HAL
open science

Microarchitectures for robust and efficent incremental backup in intermittently powered systems

Davide Pala

To cite this version:

Davide Pala. Microarchitectures for robust and efficent incremental backup in intermittently powered systems. Hardware Architecture [cs.AR]. Université de Rennes, 2022. English. NNT : 2022REN1S100 . tel-04077508

HAL Id: tel-04077508

https://theses.hal.science/tel-04077508v1

Submitted on 21 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Davide PALA

Microarchitectures for Robust and Efficient Incremental Backup in Intermittently Powered Systems

Thèse présentée et soutenue à Rennes, le 10 novembre 2022
Unité de recherche : Inria/IRISA

Rapporteurs avant soutenance :

Abdoulaye Gamatié	Directeur de Recherche, CNRS, LIRMM, Montpellier
Tanguy Risset	Professeur, INSA Lyon, CITI

Composition du Jury :

Président :	*(à préciser après la soutenance)*	
Examinateurs :	Abdoulaye Gamatié	Directeur de Recherche, CNRS, LIRMM, Montpellier
	Tanguy Risset	Professeur, INSA Lyon, CITI
	Maria Méndez Real	Maitre de Conférences, Polytech Université de Nantes, IETR
	Daniel Etiemble	Professeur, Université Paris-Saclay, LRI
Dir. de thèse :	Olivier Sentieys	Professeur, Université de Rennes, Inria, IRISA
Co-dir. de thèse :	Ivan Miro-Panades	Ingénieur-Chercheur, CEA-List, Grenoble

Invité :

Erven Rohou	Directeur de Recherche, Inria, IRISA, Rennes

Résumé de la Thèse

L'explosion de l'Internet des objets (IoT), des nœuds de capteurs sans fil et des appareils portables a suscité un intérêt accru pour la récupération d'énergie en tant que source d'alimentation de ces appareils. Beaucoup de ces systèmes ne peuvent pas se permettre la présence d'une batterie pour alimenter l'application. Cela peut être dû à des exigences strictes en matière de coût, de taille et/ou de poids du système. De plus, l'utilisation d'une batterie pour les appareils dont la durée de vie est longue peut être difficile car les batteries ont généralement une courte durée de vie et doivent être remplacées fréquemment, ce qui rend l'entretien et la maintenance de l'appareil coûteux, en particulier lorsque ces systèmes sont utilisés dans des endroits éloignés. Dans ces cas, la récupération de l'énergie de l'environnement peut être une alternative viable pour alimenter ces systèmes.

L'inconvénient fondamental de l'utilisation de l'énergie extraite de l'environnement pour alimenter un système est que l'énergie est souvent limitée et sporadique. Cela implique que les appareils connaîtront des pannes de courant imprévisibles pendant qu'ils remplissent leur fonction. Le désir de continuer à calculer tout en utilisant une alimentation intermittente a accru l'intérêt pour les ordinateurs non volatils, en particulier les processeurs non volatils (NVP). Les ordinateurs non volatils peuvent être définis comme suit :

ordinateurs non volatils ou *Non-Volatile Computers*

"systèmes qui peuvent presque instantanément sauvegarder leur état de manière non volatile afin que le fonctionnement puisse continuer même en cas d'interruption de courant imprévue" [Yu+11]

La Figure 1 montre l'architecture système d'un appareil typique alimenté par intermittence et sans batterie. Comme le montre la Figure 1, ces systèmes sont généralement construits autour des éléments de base ci-dessous : 1) Le système de récupération d'énergie, qui puise l'énergie de l'environnement et la fournit aux autres composants du système. 2) Un condensateur tampon est utilisé pour stocker une petite quantité d'énergie, afin de permettre au système de continuer à fonctionner et de sauvegarder son état, lorsque le

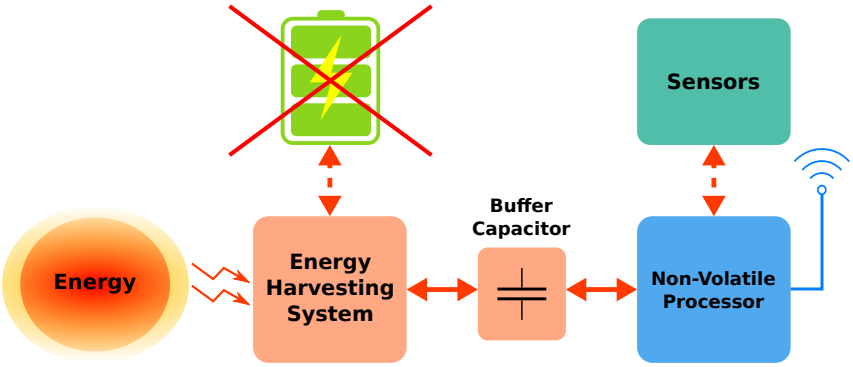


FIGURE 1 – Architecture de haut niveau d'un dispositif sans batterie, alimenté par intermittence.

système de récupération est incapable de fournir de l'énergie. La tension aux bornes du condensateur tampon est fréquemment utilisée pour évaluer la « disponibilité » de l'énergie et pour identifier le moment où une perte de puissance est sur le point de se produire. La perte de puissance est généralement identifiée par la tension aux bornes du condensateur tampon tombant en dessous d'un seuil prédéterminé. 3) Le processeur non volatil (NVP), qui est le système informatique principal, généralement connecté à certains capteurs et à une connectivité réseau via des appareils radio à faible puissance.

La figure 2 montre un exemple d'exécution intermittente, où l'exécution est interrompue par des coupures de courant, représentées par des chutes de tension et divisées en intervalles. Le calcul intermittent est donc une séquence de quatre états d'un appareil : *éteint (power-off)*, *restauration (restore)*, *calcul (compute)* et *sauvegarde (backup)*. Afin de préserver la progression de l'exécution du système en cas de panne de courant et d'éviter de redémarrer depuis le début (ou de *rebooter* le système), le NVP doit effectuer une *sauvegarde* de l'état du système avant que l'alimentation ne soit complètement coupée et que l'appareil soit *éteint*. Lorsqu'il y a assez d'énergie pour continuer l'exécution, l'état précédent est *restauré*.

Un autre problème des appareils alimentés par intermittence est de maintenir l'état non volatil cohérent. Des problèmes de cohérence surviennent lorsque, après une sauvegarde, la mémoire non volatile (NVM) est modifiée et que l'alimentation est coupée avant qu'une nouvelle sauvegarde ne soit exécutée. Ce type d'erreur inclut également les erreurs dues à une interruption soudaine pendant le processus de sauvegarde, qui laissent l'état de sauvegarde dans la NVM inachevé. De nombreuses solutions dans l'état de l'art, et en

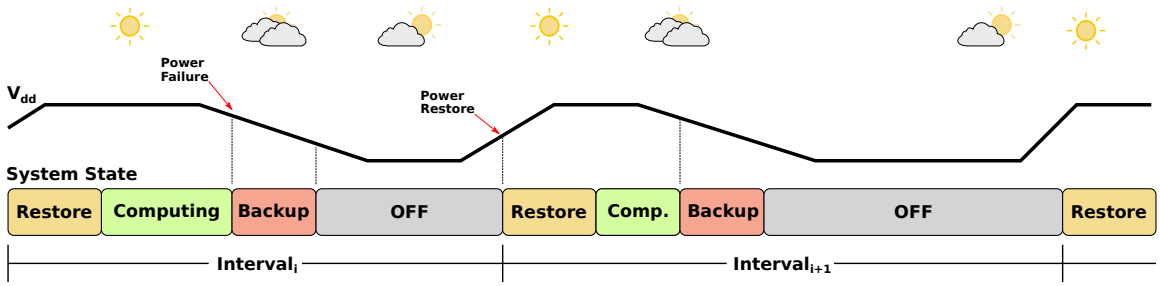


FIGURE 2 – Exemple de calcul en présence d'alimentation intermittente.

particulier de nombreux processeurs entièrement non volatils, n'offrent aucune détection ou protection contre les erreurs de sauvegarde ou les erreurs d'incohérence. Certaines solutions sont capables de détecter ces erreurs, mais ne peuvent pas récupérer et nécessitent un redémarrage de l'application.

Ce travail introduit d'abord les technologies de mémoire non volatile (NVM). Le fonctionnement de base de ces NVM est décrit en général, et les avantages et les limites de chaque technologie sont mis en évidence. Ensuite, en utilisant les informations recueillies à partir de diverses revues de la littérature, nous donnons une comparaison entre les différentes types et technologies de NVM et d'autres technologies de mémoire comme SRAM, DRAM et FLASH.

Dans cette thèse, nous décrivons des méthodes qui s'appuient sur un contrôleur matériel dédié pour permettre une informatique alimentée de manière transitoire avec des sauvegardes à la demande rapides et efficaces sur une plate-forme SRAM avec NVM pour les sauvegardes. En utilisant une conception basée sur SRAM + NVM, il est montré qu'il est possible de conserver les avantages de l'exécution rapide et économe en énergie de SRAM tout en limitant les inconvénients des NVM, tels que leur faible endurance, leur énergie active élevée et leur temps d'accès plus lent.

Dans cette thèse, nous présentons *Freezer*, un contrôleur spécialisé pour la sauvegarde et la restauration qui surveille dynamiquement les accès mémoire tout au long de l'exécution du programme. Lorsqu'une coupure de courant est détectée, le contrôleur valide les modifications sur une copie dans une mémoire non volatile (NVM). Notre procédé peut être mis en œuvre en utilisant des standards et ne fait pas appel à des mémoires hybrides sophistiquées. Freezer utilise un tableau de bits comme bitmap, pour marquer

les sections de la mémoire qui sont modifiées. L'espace d'adressage est divisé en blocs consécutifs, chaque bloc est constitué de N mots de 32 bits (par exemple, un espace d'adressage de 32 Ko peut être divisé en 1024 blocs de 8 mots). En fonctionnement normal, lorsque le CPU écrit à une adresse, Freezer identifie le bloc qui contient l'adresse et met à 1 le bit correspondant dans le bitmap. Lorsqu'une coupure de courant est détectée, Freezer lit le bitmap et copie uniquement les blocs modifiés depuis la SRAM vers la NVM.

Les résultats sur un ensemble de benchmarks montrent une réduction moyenne de $8\times$ de la taille de la sauvegarde. Grâce à notre contrôleur dédié, le temps de sauvegarde est encore réduit de plus de $100\times$, avec un ajout de surface et de puissance de seulement $0,4\%$ et $0,8\%$, respectivement, par rapport à système sans Freezer.

Les sauvegardes et restaurations d'état sont des opérations cruciales dont l'interruption pourrait compromettre la progression ou éventuellement entraîner la corruption de l'état de l'application. Pour résoudre ce problème, nous présentons deux algorithmes de sauvegarde et de restauration qui, à chaque étape, peuvent garantir l'existence d'un état cohérent du système vers lequel se replier en cas d'erreur lors de la sauvegarde. De plus, nous montrons que cette garantie nous permet d'améliorer encore l'efficacité énergétique du système en permettant des temps d'exécution plus longs avec la même valeur de condensateur. Par rapport à une approche *double-buffering* conventionnelle, les deux algorithmes que nous proposons permettent d'économiser au total plus de 23% d'énergie et de temps d'exécution.

L'approche bitmap est efficace pour suivre les modifications dans un espace d'adressage relativement petit, mais cette approche est difficile à mettre à l'échelle pour des mémoires plus grandes, car elle nécessite soit d'utiliser plus de bits, ce qui entraîne une surcharge de zone, soit de suivre de plus grandes sections de mémoire avec le même nombre de bits. Le suivi d'une grande section de mémoire réduit la précision de la sauvegarde différentielle (c'est-à-dire que davantage de mots non modifiés sont enregistrés), augmentant ainsi sa taille.

Pour améliorer l'évolutivité des systèmes de sauvegarde incrémentale tels que Freezer, sans augmenter la taille des sauvegardes, une nouvelle méthode pour suivre les régions de mémoire modifiées est proposée. Pour cela, nous définissons et étudions des structures de données abstraites et probabilistes telles que les *filtres Bloom*, en combinaison avec notre approche Freezer. Cette combinaison Bloom+Freezer permet de suivre des espaces d'adressage beaucoup plus grands, tout en améliorant la taille de la sauvegarde, même par rapport à une version évolutive et beaucoup plus chère de Freezer.

La thèse est organisée comme suit. Le chapitre 1 présente un aperçu de l'état de l'art, en analysant les forces et les faiblesses des approches trouvées dans la littérature. Le chapitre 2 présente une revue des technologies émergentes de mémoire non volatile. Dans le chapitre 3, Freezer, un contrôleur de sauvegarde implémentant la sauvegarde incrémentale à la demande, est proposé, conçu et validé. Le chapitre 4 présente deux algorithmes de sauvegarde incrémentale qui garantissent la cohérence de l'état du système. Le chapitre 5 étudie l'utilisation de filtres Bloom pour améliorer et étendre le suivi de la mémoire, pour un schéma de sauvegarde incrémentielle. Enfin, le chapitre 5.6 résume ce travail, et présente une discussion sur certaines perspectives futures intéressantes.

Table of Contents

Introduction **13**

1 State of the Art **19**

1.1 Non-Volatile Computers and Energy Harvesting 19

1.1.1 Scarce and Intermittent Energy Sources 19

1.1.2 Non-Volatile System Design Objectives 21

1.2 Solutions for Backup Robustness and Consistency 23

1.3 Software Solutions 25

1.3.1 Programming Languages and Task-Based Approaches 25

1.3.2 Static Techniques 26

1.3.3 Run-Time Techniques 28

1.3.4 Sytare 33

1.4 Architectural (Hardware) Solutions 34

1.4.1 NVM Integration 34

1.4.2 Brainshift Architecture 36

1.5 Non-Volatile Processors 38

1.5.1 MSP430F : an MCU with Embedded FeRAM 38

1.5.2 THU1010N : a Non-Volatile-Processor based on FeFFs 40

1.5.3 Non-Volatile logic Cortex-M0 with distributed FeRAM mini arrays 41

1.5.4 A 90nm 20MHz nvMCU based on STT-RAM 43

1.5.5 A 65 nm ReRAM-Enabled NVP 44

1.5.6 RRAM Non-Volatile Intelligent Processor 46

1.5.7 FeRAM Parallel Recovery SoC 47

1.6 Conclusion 48

2 Non-Volatile Memories **53**

2.1 Introduction 53

2.2 Phase Change Memory 53

2.2.1 SET and RESET write operations 54

2.2.2 PCM Cell 55
2.2.3 Characteristics 56
2.3 Magnetic RAM 57
2.3.1 Magnetic Tunnel Junction 57
2.3.2 Field Induced Magnetic Switching MRAM 57
2.3.3 Thermally Assisted Switching MRAM 58
2.3.4 Spin-Transfer Torque MRAM 59
2.4 Resistive Memories and Memristors 60
2.5 Ferroelectric RAM 64
2.6 Comparison 66

3 FREEZER a Dedicated Backup and Restore Controller 71
3.1 Introduction 71
3.2 Background and Related Work 73
3.3 System Modelling 76
3.3.1 Considered System Model 76
3.3.2 System Architecture 77
3.3.3 Modelling Memory Access Energy 79
3.4 Modeling of the Backup Strategies 82
3.4.1 Full Memory Backup 83
3.4.2 Used Address Backup 83
3.4.3 Modified Address Backup 83
3.4.4 Oracle 83
3.4.5 Block-Based Strategies 85
3.5 Trace Analysis and Improvement in Backup Size 85
3.6 Freezer 88
3.6.1 Freezer Architecture 88
3.6.2 Area and Power Results 91
3.6.3 Impact of Block Size 92
3.7 Results 93
3.7.1 Backup Size 93
3.7.2 Impact of Interval Size 93
3.7.3 Backup Time 96
3.7.4 Energy Comparison with other Memory Models 97

3.7.5 Impact of Leakage Power . . . 101
3.7.6 Energy and Area Overhead Considerations . . . 102
3.8 Discussion About The Approach . . . 104
3.9 Conclusion . . . 105

4 Robust and Consistent Incremental Backup for Intermittently Powered Systems 107
4.1 Introduction . . . 107
4.2 Architecture Model . . . 111
4.3 Related Work . . . 114
4.4 Robust and Consistent Backup . . . 116
4.4.1 Full Backup with Double Buffering . . . 116
4.4.2 Restore & Update Strategy . . . 117
4.4.3 Cumulative Updates Strategy . . . 119
4.5 Robustness Against Power Failures . . . 122
4.5.1 Robust Backup and Restore Algorithms . . . 122
4.5.2 Probabilistic Failure Model . . . 123
4.6 Results . . . 127
4.6.1 Experimental Setup and Simulation Procedure . . . 127
4.6.2 Probability of backup interruption . . . 129
4.6.3 Analysis of Memory and Backup Data Size . . . 130
4.6.4 Overhead Analysis . . . 133
4.7 Exploiting Robust Backup to Rise System Efficiency . . . 134
4.8 Conclusion . . . 138

5 Bloom-Filter Based Memory Write Tracking for Differential Backup 141
5.1 Introduction . . . 141
5.2 Background and Related Work . . . 143
5.2.1 Incremental Backup . . . 143
5.2.2 Hashing and Bloom Filters . . . 145
5.3 System Model . . . 147
5.4 Filter-Based Memory Write Tracking . . . 147
5.4.1 Plain Bloom Filters . . . 147
5.4.2 Parallel Bloom Filters . . . 148
5.4.3 Hybrid Hierarchical Bloom-Freezer . . . 149

5.5 Results 151
5.5.1 Simulation Setup 151
5.5.2 Simulation Results 152
5.5.3 Hardware Model 153
5.6 Conclusion 157

Conclusion **159**

Bibliography **163**

Introduction

The explosion of Internet-of-Things (IoT), wireless sensor nodes and wearable devices, has lead to an increasing interest in energy harvesting as a source for powering these devices. Many of these systems cannot afford the presence of a battery to power the application. This can be due to strict requirements with respect to cost, size and/or weight of the system. Moreover, the use of a battery can be difficult for devices with an expected long lifetime, as batteries usually have low endurance and require frequent replacements, making the device expensive to service and maintain, especially when such devices are deployed in remote locations. In these cases, environmental energy harvesting can be a viable alternative to power these devices.

The main drawback of powering a device with energy harvested from the environment, is that the available energy is usually scarce and intermittent. This means that the device, while executing its task, will suffer from unpredictable power interruptions. The idea of sustaining computation under intermittent power has pushed forward the interest towards *Non-Volatile Computers* and in particular *Non-Volatile Processors* (NVP). Non-volatile computers can be defined as in [Yu+11] :

Non-Volatile Computer

"systems that can almost instantly save their state in a non-volatile fashion so that operation can continue even across an unanticipated power interruption" [Yu+11]

In recent years, the emergence of faster *Non-Volatile Memories* has enabled the development and research of NVPs for intermittent computing devices powered with environmental energy harvesting.

Figure 3 shows the block-level architecture of a typical battery-less, intermittently-powered device. These systems are usually based around the following basic components, as shown in Figure 3 : 1) The *Energy Harvesting System*, which extracts some form of environmental energy and feeds it to the rest of the system. 2) A *Buffer Capacitor* that is

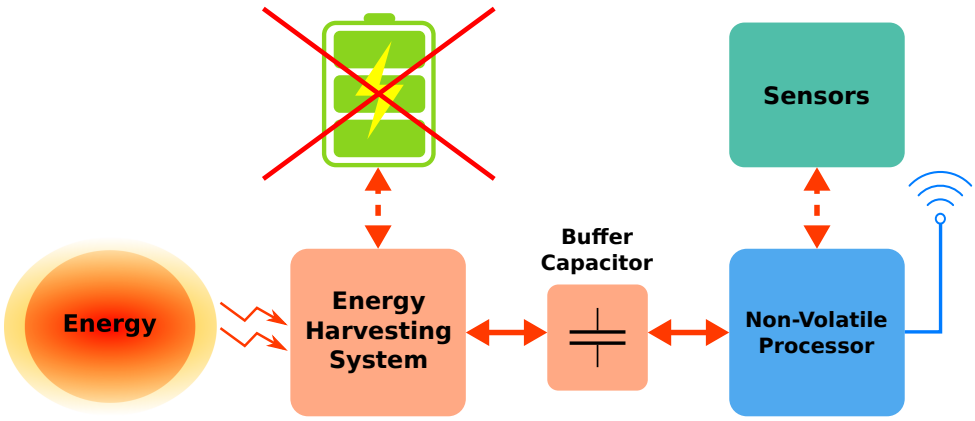


FIGURE 3 – High level architecture of a battery-less, intermittently-powered device.

used to store a small amount of energy, to allow for the system to continue its execution or to save its state when the harvester is not able to provide power. The voltage across the buffer capacitor is commonly used to measure the "availability" of the energy and to detect an imminent power loss, which is usually detected when the voltage across the buffer capacitor falls below a certain threshold. 3) The *Non-Volatile Processor*, which is the main computing device and is usually connected to some sensors and some kind of network connectivity, usually through low-power radio devices.

