virtual Machines," *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 3041–3054, oct 2016.

[122] D. Pala, I. Miro-Panades, and O. Sentieys, "Freezer: A Specialized NVM Backup Controller for Intermittently-Powered Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[123] G. Berthou, P.-E. Dagand, D. Demange, R. Oudin, and T. Risset, "Intermittent Computing with Peripherals, Formally Verified," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 85–96.

[124] M. Surbatovich, B. Lucia, and L. Jia, "Towards a Formal Foundation of Intermittent Computing," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428231

[125] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying Concurrent, Crash-Safe Systems with Perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 243–258. [Online]. Available: https://doi.org/10.1145/3341301.3359632

[126] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim, "Verifying Correctness of Persistent Concurrent Data Structures," in *Formal Methods – The Next 30 Years*, M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds. Cham: Springer International Publishing, 2019, pp. 179–195.

[127] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich, "Specifying Crash Safety for Storage Systems," in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, ser. HOTOS'15. USA: USENIX Association, 2015, p. 21.

[128] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare Logic for Certifying the FSCQ File System," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 18–37. [Online]. Available: https://doi.org/10.1145/2815400.2815402

[129] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[130] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model," in *Distributed Computing*, C. Gavoille and D. Ilcinkas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327.

[131] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[132] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, pp. 653–669.

[133] E. Koskinen and J. Yang, "Reducing Crash Recoverability to Reachability," *SIGPLAN Not.*, vol. 51, no. 1, pp. 97–108, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837648

[134] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, "Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 105–114.

[135] D. Hardy, B. Rouxel, and I. Puaut, "The Heptane Static Worst-Case Execution Time Estimation Tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, ser. International Workshop on Worst-Case Execution Time Analysis, vol. 8, Dubrovnik, Croatia, Jun. 2017, p. 12.

Appendix A

Glossary

ABI	Application Binary Interface
API	Application Programming Interface
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
DMA	Direct Memory Access
DSL	Domain-Specific Language
DUT	Device Under Test
FRAM	Ferroelectric Random-Access Memory
GPIO	General-Purpose Input/Output
I^2C	Inter-Integrated Circuit
ISA	Instruction Set Architecture
LED	Light-Emitting Diode
LPM	Low-Power Mode
Life-cycle	Period of time when the platform is continuously powered on. The electronic requirements of all hardware components, including the microcontroller, peripherals and potential external memories, are met.
MMU	Memory Management Unit
MPU	Memory Protection Unit
NVRAM	Non-Volatile Random-Access Memory
Power outage	The exact moment when the power supply fails, leaving the platform in a hardware off state. A hardware reboot is required, when the power eventually comes back. Right after that point, all volatile elements are considered lost and peripherals are at reset-state.
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
WCEC	Worst-Case Energy Consumption
WCET	Worst-Case Execution Time
WSN	Wireless Sensor Network

Appendix B

List of Publications

Journal

[1] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A Lightweight Kernel for NVRAM-based Transiently-Powered Systems," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1390–1403, 2019.

Conferences

[2] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence for Transiently-Powered Systems," in *2017 Global Internet of Things Summit (GIoTS)*, 2017, pp. 1–6.

[3] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power Embedded Systems," in *2020 Global Internet of Things Summit (GIoTS)*, 2020, pp. 1–6.

[4] G. Berthou, P.-E. Dagand, D. Demange, R. Oudin, and T. Risset, "Intermittent Computing with Peripherals, Formally Verified," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 85–96.

[5] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "MPU-based Incremental Checkpointing for Transiently-Powered Systems," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 89–96.

Workshop

[6] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence and Interrupt Management for Transiently Powered Systems," in *2018 9th Non-Volatile Memories Workshop (NVMW)*, 2018.

Appendix C

Abstract

Some electronic devices cannot embed any battery because of space- or cost-related concerns. Notably, small devices harvest energy from their environment to gap the absence of battery. Medical sensors and crop probes are examples of such devices. These devices are likely to consume energy faster than they can harvest, when considering non-solar energy or small harvesters for instance. They are thus exposed to frequent power outages and must cope with an intermittent supply and are referred to as transiently-powered systems. Instead of restarting the application from the very beginning on every boot, it is possible to resume the application where it stopped. Non-volatile memories keep their data across power outages and their usage enable persistent data storage. Non-volatile RAM (NVRAM) and traditional volatile RAM have similar access latencies, which makes NVRAM technologies good candidates for persistent storage of energy-constrained devices. This work highlights the benefits of using NVRAM for the purpose of enabling the execution of long-running application despite power outages. This study proposes to solve issues related to intermittent supply at operating system level, which role is to manage application progress persistence and maintain consistency between memories and peripherals. This work also proposes a model for transiently-powered systems to lay the ground for energy consumption estimation of code involving peripherals. A new checkpointing mechanism, based on a hardware MPU, is proposed as a checkpointing optimization. Finally, this work proposes an introduction to proof of correctness at operating system level for transiently-powered systems.

Certains objets électroniques ne peuvent embarquer de pile ou de batterie, pour des raisons de place ou de coût. C'est le cas de petits objets, récoltant de l'énergie depuis leur environnement, afin de pallier l'absence de batterie. Capteurs médicaux et sondes agronomiques en sont des exemples. En général, ces objets consomment de l'énergie plus rapidement qu'ils n'en récoltent, notamment pour les appareils non équipés de cellules photovoltaïques. En résultent de fréquentes coupures de courant lors de l'exécution de l'application, on parle alors d'alimentation intermittente. Plutôt que de recommencer l'application du début à chaque redémarrage, il est possible de développer des mécanismes afin de reprendre l'exécution à l'endroit où elle s'était arrêtée. L'utilisation de mémoires non-volatiles, qui ne perdent pas leurs données lorsque la plate-forme est éteinte, permet le stockage de données en dépit des coupures de courant. En particulier, les RAM non-volatiles (NVRAM) ont des temps d'accès voisins des RAM volatiles, ce qui permet de les utiliser comme des mémoires courantes, à l'opposé des mémoires type Flash dont le temps et l'énergie d'écriture sont trop élevés. Ce travail met en exergue l'utilisation de NVRAM afin d'exécuter une application longue en dépit des coupures de courant. Il s'articule autour du développement d'un système d'exploitation visant ce type d'objet, assurant une cohérence entre les mémoires et l'état des périphériques à chaque instant. En outre, il propose un modèle de systèmes à alimentation intermittente, servant de base pour l'évaluation des coûts énergétiques de portions de code impliquant des opérations sur les péripheriques. Un nouveau mécanisme de sauvegarde basé sur une MPU matérielle est proposé pour optimiser la sauvegarde de données. Enfin, ce travail propose une introduction à la preuve formelle d'exécution correcte pour systèmes à alimentation intermittente.

Appendix D

Résumé de la thèse

Titre de la thèse en Français: Système d'exploitation dédié aux systèmes embarqués basse consommation équipés de NVRAM.

L'informatique s'intègre dans beaucoup d'objets du quotidien, et cette tendance se renforce toujours. Des étiquettes de vêtements, des capteurs agricoles, des implants médicaux et des capteurs déployés dans des zones difficiles d'accès en sont autant d'exemples. Ces objets ne peuvent se permettre d'embarquer piles et batteries, car ces dernières occupent beaucoup d'espace et nécessitent l'intervention d'un opérateur humain pour remplacer les piles ou recharger les batteries. Au contraire, des objets autonomes doivent récolter de l'énergie depuis leur environnement. Les sources d'énergie sont variées: la lumière, les ondes électromagnétiques, le gradient de température, le mouvement, *etc.* De petits objets ne peuvent embarquer que de petits récolteurs d'énergie et, en général, de telles plates-formes consomment de l'énergie plus rapidement qu'elles n'en récoltent. En résultent de nombreuses coupures de courant inopinées, on parle alors d'alimentation intermittente.

L'alimentation intermittente empêche l'exécution correcte d'applications telles qu'elles sont conçues actuellement. Par exemple, les cartes de paiement font l'hypothèse qu'il sera toujours possible d'exécuter une application de bout en bout en une fois. Des systèmes plus contraints, faisant l'objet de cette thèse, n'émettent pas cette hypothèse et doivent faire face à des coupures de courant intempestives. L'idée principale est, au lieu de recommencer une application depuis le début après chaque coupure de courant, d'intégrer les coupures au cycle de vie des applications. En d'autres termes, cette thèse s'intéresse aux moyens de répartir l'exécution d'applications sur plusieurs *périodes de courant*, *i.e.*, périodes entre deux coupures de courant. En particulier, ce travail apporte plusieurs contributions dans le domaine des objets à alimentation intermittente, du point de vue des systèmes d'exploitation.

Beaucoup de composants de ces plates-formes sont volatiles et perdent leurs contenus lors d'une coupure de courant. Elles embarquent alors de la mémoire non-volatile pour sauvegarder un état cohérent de l'application. Les contraintes énergétiques des appareils visés par cette thèse sont telles qu'ils peuvent rester éteints pendant des heures, voire des jours, ce qui n'est pas nativement prévu par les langages de programmation classiques tels que le C.