Figure 4 shows an example of intermittent computing, with the execution interrupted and divided into intervals by power failures, which are shown as drops in the voltage. Intermittent computing is therefore a sequence of four states of a device : *power-off*, *restore*, *compute*, and *backup*. In order to preserve progress of the system execution across power failures and to avoid restarting from the very beginning (or to reboot the system), the NVP has to perform a *backup* of the system state before the power is completely lost and the device is *off*. Then, when there is again enough energy to continue execution, the previous state is *restored*.

Many solutions have been proposed to implement efficient non-volatile computing systems, that can carry computation across unpredictable power interruptions. One of the main characteristics of these systems is the backup (or check-pointing) strategy, which can either be : *On-demand*, meaning that the backup operation is triggered by an event signaling an imminent power failure, or *Static/Periodic*, in which case the check-points are placed in predetermined locations in the code of the application.

Static check-pointing strategies use compile-time techniques to optimize aspect of the

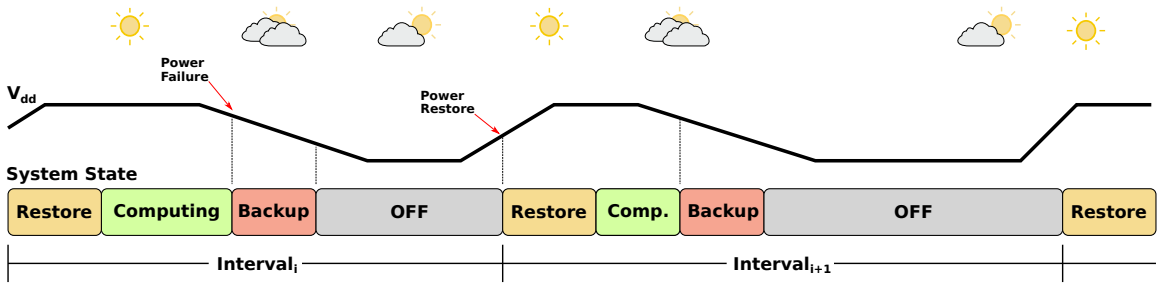


FIGURE 4 – Example of computation under intermittent power.

backup such as the size (*i.e.*, the amount of data that needs to be saved), and the number and placement of the check-points. The main drawback with this approach is that the check-points are executed even when the energy is available and, in the event of a power failure, the whole progress made from the last check-point is lost.

On-demand check-pointing on the other hand tries to execute the backup only when necessary (*i.e.*, just before the power fails). With this kind of strategies, the system responds to an event such as the voltage across the buffer capacitor falling below a predetermined threshold. In this case, the backup could be executed at any moment during the execution of the application. It is therefore difficult to optimize in advance the size of the backup. For this reason, most on-demand check-pointing strategies usually have to save the whole memory every time.

We can also categorize non-volatile computing systems by whether they are based on software-level or hardware-level solutions. Software-based approaches are usually implemented on platforms with both SRAM and an addressable Non-Volatile Memory (NVM), *e.g.*, like the commercially available msp430F [Ins21]. These solutions are usually easier to implement and to apply to existing applications, but tend to be much slower and less efficient than the hardware-based approaches.

Hardware-level solutions on the other hand exploit NVM integration, usually to implement fully non-volatile processors, where every memory element, including the internal registers, are made non-volatile. This is usually achieved by using an NVM as the only system memory, and using hybrid non-volatile flip-flops (nvFFs) (*i.e.*, flip-flops paired with a non-volatile element) for the internal registers. The aim with this kind of solutions is to minimize the backup and restore time, achieving a system that is able to respond almost instantly to power failures, and to resume as quickly as possible the execution when the power is back. These systems also come with several disadvantages, as they are difficult

to implement and adopt, due to the use of exotic technologies and need for intrusive and extensive modifications of the processor architecture, which cannot be applied to existing licence IPs and hard IPs. Moreover, these solutions usually come with great overheads in terms of area of the chip, due to nvFFs being much larger than regular FFs and much more difficult and expensive to manufacture.

Systems based only on NVM, both hardware and software based ones, also face additional problems due to the limitations of current NVM technologies, which are explored more in depth in Chapter 2. In particular, NVMs offer slower read and write access speed and increased read and write energy consumption with respect to volatile, static random-access memory (SRAM). This translates into worse performance and increased power consumption during the active times (*i.e.*, *computing* phases in Figure 4), resulting in slower progress and more interruptions. Additionally, NVMs currently suffer from limited endurance, as the NVM cells can wear out and stop working in a limited number of write cycles.

Another problem of intermittently-powered devices is to keep the non-volatile state consistent (*i.e.*, safe). Consistency problems arise when, after a backup, the NVM is modified, and power is lost before a new backup is executed. This type of error also includes errors due to sudden interruption during the backup process, which leave the backup state in the NVM uncompleted. Many solutions in the state of the art, and in particular many fully non-volatile-processors, do not offer any detection or protection against backup errors or inconsistency errors. Some solutions are able to detect these errors, but cannot recover and require a restart of the application.

This work was part of the *ZEro Power (ZEP)* computing systems project. The ZEP project gathers four INRIA teams that have a scientific background in architecture, compilation, operating system and low power, together with the CEA Lialp and Lisan laboratories of CEA LETI & LIST. The ZEP project addresses the issues related to designing small, battery-less computing nodes, in the context of IoT and ubiquitous computing, by combining non-volatile memory, energy harvesting, micro-architecture innovations, compiler optimizations, and static analysis. In the context of the ZEP project, this work has focused mainly on the micro-architectural enhancements for battery-less, intermittently powered systems.

In this work, we start from the idea of implementing an on-demand backup solution on a platform equipped with both SRAM and NVM, that uses primarily the SRAM for normal computations and the NVM for backups. Using an SRAM+NVM based architecture allows to keep the advantages of fast and energy efficient execution on SRAM, while minimizing the drawbacks of NVMs such as the low endurance. Results in Chapter 3 provide some insights about the benefits of such SRAM+NVM based architectures.

First, we propose Freezer in Chapter 3, a hardware controller to implement an efficient incremental backup technique for on-demand check-pointing. The choice of an on-demand approach guarantees a more reactive system, that does not execute unnecessary backups, and minimizes the need for rollbacks after restore. Moreover, thanks to our proposed incremental backup approach, our method optimizes the backup size by saving only the modified blocks, and achieves an 87% average reduction in backup size with respect to full memory backups.

Moreover, to address the possibility of consistency errors, due to sudden interruption during backup, we further improve on our approach, and we present two algorithms that provide robust and consistent on-demand incremental backups, while always guaranteeing the existence of a consistent state, to which the system can revert to, in case of errors.

Finally, we look at extending our Freezer approach to more capable systems, *i.e.*, with main memory capacity greater than $64KB$. $64KB$ is already much larger than what is commonly used in intermittently-powered devices. But, in the future, bigger devices are likely to be used for these tasks. However, to allow more capable devices to enter the intermittently-powered domain, more scalable approaches are required. To improve the scalability of incremental backup systems such as Freezer, without increasing the size of the backups, a new method to track modified memory regions is proposed. For this, we define and study approximate membership data structures such as *Bloom filters*, in combination with our Freezer approach. This allows to track much larger address spaces, while also improving the backup size, even when compared with a scaled and much more expensive version of Freezer.

To summarize, in this work, we present a summary and comparison of the different NVM technologies, with data end results collected from many works in the literature. We present an on-demand incremental backup scheme, that can reduce backup size by more than 87% with respect to full-memory backup. We propose two algorithms for robust and consistent incremental backup, that guarantee the existence of a consistent state to revert

to in case of errors. And, finally, we investigate extending incremental backup to larger address spaces, by using Bloom filters to track modified memory regions.

The rest of the thesis is organized as follows. Chapter 1 presents an overview of the State of the Art, analyzing the strength and weaknesses of the approaches found in the literature. Chapter 2, presents a review of the emerging Non-Volatile Memory technologies. In Chapter 3, Freezer, a backup controller implementing on-demand incremental backup, is proposed, designed and validated. Chapter 4 presents two incremental backup algorithms that guarantee the consistency of the system state. Chapter 5 investigates the use of Bloom filters to improve and extend memory tracking, for an incremental backup scheme. Finally, Chapter 5.6 sums up this work, and presents a discussion about some interesting future perspectives.

Chapitre 1

State of the Art

In recent years, intermittently-powered systems have been the object of research and studies from both industry and academia. Non-Volatile Processors (NVP) and, more in general Non-Volatile Computers, are at the core of these systems, and allow them to be powered by intermittent energy sources harvested from the environment. In this chapter, the concept of Non-Volatile Computer and energy harvesting for intermittently powered systems are introduced. A brief description of the fields of application for these devices is given, then the key concept, the challenges and design objectives for non-volatile system design are outlined. The chapter then gives an overview of some solutions, both at the software and hardware levels, that have been proposed in the literature to implement and optimize these systems. Then, some hardware based non-volatile processors that have been proposed in the literature are briefly presented. Finally, we compare these different solutions from the state of the art, outlining their strengths and weaknesses.

1.1 Non-Volatile Computers and Energy Harvesting

1.1.1 Scarce and Intermittent Energy Sources

Energy scavenging from the environment can be achieved through many sources with different types of devices. The main types of environmental energy sources are :

- Light (solar or indoor),
- Vibration or Motion,
- Thermal,
- Radio Waves, and
- Wind.

Common characteristics of these types of energy sources are the low amount of power, and their unpredictability on the availability of such power. In fact, many of these sources do not exceed few $\mu W/cm^2$ as reported in Table 1.1 from [Vul+10], and the availability is also

limited and subject to variations due to changes in the environmental conditions. Because

TABLE 1.1 – Harvested power of different energy sources (from [Vul+10])

Source	Harvested Power
Ambient Light	
Indoor	10 $\mu W/cm^2$
Outdoor	10 mW/cm^2
Vibration/Motion	
Human	4 $\mu W/cm^2$
Industrial	100 $\mu W/cm^2$
Thermal	
Human	30 $\mu W/cm^2$
Industrial	1-10 mW/cm^2
RF	
GSM Base Station	0.1 $\mu W/cm^2$

the power levels involved are so low, these types of systems usually rely on capacitors or small batteries to buffer the harvested energy and smooth out variations in the supply voltage, as shown in Figure 3. Additionally, to enable long running computations on energy harvesting platforms, the device must be able to maintain progress in the event of an unexpected power failure, because of the unpredictability and intrinsic variability of environmental energy harvesting.

To cope with these issues of scarce and intermittent energy sources, researchers have been focusing on a new class of devices called *Non-Volatile Processors* (NVPs) or, more in general *Non-Volatile Computers* or *Non-Volatile Systems*. One of the early attempts to address these issues was presented by Yu *et al.* in [Yu+11], where a non-volatile microcontroller, based on floating gate transistors, is proposed. In this work, the authors identified two main types of applications that a non-volatile processor would enable : running computations under unstable power sources, and allow idle power reduction while maintaining fast response times [Yu+11]. According to this definition a Non-Volatile Computer needs to guarantee forward-progress across unexpected power failures. This requires two fundamentals operations : *NV Store* and *NV Restore*. *NV Store* means saving the state of the system in a non-volatile fashion, while *NV Restore* indicates the operation of retrieving a previously saved state. To support these operations, non-volatile memories are used, to quickly save the state of the system before a power failure occurs, and to restore the state when the power is newly available.

1.1.2 Non-Volatile System Design Objectives

The design of Non-Volatile Computer system for intermittently-powered and ambient energy harvesting scenarios poses a new set of challenges and objectives with respect to the normal low-power design. In fact, while low-power or ultra-low-power design techniques are necessary to allow operation under extremely low energy sources, they are not sufficient to enable efficient computing in an intermittent system. In particular there are other important targets that need optimization in these types of systems, such as the backup and restore time and energy. Backup time and restore time need to be minimized to improve the efficiency of the system. Achieving fast backup and restore phases allows to have a responsive system, while minimizing the backup and restore energy is important so that more of the available energy can be used to further the computation during the execution phase.

Forward progress maximization is, in general, one of the main objectives, and one of the main design goals for transiently-powered systems. To this end, there are many different decisions that a designer can take when engineering these kind of NV systems. As an example, in [Ma+16], several processor architectures are explored, with different energy sources. Contrary to normal low-power systems, the paper shows how, in some situations, even a more power hungry architecture, such as an Out-of-Order processor (OoO), can yield improvement in the forward progress of the application. This is because when the environmental energy is plenty, an OoO processor can better exploit the abundance of energy, while a non-pipelined in-order processor would lag behind, without fully utilizing the incoming energy.

Another important design decision concerns the type of check-pointing strategy, which can be categorized as either *On-Demand* or *Static/Periodic*. With static check-pointing, the backups are performed when the application reaches a check-point, which is usually a predetermined place in the code of the program. In particular, some strategies define check-points in the code at compile time, in places such as at the beginning of loops or before function calls [RSF11]. To avoid always taking backups, when a check-point is reached, the voltage across the storage capacitor (see Figure 3) is measured, and the backup is taken only if the voltage is below a threshold [RSF11]. On the other hand, systems using on-demand checkpoints trigger backups in response of some events, that normally signifies that a power failure is imminent. Usually, such triggering event is given by the voltage across the buffer capacitor falling below a defined threshold. This normally requires some

dedicated circuit such as voltage comparators or analog-to-digital converters to measure the value of the voltage of the storage capacitor. On-demand check-pointing works when the platform is capable to rapidly save the volatile state in response to the trigger events.

Another design decision concerns the architecture of the non volatile memory, which can be distributed or centralized. Additionally, the non-volatile memory can be integrated in the system in different ways, Figure 1.1 from [Yu+11] depicts three possible architectural choices for NVM integration.

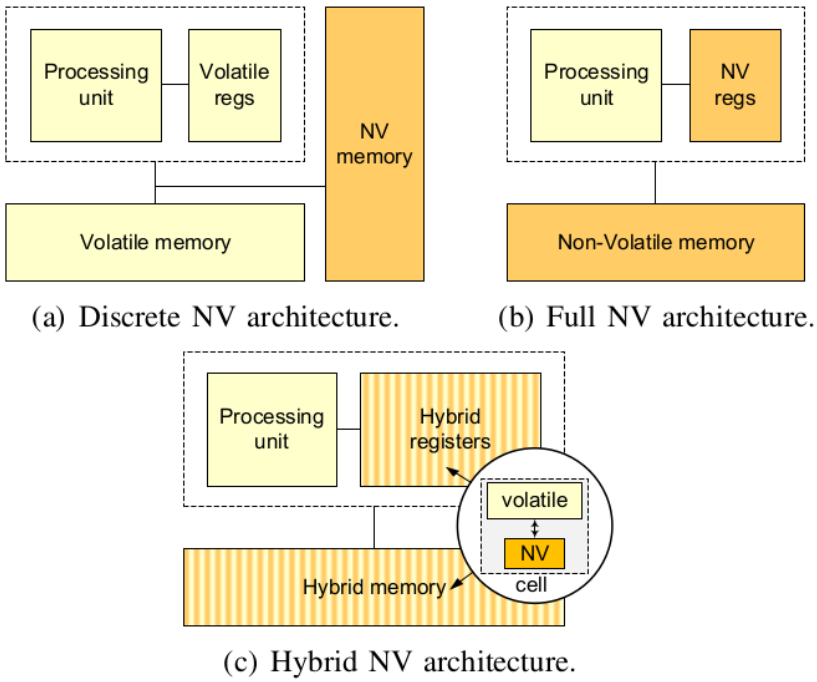


FIGURE 1.1 – NVM integration techniques for non-volatile computers (from [Yu+11])

— The *Discrete NV* approach couples a normal SRAM memory with a discrete NVM, which allows simpler integration and has the advantage of using standard memory arrays. However, the backup and restore operations are slower as they need to be performed with sequential transfers.

— The *Full NV* approach tries to solve this issue by getting rid of the volatile memories, and only using NVMs. Additionally, if also the internal registers are made non-volatile, it means that no backup operation is necessary as the main memory is already non-volatile. In practice, while it is possible to replace the main SRAM

memory with an NVM, this introduces several drawbacks as NVMs are not as fast and require more energy per operation than SRAMs. Moreover, the current NVM technologies do not allow yet the same level of endurance of SRAM, meaning that the platform could experience a much shorter life-time due to the wearing of the NVM.

— To achieve the performance and endurance of SRAM with almost instantaneous backup and restore operations, a *Hybrid NV* approach can be taken. This technique relies on using hybrid SRAM memories that are enhanced with the integration of non volatile devices for each SRAM cell [Lee+15 ; Liu+16]. Such hybrid memories can perform like SRAM during normal operations, but they can very quickly transfer the content from the SRAM cells to the non-volatile elements and vice versa. This gives the possibility of executing the backup (and restore) simultaneously on many memory cells at once. In principle, it is possible to backup the whole memory array in parallel, however some work avoid a full parallel backup, in order to limit the peak current that this operation would require [Liu+16]. This approach is also far from being mature from the technology point of view and therefore cannot be used yet as a concrete technique.

1.2 Solutions for Backup Robustness and Consistency

While the use of emerging NVMs can open the possibility for efficient intermittently powered devices, their use can also bring inconsistencies when they are paired with volatile state. According to Ransford *et al.* [RL14], two types of inconsistencies can arise when NVMs are used in an intermittent-power scenario :

— *NV-internal*, and

— *NV-external*.

Both types of inconsistencies can occur after a power failure. Internal inconsistencies are due to data stored in the NVM being only partially updated before a power failure. As an example, this can happen if the backup is not completed before the system runs out of energy. This would leave the backup in an inconsistent state and, unless protection mechanisms are applied, lead to the complete loss of progress, as it would happen in [Bal+15]. External inconsistencies instead arise when the NVM is used both for backup and to store run-time variables. They can appear after a successful backup, if a variable in the NVM is updated and a power failure happens before the next checkpoint is reached. This means

that after the restore, the volatile state is rolled-back to the previous backup, while the NVM variable remains unchanged. To address these inconsistencies, Xie et al. [Xie+15] propose a consistency-aware check-pointing algorithm, which eliminates consistency errors in a system using NVM as its main memory, and also tries to minimize the number of check-points.

Figure 1.2 shows how a consistency error can arise in systems using NVM as the main memory, when the following sequence of events happen : 1) a check-point is executed, 2) a variable is read from the NVM, 3) the variable is used for some computation, 4) the result is stored in the NVM, and 5) a power failure happens. The state (*i.e.*, the registers) is then restored from NVM, but because the power failure happened before reaching a new check-point, the state will resume from before the variable is loaded (step 2), which means that the incorrect value will be read ($n = 1$ in the example in Figure 1.2) and used. In [Xie+15],

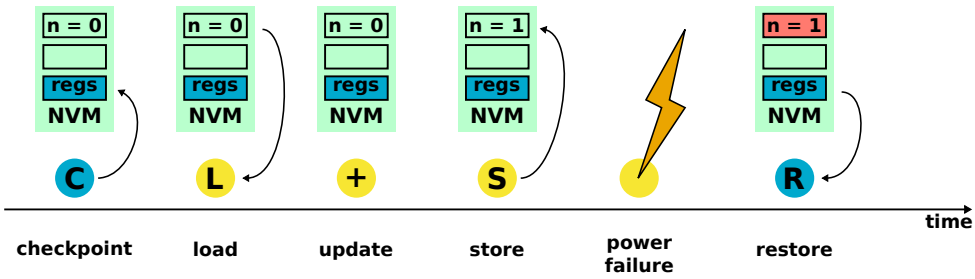


FIGURE 1.2 – Sources of consistency errors in systems using NVM as the main memory.

the authors propose to avoid these consistency errors by changing the position of the check-points, inserting a check-point in between the *load/store* error pair. The paper then presents an algorithm to identify these error pairs, and proposes an heuristic algorithm to minimize the number of required check-points. While the technique proposed in [Xie+15] addresses the inconsistency issue with NVM based systems, the approach presents some disadvantages mainly due to two main factors : the drawbacks of a system using NVM as the only system memory, the use of a static check-pointing technique. NVMs are still slower and consume more energy for read and write operation with respect to traditional SRAM. Moreover, because this is a static check-pointing scheme, the system does not respond to a power-failure events, but waits to reach a predetermined location before executing the check-point. This can result in wasted energy, due to unnecessary backup executions when the energy is available. Additionally, because the check-points are in fixed locations, additional energy needs to be spent to roll-back the state after a restore.