Les coupures de courant sont à l'origine de problèmes particuliers. L'état du micro-contrôleur et le contenu des mémoires volatiles sont perdus en l'absence de courant (problème P1). Il en va de même pour les périphériques et les données qui y sont reliées (problème P2), telles que celles portées par les interruptions matérielles. Les périphériques ne sont pas aussi simples à restaurer que la mémoire et nécessitent un traitement à part. En effet, certains accès aux périphériques requièrent des séquences spécifiques d'opérations ou des délais d'attente non-interruptibles. De surcroît, les états du micro-contrôleur, des mémoires volatiles et non-volatiles doivent être maintenus cohérents les uns avec les autres, en toutes circonstances. Le scénario se complique lorsque l'on intègre une composante temporelle aux opérations logicielles et matérielles (problème P3). D'un point de vue matériel, certaines opérations ne peuvent être interrompues par une coupure de courant, comme lorsque l'on envoie un paquet radio. L'opération doit s'exécuter de manière ininterrompue car reprendre l'envoi au milieu du paquet après un redémarrage n'a pas de sens. De même, d'un point de vue logiciel, il est légitime de se demander quelle serait la pertinence de garder une valeur lue sur un capteur après une longue coupure de plusieurs heures. Certaines

applications nécessitent alors que le traitement de données relevées par des capteurs s'exécute dans la même période de courant que l'opération d'échantillonnage correspondante, et doivent alors retenter le processus dans son intégralité si une coupure survenait.

Les travaux qui adressent les problèmes liés à l'intermittence ont recours au *checkpointing* afin de répartir l'exécution d'applications sur plusieurs périodes de courant. Les travaux qui proposent une solution au problème P3, *i.e.*, celui des opérations non-interruptibles ainsi que de l'expiration des données, ont souvent recours à des retours en arrière dans le flot de contrôle de l'application. Ces retours en arrière ne sont pas observés dans une exécution exempte de coupures de courant, ce qui induit des mouvements de flot de contrôle imprévus par la spécification logicielle. Les langages de programmation et leurs compilateurs font l'hypothèse que les instructions se suivent séquentiellement et que les mouvements de flot de contrôle sont explicites, *via* des structures de contrôle comme `if`, `while`, ou des appels de fonctions par exemple. Les retours en arrière infirment ces hypothèses, ce qui peut aboutir à une corruption des données, notamment en mémoire non-volatile, car certaines opérations non-idempotentes peuvent être amenées à être rejouées, et donc produire un résultat incorrect étant donné le caractère non-idempotent de ces opérations (problème P4).

Les travaux existants aujourd'hui qui visent à résoudre les problèmes P1, P2, P3 ou P4 reposent sur des architectures variées. La plus populaire embarque, à la fois, de la mémoire volatile et de la mémoire non-volatile, car les performances des mémoires non-volatiles sont encore en deçà de celles de leurs alternatives volatiles, pour le moment. D'autres optent pour une mémoire de travail totalement non-volatile. Enfin, certains travaux étudient la conception de micro-contrôleurs, en tout ou partie non-volatiles, afin d'étendre les propriétes non-volatiles à l'unité de calcul en elle-même, mais ils sont toujours à l'état de prototypes expérimentaux. Indépendamment de l'architecture choisie, les checkpoints sont placés statiquement ou dynamiquement sur expiration d'un timer ou sur détection de coupure de courant imminente. D'autres travaux conçoivent des applications sur un modèle multi-tâche, dans lequel les frontières des tâches sont des checkpoints par construction. Le problème de la volatilité des périphériques (P2) n'est pas souvent abordé, et les travaux qui s'y intéressent ont recours à l'enregistrement de séquences d'opérations périphériques, de manière plus ou moins compressée.