1.3 Software Solutions

In this section, we give a brief overview of some software-level solutions that are meant to enable or optimize intermittent computing platforms. Software-level solutions and techniques have been explored in several works, and they generally do not require major modifications of the hardware or any exotic hardware feature. In particular, many software-level techniques are targeted for commercially available platform,s such as the *MSP430FE* series of micro-controller that feature a byte addressable non-volatile Ferroelectric RAM (FRAM).

These minimal hardware requirements makes the software-level solutions compelling for their low cost and because they might be easier to deploy. However, they do come with major drawbacks in terms of speed and consumed energy when compared with dedicated hardware solutions. This is especially true for run-time systems that need to carry out backup and restore task.

In this brief overview, we identify three major types of software-level techniques :

— dedicated and domain-specific programming languages,

— compiler and static techniques, and

— runtime/Operating-System level frameworks.

1.3.1 Programming Languages and Task-Based Approaches

An interesting approach proposed in the literature to enable non-volatile computing on energy harvesting devices, is to use dedicated or domain-specific programming languages or frameworks.

One of these dedicated framework is *Chain*, proposed in [CL16] as a programming language and run-time framework "for programming intermittent devices". Chain is a *task*- and *channel*-based programming language and framework, implemented in C, which guarantees to be intermittent safe at the task granularity [CL16]. In particular with Chain, the program needs to be divided into tasks by the programmer, which can perform both I/O and computations. These tasks communicate with other tasks, or with future instances of themselves, through channels which are implemented on the non-volatile memory. Each task can have their own private volatile variables, and have separate channels for input and output. However, tasks cannot directly access the NVM. Each task views its inputs as if they were immutable (*i.e.*, cannot be modified by the receiving task) constants and

always available in their channel. Moreover, every volatile variable must be task local. Because of this memory access model, each task is both atomic and idempotent, meaning that they can be interrupted and re-executed any number of times without changing the final result [CL16].

The main advantage of this approach is that it does not need check-points. In fact, once the inputs for the next task are written into the non-volatile channel, and the next task is scheduled to run, it is effectively like the program has reached a new check-point. However, this approach has some disadvantages. First of all, a user needs to learn a new programming language and framework and must adapt its application, converting it into a task based design. This means that existing code cannot be easily used with this approach without some important changes. Additionally, the framework only guarantees to be intermittent safe at the task level. This means that the programmer must guarantee that each task is able to run to completion, and must take care that the computation of each task is not too energy expensive and can fit within the energy budget. Otherwise, the whole application could be blocked by a single task that is always interrupted before completing. And correctly sizing each task could be difficult. Moreover, changes in the environmental conditions, such as increase in temperature, ageing and wear could reduce the effective capacitance and increase leakage [1], thus changing the effective energy storage.

Another programming language and run-time system for "perpetual systems" is *Eon*, presented in [Sor+07]. Contrary to Chain, Eon does not deal with the problem of backup/recovery in an intermittent computing scenario. While Chain addresses the problem of backup/recovery in intermittent systems, Eon is intended to be a *"coordination language"*, that is built to be *"energy aware"* [Sor+07]. In particular with Eon, the programmer can define how the system responds to external events depending on the energy state.

1.3.2 Static Techniques

Static Techniques are a category of software-level solution that exploit some static information, usually at compile time, to optimize the backup and restore operations. As an example, in [Zha+15], the authors propose a static analysis of a program that is able to determine an efficient position for the backups, such that the required size for the non-

1. TDK, *C2012X5R0G476M125AB : Detailed Information | Capacitors - Multilayer Ceramic Chip Capacitors*, en, URL : `https://product.tdk.com/en/search/capacitor/ceramic/mlcc/info?part_no=C2012X5R0G476M125AB` (visité le 09/07/2022).

volatile space is reduced. In particular, the technique proposed in [Zha+15] analyzes the program at a basic block granularity. The idea is that, instead of executing the backup as soon as the energy warning arrives, the backup can be deferred and executed n basic blocks later, when the size of the state (the stack frame) is smaller. This means that more instructions are executed, and the size of the resulting backup is reduced. To enable this, an offline analysis of the program is executed using the following information : the available energy when receiving the energy warning, the energy model for both instruction execution and backup, and the control flow graph (CFG) at the basic block level for the application [Zha+15]. Given these, for each basic block, the feasible set of backup position is computed, as well as the optimal position, that tells for each basic block *bb* to execute the backup after *B(bb)* basic blocks.

A similar approach is proposed also in [Zha+17], where the authors demonstrate a method to reduce the NVM size. The method is based on stack analysis techniques to determine the required size of the NVM, and on the identification of efficient backup positions. The backup positions are inserted in the code as *labels*. The idea is that instead of executing the backup immediately, at run-time when a power failure is signaled, the program continues the execution until it reaches the first *label*, and then it backups the volatile state and the stack. These labels are positioned in order to minimize the amount of data to store in the NVM in a static optimization task. However, the proposed technique also relies on a modified version of the Intel 8051 core, where a reserved instruction is used as a backup label in the code. This special processor is equipped with a dedicated backup-controller and non-volatile controller to execute the backup. This reliance on a dedicated processor, makes the backup process more optimized, at the expense of reducing the applicability of this solution.

Another category of static techniques is concerned with the estimation and evaluation of system energy and power requirements, that are useful in phase of development of a intermittently-powered device and its application. To this end, in [Che+17], a technique for determining application specific peak power and energy for an ultra-low power processor is presented. This technique can allow to better size the harvester and/or energy storage of the platform. The proposed technique uses the gate-level netlist of the processor to perform a symbolic simulation of the program. In this simulation, unknown logic values (Xs) are propagated through the gates as the application inputs. This simulation

allows to perform an "input-independent gate activity analysis", where the activity of gates is recorded, and gates that are not exercised can be identified [Che+17]. This allows to limit the peak power and energy consumption of the processor during the execution of the application. The author show that the application-specific peak power and energy bounds computed with the proposed approach, are normally much lower (27%, 26% and 15%) than those based on design specifications, stress-mark and profiling [Che+17].

One drawback of this approach, is that it requires access to the gate-level netlist of the processor, which is often not available. Although, as pointed out by the authors, the vendors of the processor could offer this evaluation as a cloud-based service, without releasing the netlist of the processor.

1.3.3 Run-Time Techniques

Software-based run-time techniques can be viewed like small OS-like software layers that manage the backup and restore functionalities.

Mementos

One of the first examples of such software strategies is presented in [RSF11]. In the paper, the authors present *Mementos*, a software system that provides general-purpose programs with support for running in intermittent computing scenarios, by providing protection from power losses [RSF11]. *Mementos* comprises both a run-time system library and a set of compile-time techniques for optimizing the placement of energy checks in the program [RSF11]. The compile-time optimization wraps the program with the code for restoring the state from previous checkpoints, as well as instruments the code of the program, inserting calls to a Mementos library function, which estimates the available energy.

Mementos provides three types of instrumentation for placing the trigger points :

— *loop-latch mode*,
— *function return mode*, and
— *timer aided mode*.

The first two strategies, *loop-latch* and *function return*, place trigger points at the loop back-edges and after call instructions respectively. The third strategy, *timer aided mode*, uses a timer interrupt that is added either at function returns or at loop latches, to raise a flag. Each trigger point only checks the flag, and proceeds with the the energy check

only if the flag is raised, after each energy check the flag is lowered again.

The run-time energy estimation is done by measuring the voltage level at the capacitor with the analog-to-digital converter (ADC), (which needs to be available in the target platform). If the measured voltage is above the *checkpoint threshold voltage* (V_{thresh}), then no snapshot is performed. Otherwise, Mementos assumes that a failure is imminent and begins check-pointing the state. The check-pointed state includes the registers, the stack, whose depth is computed using the stack pointer, and the global variables, which are captured at compile time.

Mementos also takes care to protect against check-point corruption due to power losses during a check-pointing operation. In particular, the first checkpoint word it writes is an header used to detect incomplete checkpoints and reconstruct complete ones, while the last word is a magic number that ends every valid checkpoint. Thus incomplete checkpoints are detected, and can be erased after boot when the energy is available [RSF11].

While this system allows application to run under intermittent power, it does so by introducing significant overhead. Because it uses statically placed check-points, unnecessary and expensive checks could be executed. Moreover, unnecessary state check-points can also be taken if the threshold is not set properly, which incurs in an even greater cost in terms of energy and execution time. Moreover, the system does not take a snapshot until a checkpoint is reached. This means that power failures can happen relatively far from the previous checkpoint, resulting in a higher restore and roll-back cost. In the paper, the authors report an overhead ranging between 65.4% and 360% of the CPU cycles with respect to the uninstrumented program.

Hibernus and Hibernus++

In response to some of the shortcomings of Mementos and other periodic/static check-pointing systems, Balsamo *et al.* present *Hibernus*, a software-based approach to sustain computation under intermittent power supply [Bal+15]. Instead of placing checkpoints at compile time, *Hibernus* saves a snapshot only once immediately before a power failure, and then goes to sleep. The idea is to remove unnecessary checks and thus to minimize the number of backup operations, and also to minimize or remove the overhead of rollbacks. As shown in Figure 1.3, *Hibernus* compares the supply voltage (V_{CC}) with a given hibernate threshold (V_H) voltage. If the supply voltage is lower than the threshold ($V_{CC} \leq V_H$), an interrupt is generated, and the system moves from the *active* to the *hibernating* state. In the *hibernating* state, the system immediately saves a snapshot of the full system memory

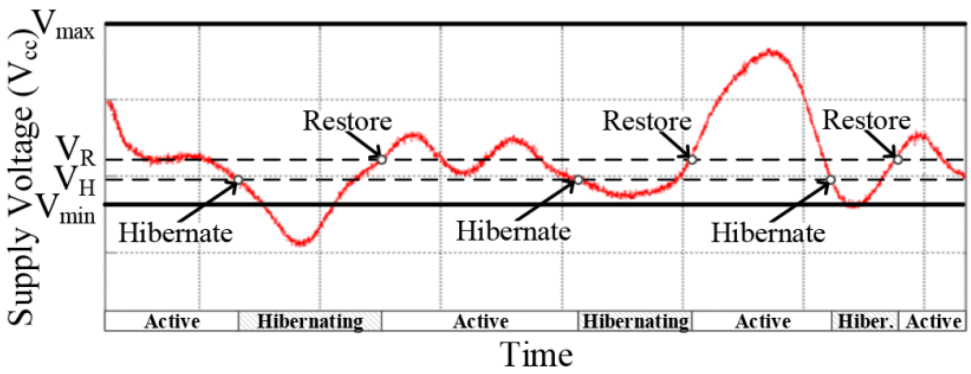


FIGURE 1.3 – Example of Hibernus execution, from [Bal+15]

and of the registers, and then goes into deep sleep. Another interrupt is used to trigger the restore operation, when the supply voltage raises above the restore threshold V_R [Bal+15].

The main advantage of Hibernus's approach over Mementos is that the backup is executed on-demand, only when a power failure is deemed to occur. This minimizes run-time overhead, as no checks are executed. However, Hibernus does require full memory backup, as no compile time optimizations are applied. This means that the backup and restore threshold must be selected pessimistically, to have enough energy for a full memory backup. Another issue with Hibernus is that the backup and restore thresholds are statically selected based on an off-line characterization of the platform [Bal+16].

To address this issue, Balsamo *et al.* propose *Hibernus++*, an evolution of Hibernus that dynamically adapts the hibernate and restore thresholds based on the system power consumption, the size of the capacitor, and the behaviour of the harvested energy [Bal+16]. When power is harvested, Hibernus++ checks if the system is calibrated, and if not it runs the calibration to set the threshold voltages. Then, the supply is checked to ensure that the system can run. If not, the system sleeps until the energy is sufficient. When the supply test is successful, the system checks if there is a snapshot to be restored. Hibernus++ also detects if the previous snapshot failed and, if so, it increases V_H. If there is a valid snapshot, Hibernus++ restores the system state and resumes operation, otherwise the computation is started from the beginning. The calibration routine is therefore an iterative process where the platform waits to reach a calibration voltage V_{cal}, and then disconnect the supply and takes a snapshot. After the snapshot, the voltage level is measured and used to compute the hibernation voltage V_H. If the calibration attempt fails, V_{cal} is increased and the calibration retried [Bal+16].

Hibernus++ also tunes the restore strategy based on the characteristics of the harves-

ted energy source. The energy source is characterized as *High-Power* if the power supplied by the harvester is sufficient to drive the platform. Otherwise, the energy harvester is characterized as *Low-Power* [Bal+16]. For *High-Power* sources, the restore can start as soon as the voltage raises above V_{min}. On the other hand, for *Low-power* sources, the voltage needs to reach a higher value. Otherwise, the platform pulling more power than the source can immediately drop the voltage, triggering the hibernation procedure and not allowing any progress to be made.

Quick-Recall

In [Jay+15], Jayakumar *et al.* present *Quick-Recall*, an hardware-software approach for transiently-powered computing. The approach relies on the use of FeRAM, as a superior alternative to Flash memory. Quick-Recall defines the system context to be preserved across power failures, as the program state, the processor state, and the peripheral state [Jay+15]. Quick-Recall uses the FeRAM as its main memory, thus there is no overhead for retaining the program state. The processor state includes all the microprocessor ISA registers, such as the program-counter (PC), stack-pointer (SP), status-registers (SR) and the general purpose registers (GPRs), Quick-Recall saves the values of all this registers to FeRAM in case of a checkpoint. The problem of retaining the peripheral state is delegated to the programmer, as Quick-Recall will use a programmer-defined initialization routine during the restore phase [Jay+15].

The main advantage of Quick-Recall is that it allows to easily adapt an existing program to run in a transiently powered scenario. Moreover, because the system uses FeRAM as its main memory, only the processor registers need to be saved, which means that the overhead of check-points and restores is much reduced. However, the drawback of this approach is that FeRAM requires more energy during the active state and it is slower, with respect to SRAM. This results in slower operation frequency, and the additional power consumption can drain the capacitor faster, resulting in an increased power failure rate. Another drawback of this approach is that is partially susceptible to a broken time machine problem [RL14], as an incomplete check-point would leave the NVM in an inconsistent state. To avoid this problem, QuickRecal writes a flag after each check-point, which is checked and then cleared after the restore. This means that, if a check-point is interrupted, the flag is never set and, at the next restore, the system will restart the application from the beginning, loosing all the progress.

eM-map

In [Jay+17], the authors start from the observation that approaches that rely only on NVMs as unified memory for transiently-powered systems (like Quick-Recall [Jay+15]), achieve reliability at the price of energy and performance inefficiencies when compared to SRAM-based systems. On the other hand, SRAM does offer better latency and energy efficiency but is exposed against power losses [Jay+17]. To address these issues, the authors propose *eM-map*, a energy-aware memory mapping technique, for hybrid FeRAM and SRAM systems [Jay+17]. *eM-map* treats *Functions* as the basic units, that can have their *text*, *data* and *stack* sections mapped to either FeRAM or SRAM, while, for simplicity reasons, the *heap* is unique and mapped exclusively to FeRAM. The best memory mapping for each function, is the one that can complete in one power cycle these three steps : 1) *migration*, sections mapped to SRAM are moved from FRAM to SRAM, 2)*execution* of the function, and 3)*check-point* (moving back data from SRAM to FeRAM), with minimum energy [Jay+17].

At the beginning, all functions are configured to have the three sections mapped on FeRAM (FFF), for which no migration and almost no check-pointing are necessary. To determine the best mapping for each function, a one-time characterization of the functions is run. The test picks a configuration (*e.g.*, FFS, meaning *text* and *data* in FeRAM, *stack* in SRAM), measures the initial voltage, performs the three steps (*migration*, *execution* and *checkpoint*), and then measures the final voltage and energy consumption for the current configuration, updating if necessary the preferred configuration [Jay+17]. To further improve the efficiency of the system, the authors also propose an "Energy-Align" function, that ensures that the three steps (for mappings different than FFF) are executed only if enough energy is available, otherwise shutting-down the platform.

While *eM-map* provides an optimization of the backup, by finding the best way to map a function in a hybrid NVM-SRAM system, it does so by relying on the assumption that the execution flow of the device is deterministic. As an example, a device might always collect the same amount of data, and performs computation on this data that takes a deterministic amount of clock cycles [Jay+17]. This assumption is necessary to ensure a correct evaluation of *migration execution* and *check-pointing* energy cost. This means that this approach is not easily applicable to more general computing task, that might have some non-deterministic or data dependent behaviour. Moreover, the guarantees of consistency against *"broken time machine"* errors [RL14], are based on the assumption that there is enough energy to complete the function atomically, this is explicitly checked

by the energy-align function. However, if the estimation of the energy consumption for a function is wrong or, for some reason, like aging or other environmental conditions, the system has less energy than what is estimated, the function could still be interrupted, which could result in an inconsistent state.

1.3.4 Sytare

In [Ber+19], Sytare, a lightweight kernel for managing check-pointing and peripheral state on transiently powered systems is presented. While many run-time techniques focus manly on check-pointing the content of main-memory, with Sytare, the authors, also address the issues of persisting the peripheral state in a transiently powered systems. The authors identify three main aspects in dealing with peripheral state persistence : *peripheral state volatility*, *peripheral access atomicity*, and *interrupt handling*. The peripheral state is assumed to be entirely volatile, as complex peripherals, such as a radio chip, would require a new configuration after a power failure. The access to the peripheral should be atomic, which means that if a power loss interrupts the hardware access (such as a radio transmission), the process must be restarted from scratch after the reboot. Finally, the interrupts could modify the data memory or the state of peripherals, causing consistency errors.

To address these issues the authors propose a separation between the application code and the driver code controlling the hardware, with the addition of a thin kernel code layer (Sytare) that works as the interface between the application code and the drivers. This kernel code manages the initialization, check-pointing and restore of the platform. Moreover the driver calls are made accessible through system-calls that wrap the driver code, dealing with guaranteeing the peripheral access atomicity as well as the initialization, save and restore of the peripheral context. Because the system call needs to be repeated in case of a power failure, the system call stack is not saved, instead the system call address, its arguments and a flag that signals that a system call was in progress are saved. This allows the system to replay the system call on reboot, after the state of the platform and the context of the drivers has been restored.

This approach makes it possible to adapt an existing application, to run in a transiently powered scenario with relatively low modifications. However, the system still requires some effort on the part of both the application programmer, as well as the driver developer which needs to write the functions for the actual low level access of the peripheral devices.

For the check-pointing of the application state, Sytare uses a double buffering backup scheme, to be able to restore in case of a power failure during the backup. Similarly to other run-time solutions, the proposed check-pointing method saves the whole volatile state of the application. This makes the backup process slower and more expensive in terms of energy consumption. An improvement on the backup process is proposed in [Ber+20], where the memory protection unit is used to implement a differential backup, so that only the regions of memory that have changed are saved during a check-point.

1.4 Architectural (Hardware) Solutions

In recent years, many ways of achieving non-volatility for intermittent platforms have been explored in both industry and academia. Many of the proposed solutions are hardware-based, involving the integration of NVMs in the architecture, which are detailed in this section. Among the hardware-based solutions, many works have presented the implementation of fully non-volatile processor, by exploiting emerging NVMs and implementing non-volatile flip-flops based on different types of NVM devices. Later, in Section 1.5, a brief description of some fully non-volatile processor solutions presented in different papers is given.

1.4.1 NVM Integration

The integration of NVMs in the different levels of the memory hierarchy has been explored by many works. NVMs can be integrated both to achieve non-volatility, as well as for other advantages, such as the reduction of leakage energy and the increased density. These characteristics can bring improvements even in domains other than intermittent computing. As an example, both in [Sen+16a] and in [Xue+11], the integration of STT-RAM in the last level cache is explored.

In [Xue+11], an introduction to three NVM technologies : PCM, STT-RAM and RRAM is given. The three technologies are presented with some possible applications especially for PCM and STT-RAM. In particular, PCM is presented both as a possible replacement for Flash, as well as a memory to replace or complement DRAM as the system memory. Figure 1.4 from [Xue+11] shows two possible organizations for a hybrid PCM/DRAM main memory system. The first possibility illustrated by Figure 1.4(a) is

to have a PCM main memory with a DRAM cache in front, to help mitigate the write latency and extend the lifetime of the PCM. This configuration is reported to achieve 3×

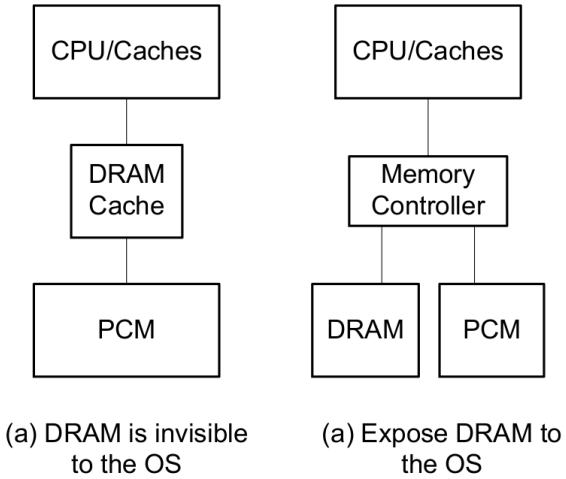


FIGURE 1.4 – Hybrid DRAM/PCM main memory organization from [Xue+11].

speedup and 3× increase of lifetime, by having a DRAM cache with a capacity of 3% of the PCM storage, while introducing a limited 13% area overhead [Xue+11]. The other organization shown in Figure 1.4(b), exposes both the DRAM and PCM to the operating system (OS). This allows various OS level optimizations to be implemented, to both increase performance and extend the life time of the PCM [Xue+11]. While this approach can be interesting for other domains (*e.g.*, in high-performance computing), it can be difficult to apply in the embedded systems domain, and especially to intermittently-powered systems, which are usually very constrained in terms of power, area and cost. The paper [Xue+11] also presents STT-RAM as a possible candidate for implementing L2/L3 caches taking advantage of 3D integration. In particular, different strategies are analyzed for the implementation of L2/L3 caches with STT-RAM considering many cases such as a full SRAM to STT-MRAM replacement for both L2 and L3 caches, SRAM for the L2 and STT-RAM for the L3, with the possibility of moving the error correction code (ECC) bits of the L2 cache into the STT-RAM based L3 cache.

The use of STT-RAM for the last levels of cache can introduce several advantages. As STT-RAM is much more dense than SRAM, a much larger L2 and L3 cache can be included w.r.t. SRAM with the same area. The presence of an SRAM based L1 can partly hide the speed and energy penalty introduced due to STT-RAM write operations. Moreover, because STT is non-volatile, having an STT-based L3 cache eliminates the

high static power consumption with respect to a standard SRAM-based cache. The main drawbacks of this approaches is cost, as it requires complex packaging (or manufacturing) techniques. Moreover, while some applications can benefit from a larger cache, others can see a performance degradation due to the STT being slower than a traditional SRAM cache.

In [Sen+16a], the impact of the integration of MRAM in the hierarchy is analyzed. In particular, the paper focuses on the L2 cache, though L1 cache is also analyzed. The analysis uses the gem5 processor simulator together with the circuit-level characteristics of the MRAM. Then, the output of gem5 simulation is extracted and the energy consumption of the memory hierarchy is computed. The analysis concluded that replacing SRAM with MRAM in the L2 cache can bring significant advantages, such as reducing the energy consumption thanks to the lower leakage of MRAM. Moreover, because the considered MRAM has a faster read time than the compared SRAM (both in 45nm), there is no more a performance penalty due to the longer write times of MRAM. However, the paper also reports that MRAM is not yet a good substitute for SRAM in the L1 cache, due to its current characteristics, in particular the high write latency.

1.4.2 Brainshift Architecture

In [Hag+17], *Brainshift* is presented, a method that exploits the *scan-chains* for the backup and restore operations. Contrary to approaches such as non-volatile flip-flops used in many NVPs [Bar+13 ; Sak+14 ; Liu+16], the proposed method tries to avoid introducing modifications and additional hardware to the processor architecture. Instead this work relies on the existing *scan-chain* circuits, normally used in digital designs for testing purposes, to perform the backup and restore of the internal state of the processor. The micro-controller is divided into two domains : the non-volatile domain (NVD) and the volatile domain (VD), this two domains defines the registers that will be saved and those that will not. Additionally, some non relevant registers inside the NVD do not need to be saved. As an example, the pipeline registers can be excluded from the backup operation. Before executing the backup the processor is put to sleep mode, this ensures that all transactions in the bus are completed. Then, the relevant state in the NVD can be shifted into an *Out-of-Place Memory* (OoPMem).

Another major difference with other NVPs is that the OoPMem being used in this work, is not a proper non-volatile memory. In fact, this paper proposes the use of a ultra-

low-power SRAM memory, capable of low-power retention, combined with the use of an auxiliary retention battery. According to the reported numbers, this type of memory, when paired with a small $50\mu Ah$ battery, is capable of retaining 10kbits for over 2 years. The paper also presents the analysis of two possible approaches for the backup and restore operations : *In-Place* and *Out-of-Place* backup. In particular, the total energy consumed during one power cycle E_{cyc} is considered, and is defined as

$$E_{cyc} = E_{S+R} + t_{cyc}((P_{on} + P_{oh})D + P_{ret}(1 - D)) \tag{1.1}$$

where E_{S+R} is the energy for saving and restoring the state, t_{cyc} the total time for a power cycle, D the duty cycle, or the fraction of t_{cyc} spent in the *ON* state, P_{on} the power consumed during the on state, P_{oh} the power overhead due to the adopted strategy, and P_{ret} the power necessary to retain the backup information during the *OFF* time, which can be zero for strategies using non-volatile memories. Figure 1.5 compares the total energy

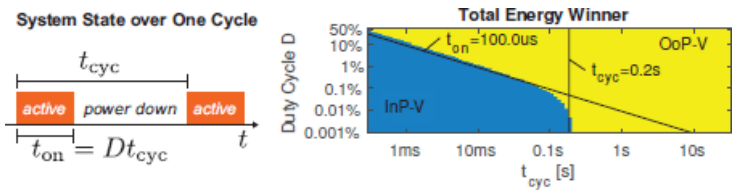


FIGURE 1.5 – In-Place vs Out-of-Place energy consumption with varying t_{cyc}.

of *In-Place* and *Out-of-Place* strategies, given the parameters reported in Table 1.2. As

TABLE 1.2 – Parameters for In-Place vs Out-of-Place comparison

	In-Place	Out-of-Place
E_{S+R}	0 pJ/b	6 pJ/b
P_{ret}	1 pW/b	30 pW/b
P_{oh}	2%	0%

shown in Figure 1.5, the *In-Place* approach provides an advantage when the interruptions are more frequent, as usually these methods have lower backup and restore energy cost, as well as faster sleep and wake up times. On the other hand, the *Out-of-Place* approach is more convenient when the cycle times are longer, thus allowing for longer execution periods, where the In-Place approach would suffer from the higher power overhead.

The proposed approach is a light-weight method for retaining the system state, while avoiding major modifications to the SoC architecture. Additionally, the energy consump-

tion analysis shows that this type of solution should be advantageous for systems with on-time longer than $100\mu s$ and interruptions happening less frequently than 5Hz. However, this solution does not offer true non-volatility, as the system requires low-power retention and the use of a battery. The proposed method still has a relatively large footprint and introduces maintenance issues.

1.5 Non-Volatile Processors

This section presents a brief overview of some selected works in the field of non-volatile processors (NVP) for transiently-powered systems. Most of the papers summarized in this section present fully non-volatile processors that have been silicon-proven and are based on emerging NVM technologies.

1.5.1 MSP430F : an MCU with Embedded FeRAM

In [Zwe+11] (also in the associated commercial product MSP430F from Texas Instruments [Ins21]), an ultra-low-power micro-controller, aimed at energy harvesting systems is presented. The processor is a 16 bit, 3-stage pipelined core, implementing the MSP430 instruction-set architecture. The proposed SoC has three types of memories : a 1 KB SRAM memory, 4 KB of ROM and 16 KB of FeRAM. The ferroelectric memory results in a great improvement when compared to Flash. In terms of speed the FeRAM offers 1000× faster writes, while the endurance is improved to more than 10^{14} cycles, versus the typical 10^5 cycles of Flash. Additionally, the proposed FeRAM provides a 100× improvement in power with respect to the typical Flash-based micro-controllers.

The SoC is capable of running at a maximum frequency of 25 MHz. However, as the access and cycle time of the FeRAM are 55 ns and 110 ns respectively, the maximum frequency the FeRAM is able to operate is 8 MHz. To increase the effective access rate, the FeRAM is coupled with a 2-way-2-line read cache. This SoC also has numerous low-power features, such as three independent voltage domains, one for the digital logic, one for the real time clock (RTC) and one for the FeRAM. The CPU supports five low-power modes, while a Power Management Module controls the three voltage domains. As this SoC is targeted at intermittently-powered energy harvesting applications, the FeRAM is equipped with a dedicated on-chip power supply system, using a $2nF$ capacitor to buffer the supply voltage. Moreover, the input voltage is monitored and, in case of a failure, it

is disconnected from the FeRAM which can complete the pending memory access thanks to the buffer capacitor. Figure 1.6 represents the die photograph and Figure 1.7 the block diagram of the SoC presented in [Zwe+11].

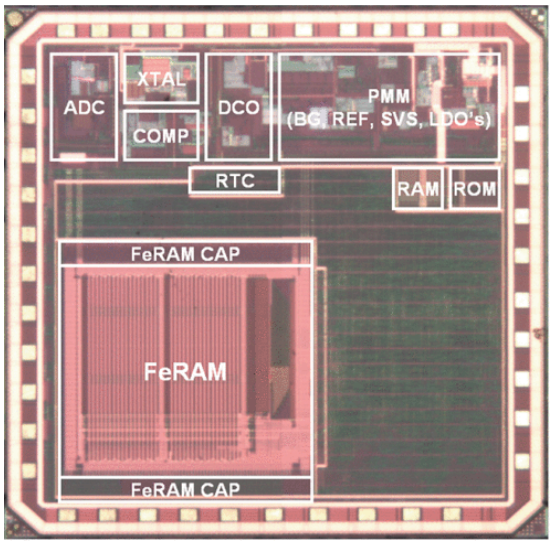


FIGURE 1.6 – Micrograph of the MCU from [Zwe+11].

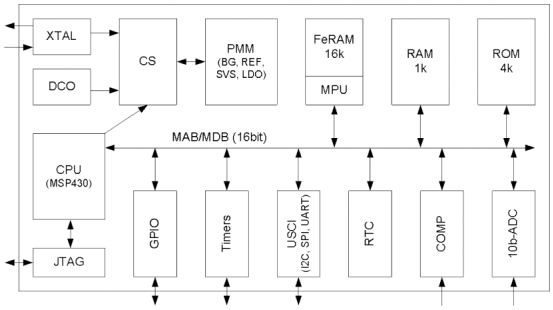


FIGURE 1.7 – MCU block diagram from [Zwe+11].

This paper by Texas Instruments in 2011 presents an SoC platform, which is now commercially available [Ins21], and has become a common choice for those works that use software-level solutions to enable intermittent computing under environmental energy harvesting, such as [Bal+15 ; Bal+16 ; Jay+15]. The addressable FeRAM represents a great improvement with respect to Flash memories, and enables software-level strategies for intermittent computing. However, this type of platform also shows some shortcomings, as the access time and energy consumption is still far behind that of SRAM, which remains the only choice to achieve fast low-power operations during execution. Moreover, with the

exception of the 2nF capacitor for completing a pending memory operation, the SoC has no actual hardware support to allow for backup and restore operations under intermittent power supply.

1.5.2 THU1010N : a Non-Volatile-Processor based on FeFFs

In [Wan+12], a Non-Volatile-Processor (NVP), named as *THU1010N*, designed for intermittently-powered computing, is presented. The processor features a simple 8-bit micro-controller unit based on the 8051 ISA, running at a maximum frequency of $25MHz$. All the registers of the processor core, including a 128-byte register file, are implemented using non-volatile ferroelectric devices called FeFF. These devices, shown in Figure 1.8, are comprised of a standard volatile two stage flip-flop, enhanced with a non-volatile storage based on two ferroelectric capacitors. The components of the SoC where FeFF are used are

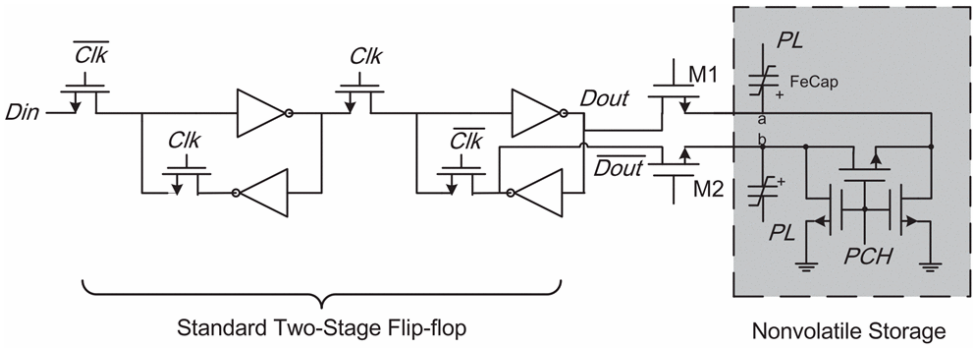


FIGURE 1.8 – Proposed ferroelectric flip-flop [Wan+12].

part of the Non-Volatile Logic domain. The control signal for the FeFF in the non-volatile domain are generated by a dedicated flip-flop controller, in response to a *sleep/wake-up* signal. The rest of the system, in the volatile domain, comprises some communication peripherals (SPI, I2C, JTAG), and an 8-KB SRAM used as system memory.

The sleep/wake-up signal used by the flip-flop controller to trigger backup and restore operations, comes from the *Configurable Voltage Detection System* (CVDS). The CVDS, depicted in Figure 1.9, contains two configurable switched capacitor arrays, one of which is attached to the power line and is used to provide energy for the backup operation. The other one is used in the voltage detection circuit, to determine the delay after which the sleep/wake-up signal is generated, after the supply voltage, V_{dd}, passes the threshold voltage.

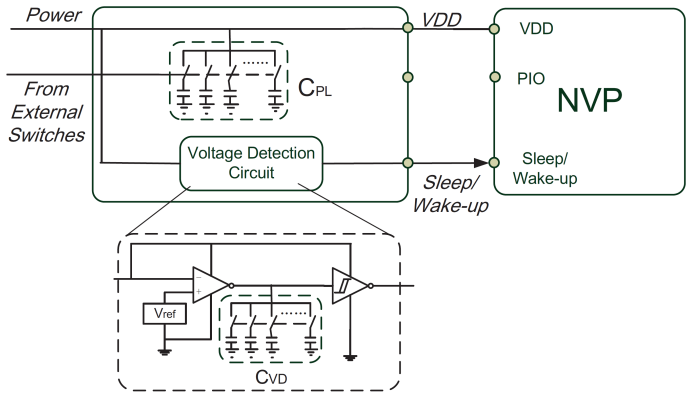


FIGURE 1.9 – Configurable Voltage Detection System, from [Wan+12]

Thanks to the hybrid non-volatile flip-flop, the THU1010N processor achieves a large improvement in sleep and wake-up time. With a $7\mu s$-sleep time, the THU1010N exhibits a speedup of $100 - 1000\times$ with respect to popular *MSP430* processors based on Flash or FeRAM [Ins21 ; Zwe+11]. In the mean time, the achieved wake-up time is $3\mu s$ at $1.5V$, which represents a speedup of $30 - 100\times$ w.r.t. existing industry processor [Zwe+11]. Thanks to the distributed FeFF architecture, the NVP achieves huge energy savings to backup and recall its 1607 registers, compared with off-chip Flash memory and with on-chip Flash.

This approach however shows severe limitations. The FeFF are larger than regular flip-flops, leading to a large overhead in terms of area for the whole chip [Bar+13]. Additionally, the THU1010N processor does not offer any non-volatile storage for the content of the main memory, thus the save and recall operations are limited to the internal registers. This means that, to backup the application memory, an external off-chip NVM is required, and the data would need to be saved in a serial fashion, jeopardizing the sleep and wake-up time and energy advantage obtained with the FeFF. Another possible problem that could incur in with the distributed FeFF approach, is that it does not offer any guarantee of consistency in the event of data corruption during the save operation.

1.5.3 Non-Volatile logic Cortex-M0 with distributed FeRAM mini arrays

Many NVPs exploit some form of non-volatile flip-flops (nvFFs) to retain the state of the internal registers and perform fast parallel backup operations. The advantage of nvFFs

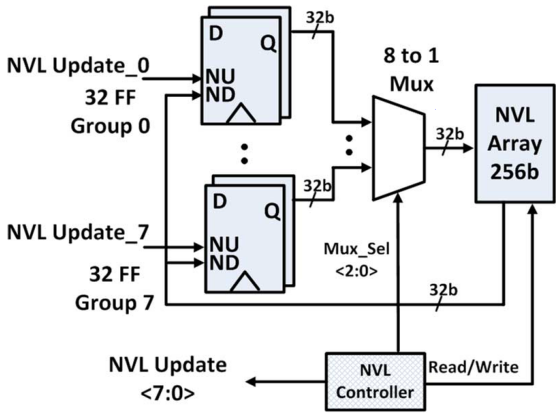


FIGURE 1.10 – FRAM mini array connected to flip-flop groups, from [Kha+14].

is the very fast backup and recovery time. This however comes at a cost in silicon area, as the nvFFs are normally much larger than the standard volatile flip-flops. Additionally, there is a limit on the maximum number of bits that can be saved or restored at the same time, given by the maximum affordable peak current, thus limiting the scalability of the distributed nvFF approach. To address the area overhead and to limit the peak current during the backup and recall operations, in [Bar+13 ; Kha+14], a non-volatile processor using distributed mini arrays of non-volatile cells is presented. The proposed processor is a low-power Cortex-M0 core, running at 8 MHz, with an average current consumption of $75\mu A/MHz$. The SoC has three types of memories : a 10-KB ROM, an 8-KB SRAM and a 64-KB FeRAM memory.

To preserve the content of the internal registers, 10 distributed mini arrays of 256 bits are used. The mini arrays are based on ferroelectric capacitors cells. Each mini array is organized into eight rows of 32 bits, thus serving eight groups of 32 flip-flops, as shown in Figure 1.10. During the backup, the rows are written sequentially one after the other, a controller selects which group of FFs to write and generates the write signals for each mini array. The use of distributed mini arrays avoids the high area and performance overhead of single non-volatile flip-flops. This also avoids the increased fabrication difficulties and reliability problems that comes with embedding non-volatile elements at the flip-flop level. Moreover, the mini arrays limits the problems of slow wake-up speed, excessive routing and power cost of a single large centralized array of NVM. This solution strikes a balance to allow relatively fast backup and restore for the internal state of the processor, while avoiding the high cost of nvFFs. The proposed SoC also comes with FeRAM allowing the use of the platform as a fully non-volatile processing system. This however will come

with the limitations due to the slower access times and higher energy consumption of FeRAM with respect to SRAM. Another problem that might limit this SoC as a non-volatile platform, like other fully non-volatile processors based on nvFFs, is that the mini arrays do not offer any protection for the backup, thus a single bit error in the mini arrays could compromise the internal state of the processor. In particular, the backup could be interrupted while the FF are still being copied to the mini arrays, resulting in an inconsistent state. A simple mitigation to protect the consistency of the backup would be to introduce some form of error correction or, more in general, some form of redundancy.

1.5.4 A 90nm 20MHz nvMCU based on STT-RAM

In [Sak+14], an NVP fabricated on a $90nm$ node is presented. The SoC features a 16-bit RISC processor based on the MSP430 instruction set running at $20MHz$. The main memory is a 64 KB unified RAM/ROM based on a SpinRAM (STT-RAM) macro designed around a *2-transistors 1-magnetic tunnel junction* (2T1MTJ) cell. The memory macro also includes redundant words and columns as well as error check and correction (ECC) circuit for write failures. The SoC also features a power management module (PMM), unified clock system and a number of peripherals, including timers, a 32-bit multiplier, DMA, 12-bit ADC, serial interfaces and 8 input output ports.

In addition to the main memory being fully non-volatile, the SoC exploits non-volatile magnetic flip-flops (MFFs), to quickly save and restore the 4072 registers, eliminating the need for a long sequential backup. The backup/restore of the CPU internal state can be triggered by two special instructions called *SAVE* and *LOAD*, allowing software control of the backup and restore process. The CPU supports two modes for the backup, one which is called *logging*, where each register access triggers the backup even if no *SAVE* instruction is fetched. The other one is the *software-controlled* mode, where the CPU activates the backup-enable signal only when a *SAVE* instruction is fetched. This last mode significantly reduces the backup energy. Moreover, to avoid unnecessary backups, each register is coupled with a dirty bit, to mark that the content of the register differs from the data saved in the magnetic element of the MFF. This *"scoreboarding"* for the internal registers can lead to significant reduction in backup energy, saving up to 50% of the energy, when only 10% of the peripheral registers are toggled. In addition to the non-volatile features, the proposed SoC also provides a number of low-power features, such as 11 isolated power domains as shown in Figure 1.11. Moreover the SoC supports 3 low-power modes : *stand-by*, *power-gating* and *sleep* mode.