Sytare est le premier système d'exploitation léger à apporter une solution aux problèmes P1, P2 et P3. Son objectif est de permettre l'écriture d'applications qui diffèrent le moins possible du modèle d'applications sur machine-nue. Sytare y parvient grâce à son mécanisme d'appel de driver au travers d'une API qui résout les problèmes P2 et P3 de manière transparente pour les développeurs d'applications. En revanche, les développeurs de drivers doivent fournir quelques routines pour chaque driver : pour l'initialisation du périphérique, ses interruptions, sa sauvegarde et sa restauration. Les deux premières seraient présentes, de tout manière, dans une application sur machine-nue, et donc seules les routines de sauvegarde et de restauration sont réellement imposées par Sytare. Les périphériques (P2) sont sauvegardés de manière incrémentale, à la fin de l'exécution de chaque routine driver qui change l'état dudit périphérique. A l'entrée d'une routine driver, Sytare enregistre l'appel et ses arguments de sorte à pouvoir ré-exécuter la routine depuis le début si une coupure de courant interrompt l'opération (P3). Le micro-contrôleur et la mémoire (P1) sont sauvegardés uniquement sur détection de coupure imminente. Avant cette thèse, Sytare existait déjà mais ne permettait pas encore les interruptions, essentielles dans le développement d'applications sur machine-nue. La présente thèse y contribue alors, en ajoutant la persistence des interruptions et des données associées, ainsi qu'un portage de Sytare sur la plate-forme ARMorik. ARMorik a été conçue et assemblée dans le cadre de ce travail, afin de fournir une alternative aux plates-formes MSP430, en proposant une architecture basée sur ARM ainsi que de plus grandes capacités de stockage, volatile comme non-volatile, afin de prévoir l'avènement de futures plates-formes.

Cette thèse propose en outre un modèle et une méthodologie afin d'estimer, de manière précise, la consommation énergétique de logiciels déployés sur plates-formes embarquées. En utilisant une machine à états distincte pour chaque composant de la plate-forme, le modèle n'a pas besoin de construire de machines à états combinatoires. Pour donner des estimations, le modèle est peuplé avec des mesures effectuées sur plate-forme réelle. Ce travail montre qu'il est possible d'obtenir des estimations précises en utilisant une plate-forme de mesure simple. Le modèle est ensuite implémenté dans un simulateur en deux parties, prenant en entrée les mêmes binaires que ceux qui sont exécutés sur machine réelle. Les parties logicielles exemptes d'utilisation de périphériques sont simulées au cycle près, tandis que les opérations sur les périphériques sont simulées de manière symbolique à la granularité d'une routine driver. L'exécution symbolique consiste à changer l'état du périphérique tel que spécifié par la documentation de

la routine, et à faire avancer le temps et l'énergie dans la simulation, de sorte à ce que la durée d'exécution et la consommation énergétique soient équivalentes à une observation réelle. Le modèle énergétique a donc besoin d'un modèle pour chaque routine driver. En pratique, il s'agit de simples constantes, mais cela dépend de l'API driver considérée. Certaines routines nécessitent un modèle un peu plus complexe, comme l'envoi d'un paquet radio dont le temps d'exécution et la consommation énergétique sont linéaires avec la taille du paquet à envoyer. Le simulateur permet d'estimer le temps d'exécution, la consommation énergétique, les endroits probables de coupure de courant, et sert aussi à conduire des expériences à plus grande échelle car il est facile d'exécuter plusieurs centaines d'instances du simulateur sur une machine, plutôt que de réaliser le même nombre de tests sur plates-formes réelles.

Dans les applications dont les périodes de courant sont courtes, il est peu probable que beaucoup de mémoire soit modifiée entre deux coupures de courant. Il est donc possible d'optimiser les réponses usuelles au problème P1, de sorte à ne copier que les parties de la mémoire qui ont été modifiées. Sytare a alors été amélioré, pour que sa sauvegarde de mémoire devienne incrémentale, à l'instar des périphériques, la différence étant que la mémoire n'est sauvegardée qu'une fois par période de courant. En utilisant une Memory Protection Unit (MPU), la mémoire est découpée en régions que Sytare configure en lecture seule à chaque démarrage. Lorsque l'application souhaite écrire dans la mémoire, une exception d'accès mémoire est alors générée. Ensuite, Sytare marque la région correspondante comme modifiée, rend les droits d'écriture à la région mémoire, et ré-execute l'instruction ayant généré l'exception, afin que la modification puisse avoir lieu. Lors de l'interruption de coupure de courant imminente, Sytare ne copie que les régions qui ont été marquées comme modifiées durant la période de courant. Ce travail propose aussi un modèle énergétique des méthodes de checkpointing faisant interagir onze paramètres. Le modèle permet de comparer différentes approches, ainsi que d'aider à la conception de nouveaux mécanismes de sauvegarde. En prévoyant que les futures plates-formes embarqueront davantage de mémoire, il deviendra de plus en plus impératif d'avoir recours au checkpointing incrémental.