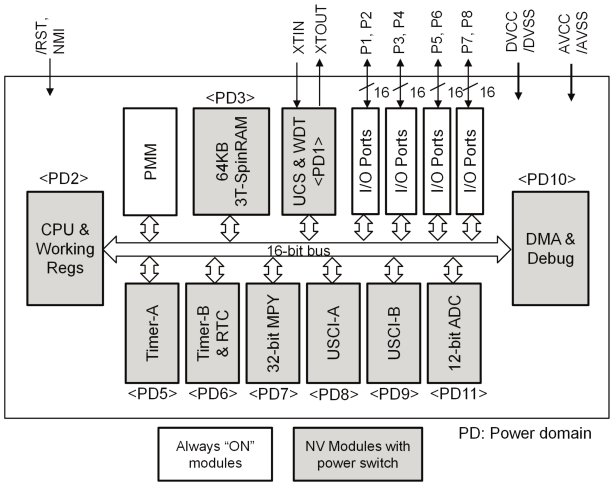


FIGURE 1.11 – Block diagram of the proposed SoC with the different power domains (PD) [Sak+14].

The proposed design gives an example of a fully non-volatile processor, where the main memory is fully non-volatile and the internal registers are made non-volatile by using hybrid MFFs instead of regular FFs. The use of STT-RAM memory also provides a speed-up when compared with similar NVPs based on other technologies such as FeRAM [Zwe+11 ; Bar+13], allowing the processor to run up to $20MHz$. However, this NVP shows similar limitations to other fully non-volatile processors, with the increase in area, performance and energy overhead due to hybrid non-volatile flip-flops. Moreover, while the addition of dirty bits avoids unnecessary backups, it also introduces a significant cost overhead as the dirty bit tracking is done at the register level. Additionally, even though STT-RAM represents an improvement over other types of NVM, it stills comes with similar limitations when it comes to access time, energy, and endurance with respect to SRAM.

1.5.5 A 65 nm ReRAM-Enabled NVP

In [Liu+16], a ReRAM-enabled NVP is presented. The chip features a 8051 compatible core, capable of running at frequencies higher than $100MHz$, as well as peripherals such as a timer, UART and GPIOs. In order to obtain the non-volatility of the processor state, the internal registers are hybrid non-volatile flip-flops (nvFFs). The memory subsystem of the platform features a 4-KB non-volatile SRAM (nvSRAM) and a 8-KB ReRAM used for code. The main nvSRAM memory is based on a hybrid design featuring an enhanced 7T-

1R SRAM cell, with a ReRAM based storage elements [Lee+15]. The nvSRAM has three modes of operation : *SRAM* where it works as a normal SRAM, with similar performance to a 6T-SRAM, *STORE* and *RESTORE*. In the *STORE* mode, the volatile data content of the nvSRAM cell are stored to the respective resistive element. Conversely, in the *RESTORE* mode, the content of the resistive elements is read and restored in the volatile cell.

The restore parallelism of the nvSRAM, can be configured, changing the number of lines (16B) that are restored in parallel, and can be selected to be 1, 4, or 16 lines in parallel. This allows to trade-off the peak current with the restore speed. Additionally, the store and restore energy and time can be further reduced by configuring the size of the nvSRAM (0/16B/256B/1KB/4KB). Moreover, both the nvFFs and the nvSRAM exploit self-write-termination (SWT) circuits, that avoid writing to the resistive element data that is already equal to the value already present in the ReRAM device. The paper reports a ratio of matched cells for their embedded benchmarks of 90%, allowing the self-write-termination to reduce the store energy by up to 172×. Finally, the nvSRAM can also be put in *RETENTION* mode, which can save energy by avoiding a backup, when the power interruptions are short. In this *RETENTION* mode, most power domain are gated, the voltage is lowered to $0.4V$ and the clock is gated, greatly reducing leakage.

The non-volatile flip-flops and nvSRAM are controlled by a *Non-Volatile Controller* (NVC), that can trigger the *STORE* and *RETENTION* modes. A 2-bit predictor and a time-out mechanism are used to decide when to switch between the *STORE* and *RETENTION* modes.

This work presents several architectural- and circuit-level techniques to realize a non-volatile processor capable of restore times between $20ns$ and $170ns$. Thanks to the faster hybrid nvSRAM, this NVP can operate at 100MHz, which is faster than other NVPs running only on NVM, such as [Sak+14 ; Bar+13]. Additionally, techniques such as adaptive nvSRAM sizing, self-write-termination and time domain retention provide great improvements in backup energy. The use of configurable nvSRAM size also benefits restore energy and time, while adaptive restore parallelism can trade off peak current with restore time.

This work brings several improvements with respect to previous NVPs, especially when it comes to execution speed, mainly thanks to the faster access times of nvSRAM. However, this NVP still suffers from several drawbacks as other NVPs, such as the increased area overhead due to hybrid flip-flops. Additionally, the endurance of the ReRAM devices, especially in the nvFFs, might be problematic, even if the problem is mitigated by the

use of self-write-termination circuits.

1.5.6 RRAM Non-Volatile Intelligent Processor

In [Su+17a], a non-volatile processor platform, based on a 8051 CPU, is presented. The processor is fabricated on a 150 nm CMOS process, with the addition of *HfO* based RRAM. The RRAM elements are used to implement the 4-KB nvSRAM, and for the non-volatile flip-flops, each of which includes two resistive elements (2R-nvFFs). Moreover, the RRAM is also used in the embedded processing-in-memory unit for fully connected neural networks (FCNNs). The FCNN unit is composed of four RRAM arrays using 1T-1R (one transistor, one resistor) cells. Additionally, the processor features a power management unit (PMU) as well as other peripherals such as I2C, SPI, GPIO, UART. The PMU handles the backup and restore decisions for the chip, as well as providing V_{dd}. Fast backup and restore operations are enabled by the 2189 non-volatile flip-flops, all of which embeds two bipolar RRAM elements, with data aware self-write-termination (SWT). The nvFFs have three modes of operation : normal, backup and restore. The PMU generates a backup signal before the power is interrupted, putting the nvFFs into backup mode, where the data contained in the flip-flop is written in the RRAM devices. When the data in the flip-flop match the content of the RRAM elements, the SWT terminates the writing operation, saving energy and improving the endurance of the RRAM devices. This also avoids a write operations when the content of the flip-flop already matches the one of the RRAM devices. On the other hand, during restore mode, the resistance state of the RRAM is read and its state is transferred to the volatile part of the flip-flop.

The FCNN, shown in Figure 1.12, enables low-power processing-in-memory. The unit is composed of four 1T-1R RRAM arrays that are used to perform matrix vector multiplication operations, where the weights are stored as resistance values in the RRAM arrays. To improve the efficiency and reduce the current required to drive all the word-lines, compared to other RRAM based matrix vector multiplications, the FCNN is designed with binary interfaces. In particular, the inputs are directly connected to the word-lines removing the digital-to-analog conversion (DAC) circuit. The outputs are obtained through the dedicated 1-to-3-bit sense amplifiers at the end of the bit-lines, also removing the conventional ADC circuits [Su+17a].

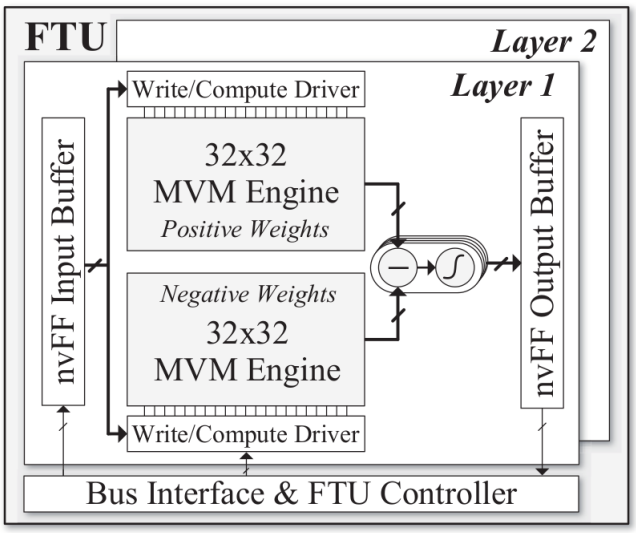


FIGURE 1.12 – FCNN array proposed in [Su+17a]

1.5.7 FeRAM Parallel Recovery SoC

In [Wan+17], Wang *et al.* propose a non-volatile system on chip (SoC) targeted at normally-off applications. The proposed SoC uses FeRAM-based memories and non-volatile flip-flops to achieve non-volatility and to accelerate sleep and wake-up[Wan+17]. The SoC features a 16MHz micro-controller, with a 32-KB FeRAM instruction memory and a 32-KB FeRAM data memory. Moreover, parallel backup and recovery is supported thanks to 1910 non-volatile flip-flops [Wan+17].

Conventional power-on detection systems use comparator circuits to compare V_{dd} with a given sleep threshold V_{sleep}, and requires a filter capacitor to smooth out power noises, which can slow down the startup by more than $32\mu s$. To further improve the wake-up speed, the proposed SoC uses a dedicated power-on voltage detection system that, using a hysteresis comparator, is capable of achieving less than $1\mu s$ of delay. Moreover, the proposed power-on system is designed to reduce leakage by 12.8×. The paper also introduces a Non-Volatile Radio Frequency controller (NVRF). This component features a non-volatile register file, so that it does not need to be reconfigured by the MCU after every power off. Because of the hardware controlled reinitialization, this peripheral can wake-up in parallel with the processor, shortening the wake-up latency from $33ms$ to $1.22ms$ [Wan+17]. Finally, the proposed SoC uses a power management unit (PMU) to control power-gating and sleep and wake-up signal generation [Wan+17].

This non-volatile SoC uses many techniques that are, in principle, common to other similar non-volatile processors, like the use of NVM as main memory, avoiding SRAM altogether, and the use of non-volatile flip-flops to achieve a fast and parallel backup and recovery of the system. These common techniques are also coupled with dedicated solutions like the power-on detection circuit and the non-volatile RF peripheral. The introduction of the NVRF is interesting as it brings non-volatility into the peripherals, allowing for fast recovery from power-off. The reinitialization of peripherals and the power-on detection contributes, according to the authors, to more than 93% of the active time of periodical normally-off tasks [Wan+17]. These components can in principle be adapted to other non-volatile processors.

However the proposed SoC does not improve upon the other shortcomings that are common to other non-volatile processor designs. In particular, the exclusive use of NVM as the main memory can cause a decrease in performance and an increase in the active power consumption, due to the slower speed of emerging NVMs and their higher energy consumption with respect to SRAM. Moreover, the use of non-volatile flip-flops comes with a significant area overhead, and can complicate the fabrication process. Consistency issues are finally not considered in this work.

1.6 Conclusion

This chapter has presented an overview of some works related to the design and optimization of intermittent systems powered with environmental energy harvesting. From these works emerge several interesting techniques, both at the software and hardware levels, to tackle the main issues of intermittent systems : the state retention and forward-progress across power failures. Some of these techniques are generic and could be applied in conjunction with one another. Below is a list of some interesting techniques taken from various works from this state of the art :

— Embedded nvRAM with integrated backup capacitor and cache [Zwe+11].
— Hybrid non-volatile memory elements [Yu+11].
— Adaptive size nvSRAM and SWT [Liu+16].
— Dedicated ISA instructions [Sak+14].
— Low-power retention and 2-bit predictor [Liu+16].
— Non-volatile autonomous peripherals [Wan+17].
— Software based peripheral state persistence [Ber+19].

The idea of adding an integrated capacitor to an NVM for completing any pending memory operation is not a novel one. This addition can be useful to mitigate the problem of bit errors if the supply suddenly drops during a write operation. However, while this is useful for the single operation, it does not give any additional guarantees to the full backup operation, which usually consists of many memory transfers. The use of NVM in conjunction with volatile memory elements to create hybrid nvFFs and hybrid nvSRAM is an interesting technique that tries to achieve the best of both worlds : fast and energy efficient access during normal operation, and state retention. However, this hybrid memory element come with a large area overhead, moreover they usually do not offer any protection against corruption of the data during backup. Finally, it is an intrusive solution, which comes with many manufacturing issues. The idea of introducing ISA level support, or new instruction to support save and restore operations, could open up the possibility of new and better software solutions. However, this will also require compiler and tool-chain support, which can make this kind of solutions difficult to apply. Low-power retention can, in some cases, avoid the cost of backup, but it cannot offer the same reliability of a true non-volatile copy.

The persistence of the peripheral state, is also a fundamental problem in the design of a transiently-powered system. In [Wan+17], a non-volatile autonomous Radio (NVRF) peripheral is presented. The main advantage of a dedicated non-volatile peripherals, is that the restore of the peripheral state, after a power failure, is much faster. Moreover, because the proposed NVRF is autonomous, its state can be restored in parallel with the restore of the main memory. While a custom non-volatile peripherals have this great advantage, they are also difficult to adopt and implement. To handle the more common volatile peripherals, in a transiently powered system, other software level solutions, such as Sytare [Ber+19], have been proposed. In this work the main focus is on the backup of the main system memory, the problem of saving and restoring the peripherals state, across power failures, is not analyzed as existing solutions such as the one presented in [Wan+17] or [Ber+19] could be applied to the techniques presented in this dissertation.

Table 1.3 offers an overview of some significant approaches from the state of the art. The table qualitatively compares these works based on high-level properties of the technique in use, such as the memory architecture, the type of backup operation, its triggering event and the size and speed of the backup. Moreover, the table reports whether

the technique offers detection and/or protection for backup errors, or whether it requires special hardware. The table also shows if a strategy comes with particular overhead or if it requires high endurance memory and if it has high or low active energy consumption Finally, the last metric is if a technique is easy or difficult to apply and adopt.

Table 1.3 shows that fully non-volatile processors require special hardware components but come, in general, with very fast backup speed. This fast backup operation is mainly due to two factors : the parallel nature of the backup (*i.e.*, all/many registers or memory location are saved in parallel), and the usually small size of the backup. In fact, fully non-volatile processors tend to use NVM as the only type of main memory, so only the internal registers need to be saved. Moreover these registers are saved in-place as each flip-flop is paired with a non-volatile element, so no transfer to main memory is necessary. The main drawbacks of these approaches come from the significant increase in area, the slower and more energy hungry read and write operations, as well as the lack of detection and mitigation for backup corruptions, which could cause the corruption of the entire system state. Moreover, a concern with these fully non-volatile processors is the endurance of the embedded NVM devices, which can die out due to wear and normal aging or even be the target of attacks, as demonstrated in [CYL18a]. Another issue often introduced by the use of non-volatile flip-flops, or by hybrid nvSRAMs, is the high peak-currents that are required to write in parallel all the non-volatile elements. In fact, due to the higher write energy required to write NVM cells, it is often not possible to backup all the registers or memory locations at the same time.

Table 1.3 also reports some of the software-level solutions presented in the previous sections. These kind of solutions can be based around compile-time static techniques and/or around run-time techniques. The main advantage of most software-based solutions is that they are simple to implement on existing platform. Software-based solutions tend to rely on similar platforms based around the *SRAM+NVM* architecture, and usually do not require any exotic or dedicated hardware solutions. However, there are some exceptions with software-based solutions requiring custom processors (*e.g.*, [Zha+17]), or software solutions implementing a custom programming paradigm like Chain [CL16], which makes these techniques more difficult to apply. Software solutions usually rely on a sequential backup, implemented in software, as summarized in Table 1.3. This means that the data transfer is slower than in the case of hardware techniques. However, the data transfer type is not the only thing that influences the backup speed. In fact, some works present techniques to optimize the size of the backup, which in turn will reduce the overall backup

time. However, despite these optimizations, software-level solutions tend to be much slower than the hardware based solutions. In summary, software-level solutions, while being simpler to implement, tend to be slow and not very energy efficient.

For all of the above reasons, in this dissertation, we present a different approach, based on the idea of a dedicated hardware controller, to enable transiently-powered computing, with fast and efficient on-demand backups, on an SRAM-based platform equipped with NVM for backups. The principles that guided these choices are to exploit the still significant run-time advantages of SRAM over emerging NVMs, as well as to improve over the slow and inefficient software-based backup process, without incurring in the cost of exotic and expensive circuit-level or device-level solutions, which are impossible to apply to existing hard IPs. Thus, this thesis makes the choice of a dedicated backup controller, which gives an acceleration with respect to software, while being just a component that can be integrated in an SoC, alongside existing IPs.

We looked at improving the *on-demand* backup scheme, by introducing a backup controller, Freezer, in Chapter 3, that uses run-time information to implement an incremental backup technique that greatly reduces the size of the saved state.To address the problem of consistency errors due to backup interruption, two incremental backup algorithms are presented in Chapter 4, that improve on Freezer by always having a safe consistent state that can be restored in case of consistency errors. Finally, the non-volatile processor approach, based on nvFF, does not offer any path for scalability to faster and more capable processors. This is mostly due to the huge overhead introduced by nvFF [Bar+13], as well as the lower clock speed imposed by a NVM-only designs. On the other hand, as shown in Chapter 5, our backup controller approach can be extended, without significantly increasing the number of registers, or the area, to support much larger address spaces.

Table 1.3

Work	Arch	Backup Type	Backup Trigger	Backup Data	Backup Speed	Backup Error Detection	Backup Error Penalty	Required Special HW	Overhead	Required Endurance	Active Energy	Applicability
Chain [CL16]	SRAM + NVM	message passing	task end	task	NA	detection	task rollback	none	NA	low	good	difficult
Stack analysis [Zha+15]	SRAM + NVM	serial	on-demand (V_{th})	optimized stack +regs	slow	possible	rollback	none	NA	low	good	good
Stack analysis [Zha+17]	SRAM + nvFFs	hardware serial	on-demand (V_{th})	optimized stack regs	fast	none	system corruption	custom CPU nvFFs	very high area high peak power	low	good	difficult
Mementos [RSF11]	SRAM + NVM	software serial	compile time static/periodic	stack + regs	slow	detection	rollback	none	very high time/energy	low	good	very good
Hibernus [Bal+15 ; Bal+16]	SRAM + NVM	software serial	on-demand (V_{th})	full memory	very slow	possible	restart application	none	high time/energy	low	good	very good
Quick-Recall [Jay+15]	NVM only	software serial	on-demand (V_{th})	regs	fast	detection	restart application	none	very high time/energy	high	bad	very good
EM-map [Jay+17]	SRAM + NVM	software serial	on-demand (V_{th})	regs / regs + stack	slow	none (mitigation)	system corruption	none	NA	low	good	limited
NVP with FeFFs [Wan+12]	SRAM + nvFFs	hardware parallel (FFs)	on-demand (V_{th})	regs	very fast	none	system corruption	custom CPU nvFeFFs	very high area high peak power	low	good	very difficult
nvFFs mini arrays [Bar+13 ; Kha+14]	NVM only + nvFFs mini arrays	hardware mini array	on-demand (V_{th})	regs	very fast	none	system corruption	nvFFs mini arrays	area time energy	high	bad	difficult
STT nvMCU [Sak+14]	NVM only + nvFFs	hardware parallel	on-demand software cycle logging	modified regs	very fast	none	system corruption	nvFF + dirty bit custom CPU	high area high energy time	high	bad	very difficult
nvSRAM NVP [Liu+16]	nvSRAM + nvFFs	hardware parallel	timeout/ predictor	active SRAM regs	very fast	none	system corruption	nvSRAM nvFFs	very high area	low	good	very difficult
Brainshift [Hag+17]	scan FF + retention SRAM	hardware serial	on-demand (V_{th})	regs retention sram	slow	none	system corruption	retention SRAM special scan mode	NA	NA	good	limited/ volatile solution

Chapitre 2

Non-Volatile Memories

2.1 Introduction

In this chapter, the main emerging Non-Volatile Memory (NVM) technologies are introduced. This chapter gives an overview of the basic working mechanism of these NVMs, as well as it highlights the respective advantages and drawbacks of each technology. Finally, the main contributions of the chapter is a presentation of a comparison between the different NVMs, as well as the other memory technologies such as SRAM, DRAM and FLASH, with data gathered by various surveys and reviews from the literature.

The rest of the chapter is organized as follows : Section 2.2 presents the Phase Change Memory (PCM) technology. Section 2.3 introduces Magnetic RAM (MRAM) with all its variations. Section 2.4 gives an overview of Resistive RAM (RRAM). Section 2.5 presents the Ferroelectric RAM (FRAM). Finally, Section 2.6 provides a comparison of the different memory technologies with a collection of data from different works from the literature.