Enfin, le domaine des systèmes à alimentation intermittente a besoin de preuves formelles quant à l'exécution correcte d'applications en dépit des coupures de courant. En partant du principe que P1 est déjà correctement géré par les systèmes actuels, on peut définir une spécification de l'exécution d'applications comme une trace d'opérations sur les périphériques. Il s'agit de prouver que les traces réellement observées correspondent à la spécification, grâce au concept de raffinement de trace. La résolution de P3 implique des retours en arrière, et notamment les potentielles opérations sur périphériques doivent être rejouées. Le raffinement de trace doit alors tenir compte de ces retours en arrière et, si l'impact des opérations périphériques sur l'environnement est idempotent, alors cela n'affecte pas la preuve. En pratique, certaines opérations ne sont pas idempotentes, comme l'envoi d'un paquet radio. Mais dans ce cas précis, les communications sont régies par une pile réseau dont les protocoles prévoient, entre autres, la redondance de paquets. La preuve introduite dans ce travail se calque sur des systèmes à sauvegarde lors de coupures de courant imminentes, et prouve l'exactitude des systèmes KARMA et Sytare en l'état, ainsi que RESTOP si on lui implémente une solution à P3.

Cette thèse se positionne dans une vision des systèmes à alimentation intermittente orientée vers les systèmes d'exploitation, bien que rejoignant d'autres domaines tels que l'architecture et l'électronique, avec la conception et le développement d'ARMorik ainsi que de plates-formes de tests et de mesures. Il est possible d'étendre les travaux présentés dans cet ouvrage. Les données énergétiques peuvent servir à mettre au point un ordonnanceur intelligent. Le checkpointing incrémental peut être davantage optimisé en restaurant les régions mémoire uniquement lorsqu'elles sont lues, afin de ne pas avoir à restaurer des régions qui ne seraient pas utilisées pour une période de courant donnée. L'avènement de nouvelles plates-formes embarquées contenant davantage de mémoire et les avancées technologiques à prévoir dans les mémoires non-volatiles pourront remettre en question certains choix actuels de placement des données en mémoire. Il sera intéressant d'étudier les améliorations rendues possibles par ces nouvelles architectures.

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : BERTHOU

DATE de SOUTENANCE : 29/03/2021

Prénoms : Gautier, Philippe

TITRE : Operating system dedicated to NVRAM-based low power embedded systems

NATURE : Doctorat

Numéro d'ordre : 2021LYSEI021

Ecole doctorale : 512

Spécialité : Informatique

RESUME :
Certains objets électroniques ne peuvent embarquer de pile ou de batterie, pour des raisons de place ou de coût. C'est le cas de petits objets, récoltant de l'énergie depuis leur environnement, afin de pallier l'absence de batterie. Capteurs médicaux et sondes agronomiques en sont des exemples. En général, ces objets consomment de l'énergie plus rapidement qu'ils n'en récoltent, notamment pour les appareils non équipés de cellules photovoltaïques. En résultent de fréquentes coupures de courant lors de l'exécution de l'application, on parle alors d'alimentation intermittente. Plutôt que de recommencer l'application du début à chaque redémarrage, il est possible de développer des mécanismes afin de reprendre l'exécution à l'endroit où elle s'était arrêtée. L'utilisation de mémoires non-volatiles, qui ne perdent pas leurs données lorsque la plate-forme est éteinte, permet le stockage de données en dépit des coupures de courant. En particulier, les RAM non-volatiles (NVRAM) ont des temps d'accès voisins des RAM volatiles, ce qui permet de les utiliser comme des mémoires courantes, à l'opposé des mémoires type Flash dont le temps et l'énergie d'écriture sont trop élevés. Ce travail met en exergue l'utilisation de NVRAM afin d'exécuter une application longue en dépit des coupures de courant. Il s'articule autour du développement d'un système d'exploitation visant ce type d'objet, assurant une cohérence entre les mémoires et l'état des périphériques à chaque instant. En outre, il propose un modèle de systèmes à alimentation intermittente, servant de base pour l'évaluation des coûts énergétiques de portions de code impliquant des opérations sur les peripheriques. Un nouveau mécanisme de sauvegarde basé sur une MPU matérielle est proposé pour optimiser la sauvegarde de données. Enfin, ce travail propose une introduction à la preuve formelle d'exécution correcte pour systèmes à alimentation intermittente.

MOTS-CLÉS : Systèmes d'exploitation, NVRAM, Consommation énergétique, Alimentation intermittente, Récolte d'énergie

Laboratoire (s) de recherche : CITI

Directeur de thèse: Tanguy Risset

Présidente de jury : Isabelle Puaut

Composition du jury : Isabelle Puaut, Lionel Torres, Brandon Lucia, Isabelle Guérin Lassous, Olivier Sentieys, Ivan Miro-Panades, Tanguy Risset, Kevin Marquet