2.2 Phase Change Memory

The basic working principle of Phase Change Memories (PCMs) is to exploit the large difference in resistance, between the amorphous and crystalline state of so called *Phase Change Materials* to represent the information. This idea has been first explored in the 1960s [Ovs68 ; Bur+10 ; HAW14], however, thanks to more recent advances in the field of phase change materials, it has received a renewed interest from both academia and industry [Riz+19]. PCM is one of the most mature among the so called *"Emerging Non-Volatile Memories"* [Riz+19], with many prototypes and devices that have been demonstrated and with some devices already available on the market.

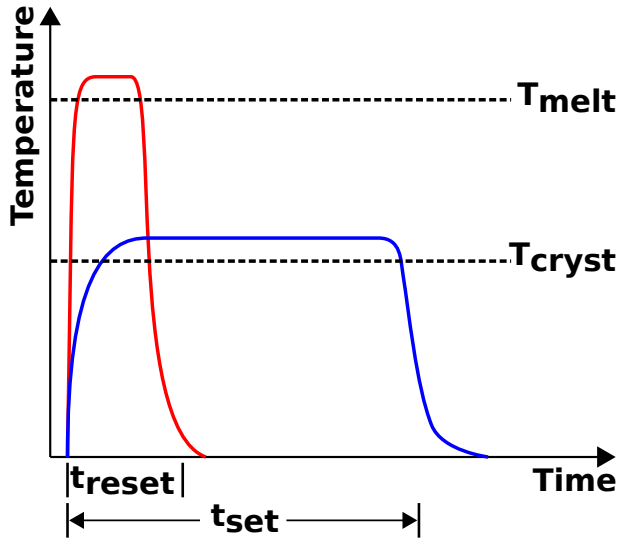


FIGURE 2.1 – SET and RESET pulses to switch state in PCM.

2.2.1 SET and RESET write operations

The PCM memory cell is based on the property of chalcogenide materials to exist in either the amorphous or the crystalline state at room temperature. These two states show a large difference in resistance, typically between 100× and 1000× but that can be up to four orders of magnitude [Bur+10 ; YC16]. The crystalline phase is associated to a low resistance state *LRS*, whereas the amorphous state has a relatively high resistance *HRS*.

Figure 2.1 shows the programming pulses required to switch from the *RESET* amorphous *HRS* state, to the crystalline *LRS SET* state, and vice-versa. The switching between these two phases is thermally induced through Joule heating. In particular, to transition from the crystalline to the amorphous phase, a very short pulse of high current is applied to the device. This rapidly heats the chalcogenide material above the melting point. The short duration of the current pulse allows the material to cool down quickly and settle to an amorphous state. To obtain the opposite transition from the amorphous *HRS* to the crystalline *LRS* phase, a current pulse with lower intensity is applied for a slightly longer period of time. This current pulse heats the memory element above the crystallization temperature, and lasts long enough to allow for the material to crystallize. As it can be seen from Figure 2.1, the *SET* transition, shown in blue on the figure, from amorphous to crystalline phase, is the limiting factor in terms of write time for this types of memory. On the other hand, the *RESET* transition, shown in red in the figure, is faster but it imposes the limit on the peek power, due to the required high current.

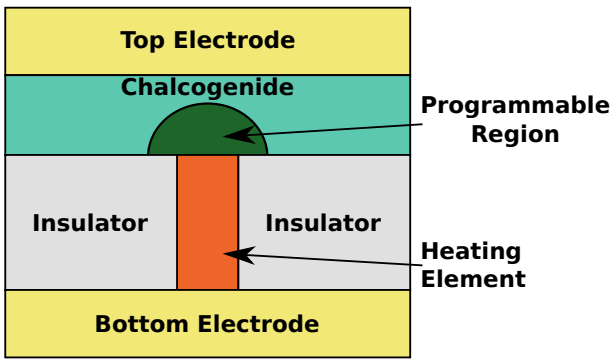


FIGURE 2.2 – Typical mushroom type memory cell structure of a PCM memory.

2.2.2 PCM Cell

Figure 2.2 shows the typical PCM cell with a so called "mushroom" structure. The cell is composed of two electrodes that sandwich the heating element and a thin layer of the phase-change material, which is the memory element itself. There are several materials that have been identified, which have the properties of the two stable state at room temperature. PCM cells are commonly based on the so called *GST* materials like $Ge_2Sb_2Te_5$ [YC16 ; HAW14 ; Xue+11], compounds of *Germanium Antimony* and *Tellurium*. These and other materials, such as silver and indium doped Sb_2Te (AIST), have attracted the interest in the research community, due to their sub-100ns crystallization times [Bur+10 ; HAW14]. Other structures than the typical mushroom structure have been proposed and are summarized by Fujisaki in [Fuj13]. Moreover, as reported by Yu and Chen in [YC16], the general trend seems to be to move to a so called *pillar* structure, which improves write currents by better confining the heat dissipation.

The memory cells in PCM arrays can be as simple as a one transistor one resistor (1T-1R) devices, with the resistive memory element and an access transistor. The access transistor can either be a FET or a BJT, with BJTs being more promising for both speed and scaling [Lee+09]. However, because the transistor can be the limiting factor to scalability [Der+09], alternative selector devices have been proposed, such as a diodes [YC16] or Ovonic-threshold-switch (OTS), where the selector is in series with the memory element and occupies the same footprint [HAW14 ; Der+09]. As an example, a *Cross-Point* PCM memory was demonstrated by DerChang *et al.* in [Der+09], where they integrated vertically the PCM and the OTS, as shown in Figure 2.3.

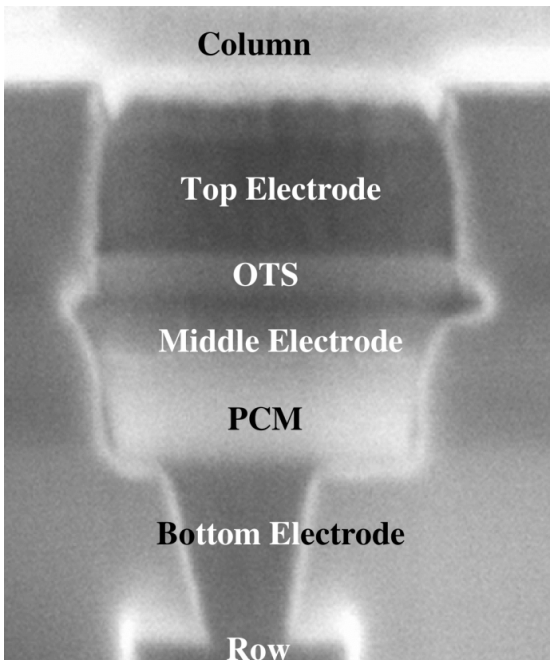


FIGURE 2.3 – Vertical stacking of PCM and OTS as presented in [Der+09]

2.2.3 Characteristics

The main advantages of PCM technology are the maturity of the process, its compatibility with CMOS, and the promise of good scalability and high density. The write speed is limited by the *SET* operation, which, as shown in Table 2.1 can be as low as $20ns$, even though more common values are above $100ns$ [Lee+09]. The *RESET* transition on the other hand can be relatively fast, usually less than $60ns$ [Lee+09], or even sub $10ns$ as demonstrated in [Der+09]. However, the transition to an amorphous state also requires a much higher current, that can be up to two times as much as the current for the *SET* transition. However, because of its fundamental working principle, the technology shows slightly worse performance, especially in terms of energy per bit, which is in the order of $\sim 10pJ/b$ for write operations, than other emerging NVMs as summarized in Table 2.1. Additionally, the endurance is also limited, with values ranging from 10^5 to 10^9, which is an improvement with respect to Flash memories, but is still limited when compared to DRAM or other NVMs such as the STT-MRAM presented in the following section.

2.3 Magnetic RAM

In this section, we use Magnetic RAM, or *MRAM*, to indicate a family of memories that use a device called *Magnetic Tunnel Junction* (MTJ) to store the information. The working principle of the MTJ, which is common to many different types of MRAMs, is to store the information as the direction of the magnetization of two ferromagnetic layers. The different types of *MRAMs* use different techniques and physical phenomenons to switch the direction of one of the two layers of the MTJ. These techniques are used to put the two magnetic directions into either a *parallel* or an *anti-parallel* state. These two configurations are used to encode the information, as they show a significant difference in resistance through a phenomenon that is called *"tunnel magnetoresistance"* (TMR) [Moo+95].

2.3.1 Magnetic Tunnel Junction

The Magnetic Tunnel Junction (MTJ) is the basic building block of a variety of MRAMs. Conceptually, MTJ is a simple device composed of a thin tunneling dielectric (*e.g.*, MgO), sandwiched by two ferromagnetic layers. One of the ferromagnetic layers is called the *"fixed"* or *"pinned"* layer, because it holds a fixed magnetic direction. On the other hand, the other layer is called the *"free"* layer, as its magnetic direction can be changed. This creates two possible configurations : one where the direction of the magnetization of the two layers is parallel, and one in which they have opposite direction and are anti-parallel. When the magnetization of the two layers is parallel, the resistance of the MTJ is smaller than when they are anti-parallel. This difference in resistance is due to the TMR effect and can be as high as $200 - 300\%$ [And+17].

2.3.2 Field Induced Magnetic Switching MRAM

The first generation of MTJ-based MRAMs used *Field Induced Magnetic Switching* to rotate the magnetization direction of the free layer. One type of memory that uses this mechanism is the so called *Toggle MRAM* [Eng+05], which is a relatively mature technology with products that are available in the market, manufactured by companies such as NXP [Dur+03] and Everspin [Evec]. The typical memory cell for Toggle MRAMs is a one-transistor one-MTJ cell, as depicted in Figure 2.4. As shown in the figure, the cell uses two write wires that carry a pulsed current, which generates a field that induces the

switching in the MTJ. The read operation is performed by activating the access transistor

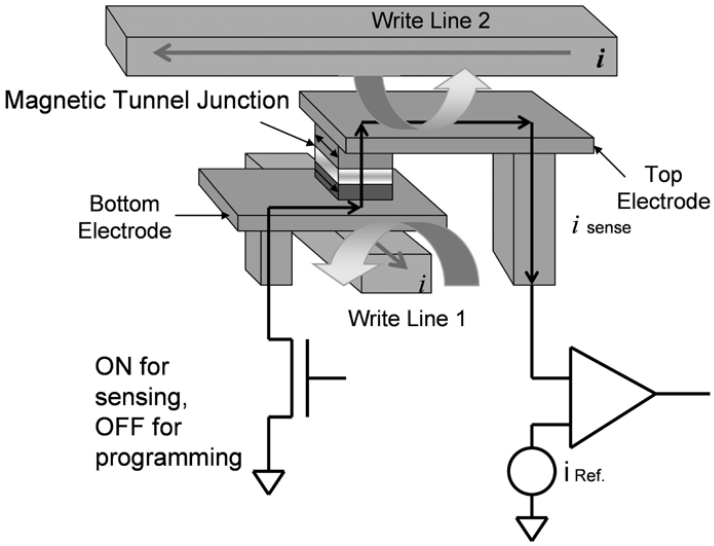


FIGURE 2.4 – Toggle MRAM cell from [Eng+05]

and letting a sense current flow through the device.

This type of memory can be arranged in a crossbar array, and has good performance with vendor such as Everspin producing devices with $35ns$ symmetric read/write access and claiming unlimited endurance and long retention times [1]. However, due to the additional wires and the high current required for the switching (in the order of few mA), which does not reduce with scaling [And+13 ; JBT15], this type of MRAM are difficult to scale below 90 nm [Sen+16a]. For this reason, both the research community and the industry have researched other switching techniques to overcome the scaling problems of Toggle MRAM.

2.3.3 Thermally Assisted Switching MRAM

Thermally Assisted Switching MRAM (TAS-MRAM) uses heat to favor the switching of the MTJ state. This technique is also a form of field induced switching, however, instead of requiring two high currents lines to induce the switching, it requires a single high switching current to generate the magnetic field [HAW14 ; JBT15]. Additionally, a small current is also used to heat the MTJ before the switching current is activated. The structure and switching mechanism of TAS-MRAM is illustrated in Figure 2.5. The stack

1. EVERSPIN, *Toggle MRAM Technology | Everspin*, URL : `https://www.everspin.com/toggle-mram-technology` (visité le 10/02/2021).

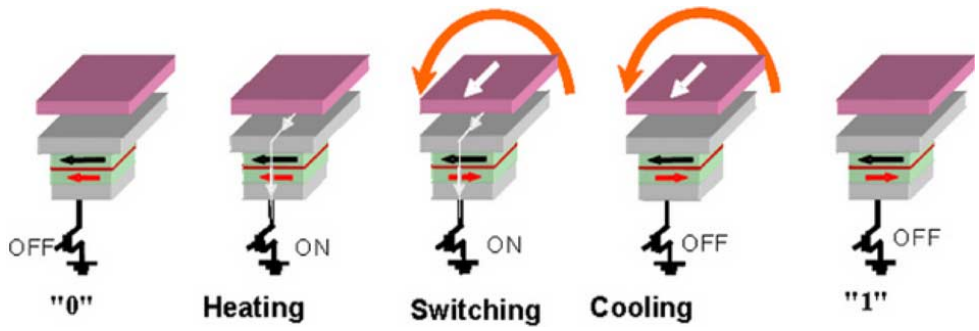


FIGURE 2.5 – TAS-MRAM cell switching from [Sen+16a]

of a *TAS-MRAM* MTJ is slightly different as it requires the replacement of the free layer, with an exchange bias layer, obtained by coupling the free layer with an anti-ferromagnetic layer [HAW14]. This layer has a lower blocking temperature than the one used to lock the *reference* layer, and it is used to pin to the free layer.

As shown in Figure 2.5, the switching process requires three major steps : *heating*, *switching*, and *cooling*. To begin the switching process, the access transistor is switched on, allowing a current to flow through and heat the device. Once it is heated above the free layer blocking temperature, the free layer magnetization can be changed through the application of an external field. To terminate the write operation, the heating current is switched off, while the external field is kept also during the cooling of the device. The addition of the blocking layer improves the data reliability, by improving write selectivity and thermal stability, while also reducing the power consumption with respect to Toggle-MRAMs [HAW14 ; JBT15]. However, one of the drawbacks of this write scheme, is that it increases the write time, due to the time required by the heating and cooling processes. Additionally, even though the write current and power are reduced when compared to field-switching MRAM, writes still require a significant amount of current (in the order of few mA) [Sen+16a ; JBT15].

2.3.4 Spin-Transfer Torque MRAM

STT-MRAM is a type of magnetic memory that exploits the *Spin-Transfer Torque* effect to switch the direction of the free layer. This effect allows the storage layer to be written only by the current flowing through the MTJ, without needing the strong external magnetic field, like the previous types of MRAMs [HAW14 ; Sen+16a]. The basic idea behind this type of switching is to let a current carrying spin-polarized electrons

flow through the device. These spin-polarized electrons exert a torque on magnetization of the free layer [Bri+14]. When this current is above a certain threshold, called switching current, it can change the magnetization direction of the free layer. The amplitude of this current depends on the physical characteristics of the MTJ such as materials and structure, but it is also heavily related to the amount of time the current is applied to the MTJ [LC17]. An advancement over the standard *In-Plane* MTJ-based STT-MRAM is represented by the *Perpendicular* MTJ based STT-MRAM. In perpendicular MTJ, the magnetization direction is perpendicular to the surface of the device. This type of structure is promising as it shows better thermal stability and reduced switching current, which should enable better scaling [HAW14 ; Bri+14].

STT-MRAM promise good scalability as the switching current is reduced proportionally by scaling the area of the MTJ [And+13]. The cell size can in theory be as small as $6F^2$ [And+17]. Products with different capacities have been demonstrated and commercialized, going from a 64Mb memory produced on a 90 nm CMOS node [Sla+12], up to the more recently presented 1Gb memory, produced on a 28 nm node [Agg+19]. Finally, STT-MRAM can be considered as one of the more promising types of memory among the emerging NVMs, with the fastest access time, in the order of ns, and the best endurance ($\sim 10^{15}$). For these reasons, STT-MRAM is seen as a good candidate to replace DRAM or even SRAM in some applications.

However, STT-MRAM, still faces some problems that have to be addressed before it can truly replace DRAM or SRAM. STT-MRAM has a relatively small on/off resistance ratio, which complicates the design of the sensing circuitry [Che16 ; Riz+19]. STT-MRAM is also affected by variability, posing a problem in the design of highly integrated memories [Fuj13]. Moreover, the fabrication and integration with the CMOS process also poses some issues related to the use of exotic materials and to the required temperatures [YC16 ; Che16].

2.4 Resistive Memories and Memristors

While both MRAM and PCM exploit a change in resistance to store the information, the term Resistive RAM, *ReRAM* or *RRAM*, refers to a particular class of devices that store the information as the resistance change across a dielectric solid-state material. These types of two terminal devices are also known as *memristors*, and were first defined theoretically by Chua in 1971 [Chu71]. However, it was only in 2008 that HP

Labs demonstrated the appearance of memristance in some nanoscale systems [Str+08]. Since then, the research has progressed and several types of RRAM devices have been demonstrated, with different types of materials.

The basic structure of an RRAM device is a *metal-insulator-metal* structure, where a thin oxide film or a solid-state electrolyte is sandwiched between two metal electrodes [Fuj13 ; LC17 ; Riz+19]. The switching mechanism exploits the formation or rupture of a conducting filament, which is induced in the insulating film by an applied voltage, as shown in Figure 2.6 [Val+11 ; And+17]. The resistance switching can happen either in an

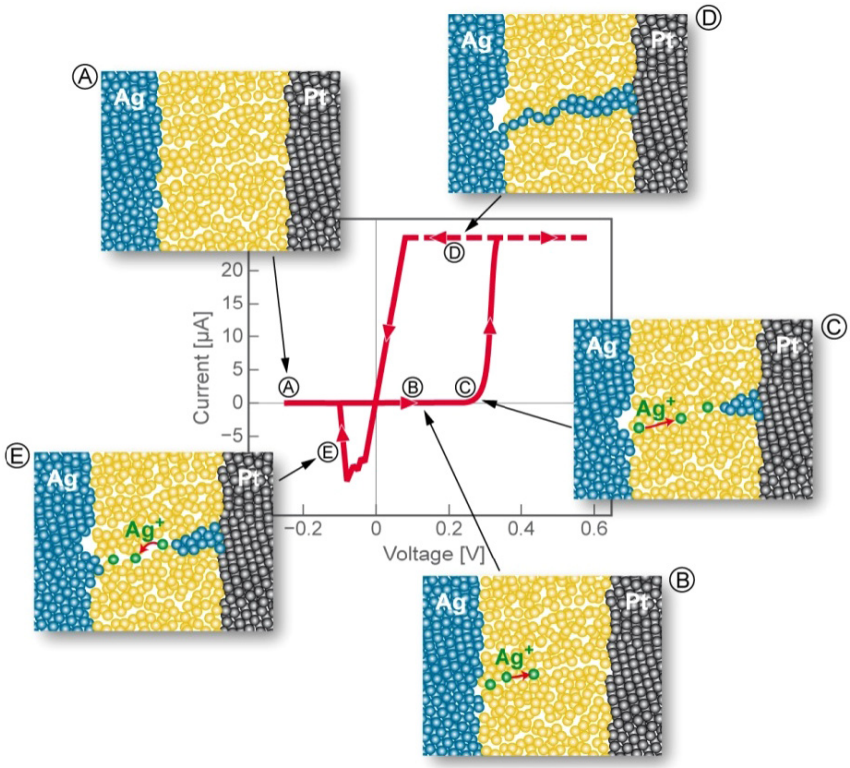


FIGURE 2.6 – The formation ($A \rightarrow D$) and rupture (E) of the conductive filament in a CBRAM device [Val+11].

unipolar or a bipolar fashion, depending on the type of RRAM device. Figure 2.7 shows the switching $I - V$ curve, in both unipolar and bipolar modes, for a HRS to a LRS transition, called *Set*, and the opposite transition from an LRS to a HRS called *Reset*, as reported in [Yu16]. Devices that operates in the unipolar mode, such as many oxide-based RRAM, make it easier to achieve high integration with cross-point cells array, using a diode as a selecting device [Fuj13 ; And+17]. However, the bipolar mode is considered more promising as it shows improved stability, requires lower currents and exhibits better

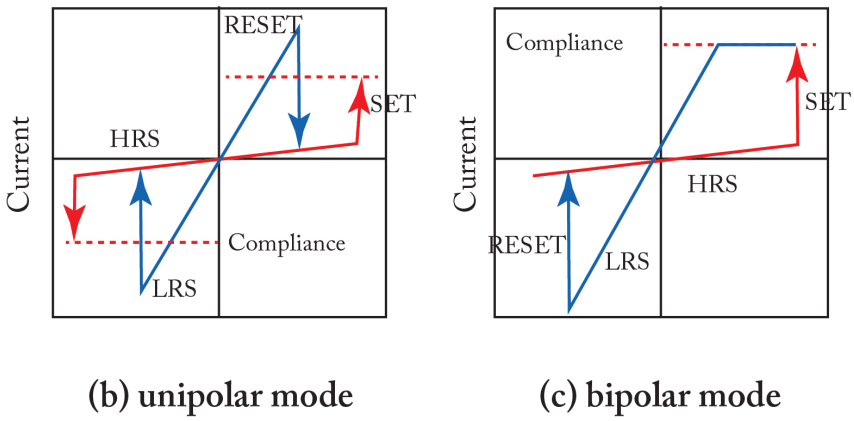


FIGURE 2.7 – RRAM switching $I - V$ characteristics in unipolar and bipolar mode, from [Yu16].

endurance. Moreover, more advanced structures, with two combined resistive elements, could enable large passive cross-bar arrays without the need for a selector device [Val+11 ; Fuj13].

Before being able to perform these resistance switching, many RRAM devices, require to undergo a procedure called *forming*. This is a one-time process, that is similar to the *Set* operation, but with higher voltages that need to cause a soft breakdown of the Metal-Oxide-Metal structure [AS10]. The voltage required for the forming process used to be relatively high, and was one of the main disadvantage of early resistive memories. However, technological advancement and the shrinking of the oxide thickness, have made possible to eliminate the requirement for the forming process [Yu16].

There are two main families of RRAM devices, based on two different types of switching : oxide-based RRAM or *OxRAM*, and conductive-bridge RRAM, also known as **CBRAM**. OxRAM exploits resistance switching in metal-oxide materials, examples are HfO_2, Ta_2O_5, NiO, Cu_xO, TiO_x, and some of the so called perovskite oxides [MSS12 ; Che16 ; Fuj13]. The materials involved are already commonly used in the semiconductor industry, and RRAM shows a great compatibility with CMOS process [AS10 ; YC16]. In OxRAM devices, the conductive filament is formed by oxygen vacancies, these vacancies are introduced with the movement of oxygen atoms induced by an electrical field. Additionally, this process may be also aided by the diffusion of oxygen atoms due to Joule heating [And+17]. The rupture of the conductive filament is due to a redox reaction in the oxide layer [MSS12].

Conductive bridge RRAM (CBRAM) are a family of RRAM devices based on the formation of a conductive filament made of metals ions, such as Ag or Cu, formed through electrochemical reactions, in a solid-state electrolyte [MSS12 ; Che16 ; And+17]. These metal ions are provided by one of the two electrodes as shown in Figure 2.6. CBRAM devices generally operate in a bipolar way [Val+11] and they show high On/Off resistance ratio ($100\times - 1000\times$). CBRAM have usually faster switching speed with respect to OxRAM devices, but they generally show reduced endurance, with values in the order of $10^4 - 10^5$ cycles, even if higher values have been demonstrated [Val+11 ; Fuj13 ; YC16]. In OxRAM devices, On/Off resistance is usually smaller than CBRAM, in the order of $10\times - 100\times$ [YC16 ; And+17]. However OxRAM shows better endurance than CBRAM, with values that can be higher than 10^9 cycles.

RRAM has attracted a lot of attention from both research and industry. Its main advantages include its excellent compatibility with the CMOS process and the potential for great scalability. RRAM devices have been demonstrated to scale down to sub-10 nm [Yu16]. Moreover, test chips with large capacities have been fabricated, such as a 32Gb memory based on OxRAM [Liu+13] and a 16Gb memory based on CBRAM [Fac+14 ; YC16 ; Che16]. RRAM also shows a lot of potential for high densities and high capacities, as cross-point arrays enable $4F^2$ cell sizes, where F is the feature size (e.g. 28 nm). Moreover, 3D stacking has been demonstrated for both unipolar and bipolar RRAM devices, and should enable even higher capacities [LC17 ; Liu+13]. A further increase in capacity could be enabled by RRAM devices supporting multi-level cells, 2-bit and 3-bit multi-level devices have already been demonstrated [Yu16].

With these characteristics, RRAM could replace Flash at least for those applications that require faster access times, or complement it by sitting between DRAM and Flash as a storage class memory [Yu16]. While the theoretical switching time for RRAM could be reduced below $10ns$, its current performance is still not on par with those of DRAM. Moreover, in order for RRAM to replace DRAM, the biggest obstacle is the endurance, which is till not sufficient and would require both technological improvements as well as architectural wear-leveling techniques [AS10 ; Yu16].

Other interesting applications for RRAM are also being explored, these include its use in neuro-inspired computing where RRAM cells are used as analog devices to emulate the function of synapses in neural network applications [YC16 ; Yu16].

2.5 Ferroelectric RAM

Ferroelectric RAM, commonly called *FRAM* or *FeRAM*, is a type of memory that store the information under the form of a remnant polarization in a layer of ferroelectric material. Polarization is the effect by which the dipoles present in a dielectric material orient themselves in an anti-parallel fashion under the effect of an electric field, as shown in Figure 2.8 [GP10]. Normally, the polarization disappears when the electric field is removed, however ferroelectric materials show a remnant polarization.

There are many materials that show these ferroelectric properties, however the only relevant ones are oxides [GP10]. In particular, FRAM is commonly based on *lead zirconate titanate* materials $Pb(Zr_xTi_{1-x})O$, also called *PZT* [GP10 ; HAW14 ; Bou+17]. Other materials that are also common in FRAMs are the so called *SBT*, based on *strontium bismuth tantalate* $SrBi_2Ta_2O_9$ [GP10 ; HAW14]. Besides these materials, others have also been investigated, focusing in particular on the reduction of the thickness and dimensions of the ferroelectric layer [Fuj13 ; HAW14].

The cell structure of FRAM memories is similar to that of DRAM, consisting of one transistor and one capacitor (1T-1C) [Bou+17]. In the case of FRAM, the capacitor is formed with a ferroelectric layer sandwiched between two metallic electrodes [HAW14 ; Bou+17]. With this ferroelectric capacitor *FCAP*, the information can be encoded as the two possible directions of the remnant polarization, as shown in Figure 2.8. This allows FRAM to retain the information in a non-volatile way. Moreover, unlike DRAM, FRAM does not require any refreshing operation to retain the information [Bou+17]. The typical polarization versus voltage characteristic of FRAM creates a hysteresis loop, as shown in Figure 2.9. The figure also shows the two remnant polarizations, P_R and $-P_R$, that

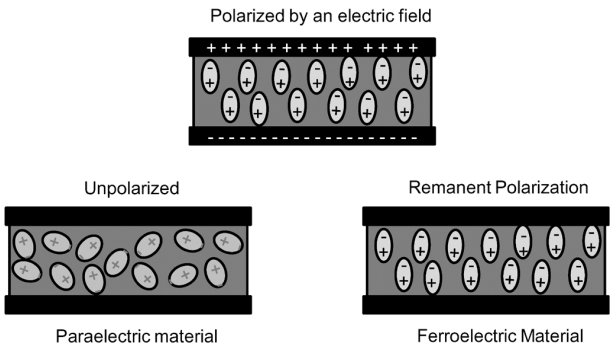


FIGURE 2.8 – Polarization of a dielectric and remnant polarization in ferroelectric materials from [GP10].

persist even when the applied voltage is zero. The presence of a remnant polarization

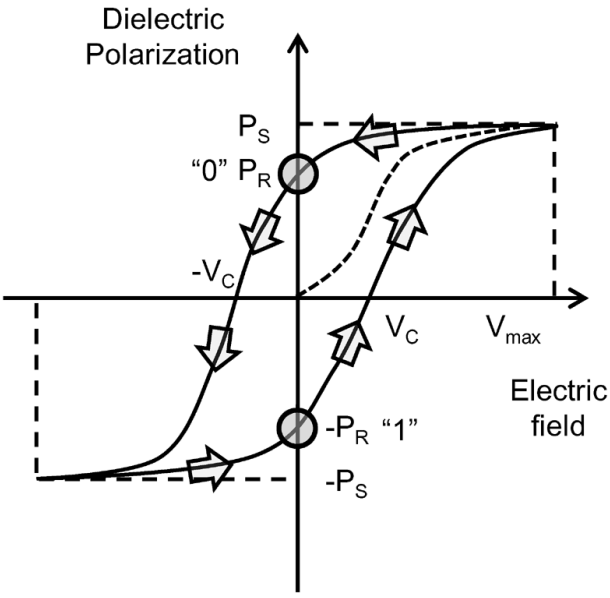


FIGURE 2.9 – Hysteresis loop of the polarization vs voltage curve, from [GP10]

allows the FCAP to retain charge, which can be sensed in a destructive way similarly to what happens in DRAM [GP10]. This means that every time that a cell is read, it must also be rewritten to retain its content.

FRAM was one of the first, among the emerging NVMs, to be transferred to production, where it has found its place as an embedded memory for smart-cards and low-power micro-controllers [Fuj13 ; HAW14]. FRAM has performance comparable to that of DRAM with access times smaller than $55ns$ [And+17 ; GP10 ; Bou+17]. Moreover, FRAM is the only NVM that shows high speed write operations, with write times smaller than $6ns$ that have been demonstrated on an SBT-based device, where the speed was limited by the CMOS devices [HAW14]. FRAM also shows excellent endurance characteristics when compared with other NVMs, with usual values larger than 10^{12}, and values as high as 10^{15} cycles, as reported in [Bou+17].

The main drawback of FRAM is scalability, as it is reportedly difficult to scale to advanced nodes [Bou+17]. In particular, FRAM does not offer the same level of density of DRAM, with typical cell sizes around $\sim 22F^2$ [GP10 ; And+17]. To achieve higher density and scalability, advancements in the manufacturing technology are needed, as well as the development of 3D architectures [HAW14 ; And+17].

Beyond the DRAM like capacitor cell, ferroelectric materials are being researched to develop new types of memory cells. Other types of cells include architectures such vertical multi-level capacitor cell, as the one described in a recent patent [FX20]. Another class of ferroelectric memories are the ones based on *transistor-type* cells, where the ferroelectric layer replaces the oxide in a MOS-like structure [Fuj13 ; Che16]. These types of ferroelectric memories based on field effect transistors are called *Ferroelectric-FET* or *FeFET* memories [Che16]. Although FeFETs have demonstrated fast access times $\sim 20ns$ and promise good scalability [Che16], they are the least mature technology among the emerging NVMs. Moreover, FeFETs currently suffer from severe limitations when it comes to endurance, which is in the order of $10^4 - 10^5$ cycles, and show poor retention [Fuj13 ; Che16].

2.6 Comparison

This section offers a brief comparison of the different NVM technologies previously discussed, as well as presenting the data collected from various works in the state of the art. Figure 2.10, extracted from [Che16], shows the evolution of *PCM*, *STTRAM* and *RRAM* memories, by reporting the capacity and technology node of various fabricated chips, in the years between 2003 and 2015. The figure shows how different manufacturers are investigating these emerging NVM as well as the evolution in the capacity and integration over the years. One clear data point that emerges from Figure 2.10 is the relative cell size difference of the different technologies. In particular, it is evident the higher density of *RRAM*, and especially *PCM*, with respect of *STT-MRAM*. As an example, in Figure 2.10, a fabricated chip by Samsung in 2004, on a $180nm$ node, achieved 64Mb of capacity, whereas, only in 2012, Everspin presented a 64Mb STT-MRAM fabricated on a $90nm$ node.

This point is also generally in agreement with the data collected from the various surveys and works from the state of the art. This data is summarized in Table 2.1, where the relevant parameters, such as cell area, read and write time, endurance etc., are reported for the different memory technologies. As also shown in Figure 2.10, some technologies such as *RRAM* and *PCM* do show better cell area and thus density, with respect to other technologies. However, Table 2.1 also shows the other relevant metrics that are missing from Figure 2.10, such as the better write and read time and energy generally shown by *STT-MRAM* with respect to *PCM* and *RRAM*.

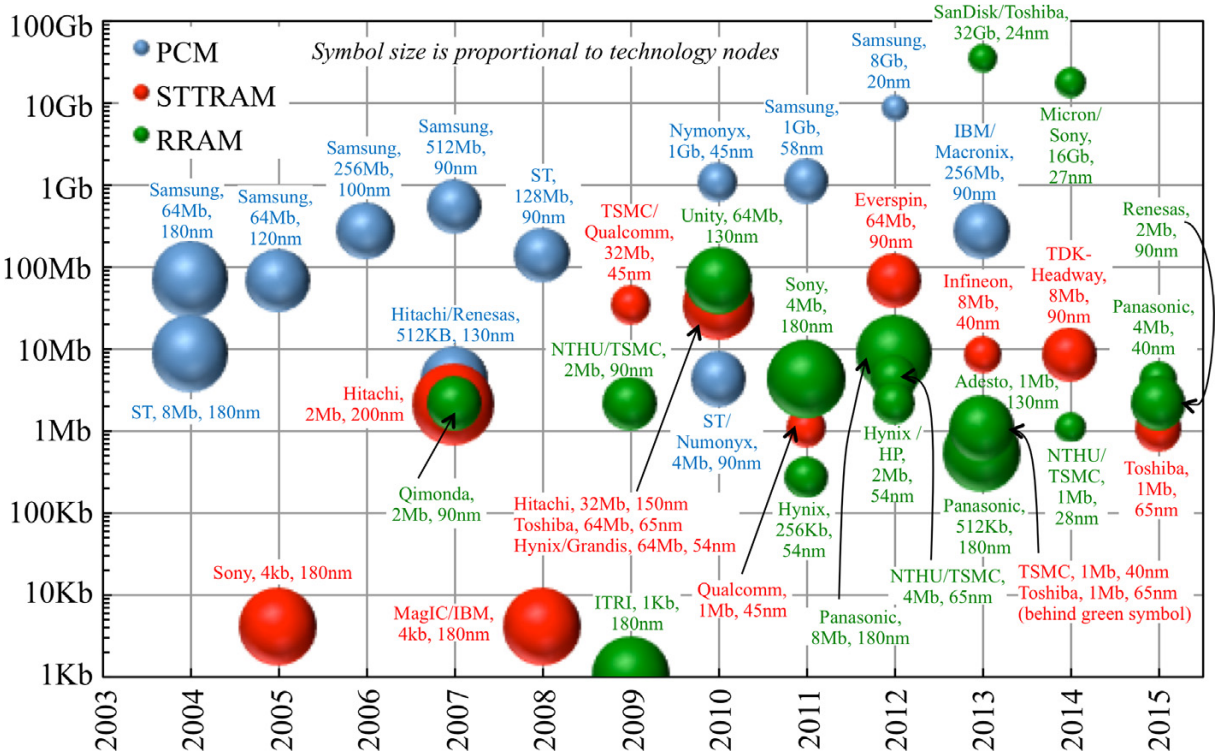


FIGURE 2.10 – Comparison of different NVM chips from [Che16]

TABLE 2.1 – Comparison of different memory technologies according to different authors and works.

	SRAM	DRAM	NOR	NAND	STT-MRAM	PCRAM	RRAM	FeRAM	Paper
Cell area F^2	> 100	6	10	< 4 (3D)	6∼50	4∼30	4∼12	-	[YC16]
	120-200	60-100	-	4-6	6∼50	4-12	4-10	6-40	[Bou+17]
	-	-	-	4	20-40	4-20	20	22	[Chi+16]
	100 - 300	6	30	2 -4	6	4	4	20	[And+17]
Read time	∼1 ns	∼10 ns	∼50 ns	∼10 us	< 10 ns	< 10 ns	< 10 ns	-	[YC16]
	∼0.2-2 ns	∼10 ns	-	15-35 us	2-35 ns	20-60 ns	∼10 ns	20-80 ns	[Bou+17]
	-	3 ns	-	75 ns	10 ns	30 ns	10 ns	60 ns	[Chi+16]
	2.252 ns	50 ns	-	∼100 us	6.181 ns	50-100 ns	-	-	[Xue+11]
	10ps - 5 ns	10 - 40 ns	5 ns	10 us	2 - 4 ns	20 ns	10 ns	55 ns	[And+17]
Write time	∼1 ns	∼10 ns	10 us - 1 ms	100 us - 1 ms	< 10 ns	∼50 ns	< 10 ns		[YC16]
	∼0.2-2 ns	∼10 ns	-	200 - 500 us	3-50 ns	20-150 ns	∼50 ns	50-75 ns	[Bou+17]
	-	6 ns	-	2 us	20ns/6ns	100ns/20ns	25 ns	60 ns	[Chi+16]
	2.244 ns	20-50 ns	-	∼100 us	6.181 ns	∼1 us	-	-	[Xue+11]
	10ps - 5 ns	10 - 40 ns	5 ns	500 us / 1ms	2 - 4 ns	1 us	10 ns	55 ns	[And+17]
Retention	N/A	∼64 ms	> 10 y	> 10 y	> 10 y	> 10 y	> 10 y	-	[YC16]
	-	-	-	10 y	10 y	10 y	> 10 y	10 y	[Chi+16]
	-	-	< 10 y	< 10 y	< 10 y	< 10 y	< 10 y	< 10 y	[And+17]
Endurance	$> 10^{16}$	$> 10^{16}$	$> 10^{5}$	$> 10^{4}$	$> 10^{15}$	$> 10^{9}$	$10^{6} - 10^{12}$	-	[YC16]
	10^{16}	$> 10^{15}$	-	$10^{4} - 10^{5}$	10^{12} - 10^{15}	10^{8} - 10^{9}	10^{8} -10^{11}	10^{14}-10^{15}	[Bou+17]
	-	-	-	10^{5}	10^{12}	10^{10}	2x10^{8}	10^{13}	[Chi+16]
	∞	∞	$< 10^{5}$	$10^{2} - 10^{6}$	$< 10^{15}$	$10^{6} - 10^{8}$	$< 10^{9}$	$< 10^{14}$	[And+17]
WR energy	∼fJ	∼10 fJ	∼100 pJ	∼10 fJ	∼0.1 pJ	∼10 pJ	∼0.1 pJ	-	[YC16]
	0.956 nJ *	∼0.1 nJ/b	-	0.1-1 nJ/b	2.997 nJ *	< 1 nJ/b	-	-	[Xue+11]
	1 - 10 uW/bit	100 - 200 nW/b	-	several W/page	0.09 pJ *	100 nJ	1 nJ	-	[And+17]
Iset/Ireset	-	-	-	10 mA	94uA/51uA	300uA/600uA	200uA/20uA	100uA	[Chi+16]

STT-MRAM is generally regarded as one of the fastest emerging NVM, with timings that are close to those of *DRAM*, and in certain cases approaching those of *SRAM*. Magnetic memories also show a relatively high retention, with commercially available product promising 10 year retention at $85C$ [Eveb ; Evea]. Moreover, as reported in various studies (see Table 2.1), the expected endurance is also relatively high, ranging from 10^{12} to 10^{15} cycles. When looking at the most optimistic figure, this puts *STT-MRAM* close to volatile memories, while the more conservative number is still 2 to 3 orders of magnitude better for the other emerging NVM technologies, and more than 7 orders of magnitude better than Flash memories. For these reasons, *STT-MRAM* could be a good candidate replacing *DRAM* as a non-volatile system memory, and could even find its uses as a substitute for *SRAM* in L3 or L4 caches. *STT-MRAM* however is not a good candidate, as it stands, for a storage memory, as the scaling and cost are not competitive with other NVM technologies, let alone with Flash memories. For intermittently-powered systems, while still not a definitive replacement for *SRAM* due to higher timings and energy, it is one of the most promising NVM technologies and definitely a good candidate.

On the other hand, *PCM* is one of the more mature NVM, it shows very good scaling, with cell area that can approach that of Flash memories (though not as good as multi-level Flash). While it is relatively mature, with early commercial chips that have been manufactured, it offers good scaling, and better timing with respect to Flash, it does not seem like the best option for an intermittently-powered computing device. In particular, the high write energy and lower endurance make it a worse choice with respect to other NVMs. Moreover, *PCM* is generally worse than *RRAM* which also shows a comparably small cell area, while showing better timings, energy and similar or better endurance characteristics.

RRAM is also a very promising family of emerging NVMs, which could end up competing or coexisting with *DRAM* as a system memory. However, its low endurance and relatively high write energy, makes it difficult to adopt as the main memory for intermittent computing.

FRAM on the other hand shows better endurance and it is already well proven and available on the market as embedded memory in MCUs [Ins21]. However, these products are fabricated on older nodes and, at the moment, scaling *FRAM* to more advanced nodes is very difficult [Bou+17]. New memory technologies based on ferroelectric materials might be able to solve the scaling problem[Fuj13 ; Che16 ; FX20], but are not at the moment as ready as the other NVMs summarized in this work.

Table 2.1 also shows how sometimes different studies present significant variations in the magnitude of these parameters. This is due to the advancement of the technologies over time, but also to the fact that common tools used for estimating the size energy and timing of these memories are not very accurate [Don+12]. This brief overview of some emerging NVMs shows how the research is advancing in the pursuit of the prefect non-volatile memory. Some of these memories might find their use in niche applications or in conjunction with existing memories. While these NVMs have yet to reach their full potential, and they still pose some significant challenges, especially when it comes to write energy and latency, these NVMs are ultimately enabling intermittent systems powered with harvested environmental energy.

Chapitre 3

FREEZER a Dedicated Backup and Restore Controller

3.1 Introduction

In the context of IoT, many applications cannot afford the presence of a battery because of size, weight and cost issues. The recent advancement in the Non-Volatile Memory (NVM) technologies is paving the way for Non-Volatile Computing Systems. These systems are able to sustain computations under unstable power, by quickly saving the state of the full system in a non-volatile fashion. Thus, Non-Volatile Processors (NVPs) may allow battery-less designs without suffering from frequent power losses inherent in energy harvesting scenarios.

In related work, both software- and hardware-level solutions were proposed to cope with the backup and restore problem. Software-based approaches are implemented on platforms that include both some SRAM and an addressable NVM used to store the backup, as the one presented in [Zwe+11]. Checkpoints are placed at compile time [RSF11]. Then, at run-time the supply voltage is checked and, if an imminent power failure is identified ($V_{dd} < V_{th}$), a backup of the stack and the registers is executed. In some works, backups are only executed when a power failure interrupt is triggered and the full volatile state (SRAM and registers) is copied to the NVM [Bal+15 ; Bal+16]. Other approaches do not take advantage of the volatile SRAM and exploit the NVM as the only system memory, backing-up only the registers in the event of a power outage [Jay+15 ; Cho+19]. Software-level solutions can be implemented on available hardware, but they normally come with a big overhead in terms of both backup time and energy.

Hardware solutions on the other hand usually implement fully Non-Volatile Processors (NVP). NVPs mostly make use of emerging NVM technologies to implement complex hybrid memory elements (nvFF and nvSRAM, non-volatile registers, and SRAM memory, respectively) that allow for very fast parallel backup and restore operations [Yu+11 ;

Wan+12 ; Liu+16 ; Wan+17 ; Su+17a ; Sak+14 ; Sen+16b]. However, introducing these hybrid memory elements is intrusive. Moreover, it usually comes with a significant area overhead and often results in increased delay and active power. Additional limitations on the amount of data that can be saved and restored in parallel is imposed by the peak current consumption required to drive all the NVM bit cells at the same time. To mitigate these problems, the use of distributed small non-volatile arrays, where groups of flip-flops are backed-up in sequence is proposed in [Bar+13]. An adaptive restore controller for configuring the parallelism of the nvSRAM restore operation, trading off peak current with restore speed is instead presented in [Liu+16].

The use of NVM enables persistence across power failures but it also introduces the problem of consistency for the data stored in the NVM [RL14].

To address the consistency issue and improve reliability of the system, a software framework that performs a copy-on-write of modified pages of the NVM in a shadow memory area is developed in [Cho+19]. The consistency problem can be also addressed via static analysis or with hardware techniques [LJ16]. In particular, hybrid nvFFs can be used in a hardware scheme where an enhanced store buffer is used to treat the execution of stores to the NVM as speculative, until a checkpoint is reached [LJ16]. Two counters are also used to periodically trigger checkpoints based on the number of executed stores or on the number of executed instructions. Previous work has also focused the attention to the problem of optimal checkpoint placement, as in [GGK17] where online decisions on checkpoints are taken based on a table filled offline using Q-learning.

In this chapter, we propose Freezer, a hardware backup and restore controller that is able to reduce the amount of data that needs to be backed-up. Our approach avoids the high cost of hardware fully NVP architectures since it can be implemented with plain CMOS technology. Furthermore, contrary to other hardware based approaches such as non-volatile processors [Liu+16 ; Sak+14 ; Ma+15], our proposed controller is a component that can be integrated in existing SoCs, without requiring modification of the processor architecture. Moreover, Freezer achieves better performance than pure software approaches. Our contributions can be summarized as follows :

— We propose an analysis of different backup strategies based on the use of memory access traces.
— We introduce an oracle based backup strategy that provides the optimal lower bound for the backup size.
— We present a hardware backup controller, Freezer, that dynamically keeps track of

the changes in the program state and commits these changes in the NVM before the power failure. The controller spies the address signal of the SRAM and uses dirty bits to track modified addresses with a block granularity.

— We conduct an analysis of the trade-offs and a design space exploration for our proposed strategy. Results on a set of benchmarks show an average 8× reduction in backup size. Thanks to Freezer, the backup time is further reduced by more than 100×, with a very low area and power overhead.

— We compare the memory access energy of three different system architectures : SRAM+NVM, NVM-only and cache+NVM, showing that NVM-only systems take on average 3.74× to 3.35× more energy than SRAM+NVM with full-memory backup and 6.19× to 4.22× more when compared to Freezer. Our strategy shows a clear advantage also when compared to cache+NVM architecture, requiring in average 7.8× and 5.9× less energy, with respectively RRAM and STTRAM as main memory.

The rest of the chapter is organized as follows. In Section 3.2, we present some background information and related works. In Section 3.3 we describe the main system models and architectures for a transiently powered device, and we present a model for evaluating the memory access energy of different system architectures. In Section 3.4, we introduce and discuss the model for the backup strategies. Section 3.5 explains how the memory access traces of the benchmarks are processed and analysed. Section 3.6 presents Freezer backup controller, its algorithm, and some area and power synthesis results. We report several comparison results of our study in Section 3.7. Finally, we briefly discuss our approach and draw the conclusions in Sections 3.8 and 3.9.

3.2 Background and Related Work

In this section, we briefly present the context around non-volatile processors and the problem of state retention in energy harvesting applications. We then present the motivation from which this chapter is derived.

In related work, both software- and hardware-level solutions were proposed to guarantee forward progress across unpredictable power failures. There are two main approaches to cope with the backup and restore problem :

— periodic check-pointing [RSF11] [Cho+19], and

— on-demand backup [Bal+15] [Bal+16] [Jay+15].

Periodic check-pointing systems try to guarantee forward progress by repeatedly executing some check-pointing tasks, interleaved with the computation. These check-points are usually placed by the compiler, according to some heuristic. At run-time, when a check-point is reached, the system decides if a backup should be executed. In [RSF11], for example, the supply voltage level is checked to determine whether there is enough energy or if a snapshot should be taken. After a power outage, the state will be rolled back to the last saved state and the execution will resume from the last check-point that was reached. This approach has the advantage that backup size can be optimised, as the location of each check-point is known in advance. In [Cho+19], checkpoints are instead taken based on the expiration of a timer, but only the registers are saved as the system uses only NVM as its main memory. To avoid consistency issues with NVM updates happening between a checkpoint and a power failure, the modified NVM pages are saved with a copy-on-write mechanism on a shadow memory area. These periodic check-pointing techniques also introduce overhead due to the execution of unnecessary checkpoints and backups, moreover they may lead to the re-execution of part of the code after the rollback.

On-demand backup tries to avoid the run-time overheads introduced with periodic check-pointing by waiting until a power failure is detected before executing the backup. The typical behavior of an on-demand backup system is depicted in Fig. 3.1, which shows

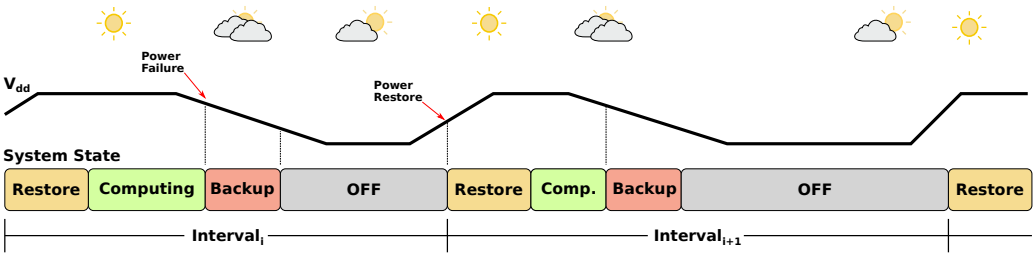


FIGURE 3.1 – Division of execution time in intervals and system state during an interval.

how the system responds to a power failure, signaled by a decrease in the supply voltage (Vdd), by interrupting the computation and by entering in the *Backup* phase. When the backup is completed, the system goes in the *OFF* state, where it will wait until the power resumes. When the power is newly available, the platform can leave the *OFF* state and start the recovery. The new interval begins when the system enters the *Restore* phase, to recover the state saved in the previous backup. When the restore is completed the system can resume the computation.

Some hardware-based solutions can also be considered as implementation of on-demand backups. As an example, in [Su+17b], the non-volatile processor is paired with a dedica-

ted voltage detector used to trigger the backup mechanism. The main disadvantage with these techniques is that they often require a full backup of the system memory, as it is difficult to know in advance when a power failure will happen and thus saving only the required memory is complicated.

To mitigate this problem some offline static analysis technique have been proposed [Zha+15 ; Zha+17]. In particular, in [Zha+17], an offline analysis of the code is used to find the backup positions that reduce the stack size. These positions are marked in the code with the insertion of special label instructions. At run-time a dedicated hardware module will wait for the power failure signal. After this signal, the execution continues until the program reaches the label instruction. Then, this dedicated hardware module executes the backup. These techniques require a compile-time analysis, with a detailed energy model of the platform. Moreover they tend to introduce overhead as they need to modify the program code [Zha+15] and the internal architecture of the processor [Zha+17].

Non-volatile processors can also be considered implementation of on-demand backup, as they focus on having very fast backup (and restore) in response to power failure. In [Ma+15], architectures and techniques for implementing non-pipelined, pipelined, and out-of-order (OoO) non-volatile processors are proposed. The proposed techniques try to optimise the backup size of the internal state of the processor, using techniques such as dirty bits for a selective backup of the register file. Contrary to our approach these architectures rely on NVM or hybrid memories for the persistence of the main memory. Moreover these techniques are in general very intrusive, as they require an in depth modification of the internal architecture of the processor.

To address the problem of full memory backup in an on-demand scheme, we propose a hardware backup controller, Freezer, that is able to optimise the size of the backup based on the information collected at run-time. Our proposed controller is an independent component that can be integrated in existing SoCs, without requiring changes to the internal architecture of the processor core.

In this work, we focus on how to optimise the backup of the main memory and we do not consider the problem of saving the internal state of the processor. However the state of the CPU could be managed via software by the processor by copying its internal register into the main memory before starting the back up. Other techniques are proposed in the literature to save the internal registers. Common hardware-based solutions use nvFFs based on different technologies, such as STTRAM [Sak+14], MRAM-based nvFFs [Sen+16b], FeRAM [Su+17b], ReRAM [Liu+16], and the use of FeRAM distribu-

ted mini arrays [Bar+13] or the use of nvFFs and NVM blocks for the backup of internal registers [Ma+15].

3.3 System Modelling

3.3.1 Considered System Model

Energy harvesting is seen as a promising source to power future battery-less IoT systems. However, due to the unpredictable nature of the energy source, these systems will be subject to sudden power outages. This could cause the execution of program to be unexpectedly interrupted. Thus, in these intermittently (or transiently) powered systems, the execution is divided in multiple power cycles, i.e., intervals, as shown in Fig. 3.1. The timing break-down of one of these intervals is depicted in Fig. 3.2. t_{cyc} is the duration of this on-off cycle and is defined as $t_{cyc} = t_r + t_a + t_s + t_{off}$, where t_a is the time in active state where the system is executing some software tasks, t_{off} the time in the power-off state, and t_r, t_s the time to restore, save (backup) the data from, to the NVM, respectively. The

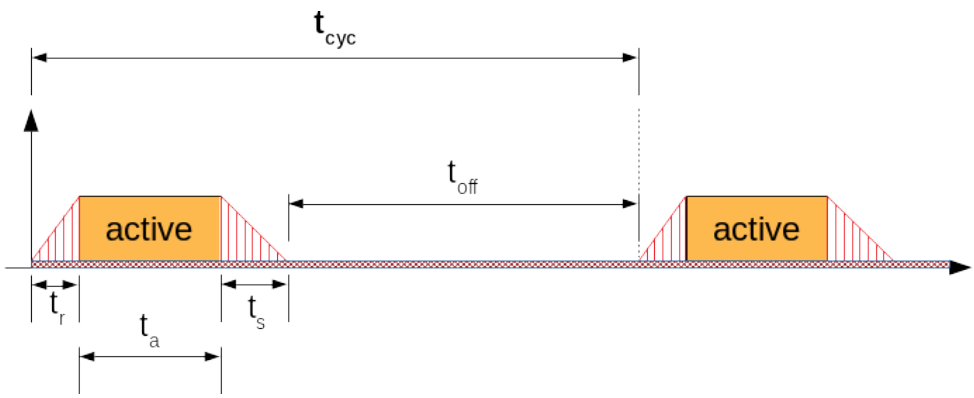


FIGURE 3.2 – Detail of an execution cycle between two consecutive power outages.

energy consumed by the system during t_{cyc} can be modelled as (adapted from [Hag+17])

$$E_c = E_s N_s + E_r N_r + P_{on} t_a + P_{off} t_{off}, \tag{3.1}$$

where E_s and E_r are the energy required respectively for saving and restoring one word, N_s and N_r the total number of words to save and restore. P_{on} and P_{off} are the power consumed during the active state and off state, respectively. In this type of intermittently-powered systems, usually P_{off} is zero as the state is retained in a non-volatile manner, thus

the all system including the processor core can be fully shut-down. Moreover, N_s and N_r are usually equal and often coincide with the full size of the volatile system state [Bal+15].

Considering an *on-demand backup* system that only performs a backup before a power failure, the total execution time t_{exec} of a program can be modeled as (adapted from [Bal+15])

$$t_{exec} = t_{prog} + n_i \times (t_s + t_r + \overline{t_{off}}), \tag{3.2}$$

where t_{prog} is the time needed for running the whole program without interruptions, n_i the number of interruptions, t_s and t_r the save and restore time, respectively, and $\overline{t_{off}}$ the average off time.

Our approach, Freezer, aims at reducing the size of the backup (N_s), thus also reducing t_s and the total execution time and backup energy. Moreover, the hardware implementation of our approach guarantees an additional decrease to the backup and restore time and energy, by eliminating the overhead due to software operations.

In this chapter, we assume that the system has a reliable way to detect a power failure and we also assume that the system has enough power to complete the backup. Therefore we do not investigate the problem of how to deal with incomplete backup. To have a stronger guarantee on the consistency of the system state after recovery a double buffering scheme can be applied, such that a new backup does not overwrite the previous one on the NVM. Moreover, we do not deal with the issue of how to detect a power failure. For this problem there are also solutions proposed in the literature, such as dedicated voltage detector [Su+17b].

3.3.2 System Architecture

In the field of non-volatile processors for energy harvesting applications, there are several possible architectural choices for achieving state retention. The most common approaches are the following :

— A CPU with an SRAM and an addressable NVM. The NVM might serve as a backup of the full memory space of the SRAM but might also be addressable by the processor.

— A CPU with an SRAM and a backup-only NVM or a CPU with a hybrid nvSRAM as in [Liu+16][Su+17a].

— A CPU with an NVM as main system memory as in [Sak+14 ; Wan+17 ; Jay+15 ; Sen+16b ; Cho+19].

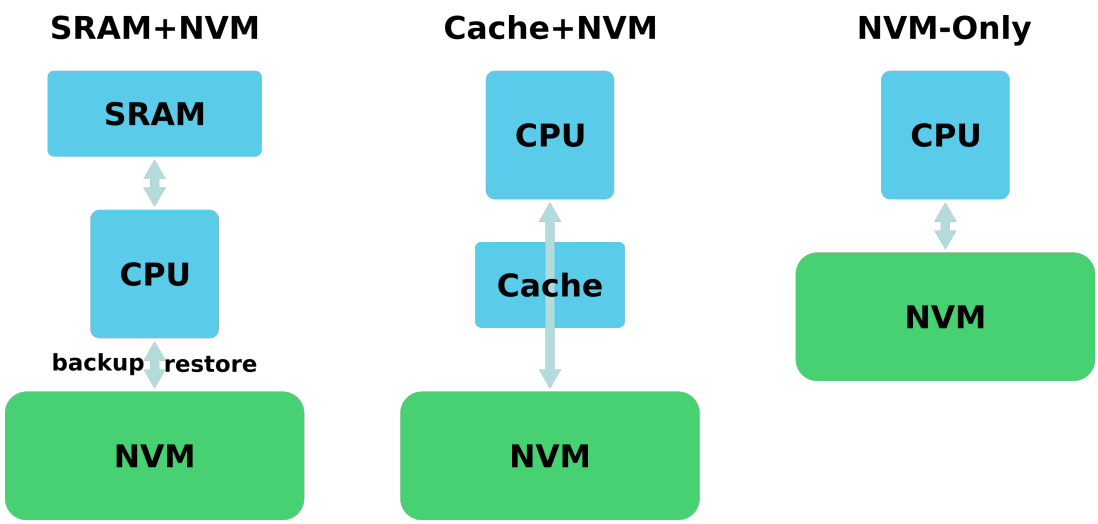


FIGURE 3.3 – Architectural models for non-volatile state retention.

— A CPU with an SRAM-based cache and an NVM as the only system memory [GGK17 ; Ma+15].

These approaches can be grouped into the three basic architectures depicted in Fig. 3.3. The first two approaches have in common the SRAM+NVM architecture, which, as shown in Fig. 3.3, exploits SRAM for execution and NVM for enabling backup and restore operations. The NVM-only approach relies solely on NVM as its main memory. Cache+NVM uses NVM as the main memory with the addition of a volatile cache.

A common choice for implementing intermittently-powered systems is to use commercially available SoCs with an embedded addressable NVM. As this NVM is addressable, this type of systems is the common choice for implementing software-based retention schemes [Bal+15 ; RSF11]. Another option explored in related work is that of using hybrid nvSRAM [Yu+11 ; Liu+16 ; Su+17a]. This choice allows to exploit the main advantages of SRAM (fast read/write and low access power), while also obtaining fast parallel backup through the paired non-volatile memory elements. This means that the non-volatile elements are not directly accessible by the programmer, instead the non-volatility is made transparent by the hardware. A conceptually simple solution to guarantee state retention is to exploit only an NVM as the main memory. This solution is proposed in [Sak+14], where the system is fully based on STTRAM. Another example is given by the software approach of QuickRecall [Jay+15], where the available SRAM is not used and the system runs only on the FeRAM. As with hybrid nvSRAM, the non-volatility is transparent to the programmer. Also, in this case, there is no need to copy the data in the event of a

power failure. In [Ma+15] methods for the backup and recovery of the internal state are proposed and compared considering non pipelined, pipelined, and out of order (OoO) processor architectures. These solutions can also be considered NVM-only type of systems, as they use NVM as their main memory, with the addition of hybrid or NVM caches in the case of the OoO processor.

Unfortunately, some of these new NVM technologies are still immature and often they do not provide the same level of performance in terms of access time and access energy as the SRAM [YC16]. Moreover, NVM-only designs must also face the issue of wear and the reduced endurance that characterises many of the emerging NVM technologies. To mitigate this problem, a possible solution could be to use register-based or SRAM-based store buffers. As an example, enhanced store buffers are proposed in [LJ16] to postpone the execution of NVM writes, treating store operations as speculative. Though the limited size of the store buffer still results in very frequent checkpoints and a large number of NVM writes. Another possible answer to mitigate NVM writes speed and endurance problem could be to use an SRAM-based cache to buffer the accesses to the main NVM. Although this type of architecture could be of some interest for higher performance systems, it is not very common in small IoT edge nodes. This is because adding a cache would significantly increase both the dynamic and static power consumption during the active period.

In this work, we consider an architecture that comprises a micro-controller with an SRAM as the main memory, and an NVM that is used by our proposed backup controller, Freezer, to save (and restore) the state of the system before (and after) a power failure. The general overview of such architecture is depicted in Fig. 3.4. The micro-controller we consider implements the RISC-V Instruction Set Architecture (ISA).

3.3.3 Modelling Memory Access Energy

The energy required for a backup operation is dominated by the data transfers between the SRAM and the NVM, and will be proportional to the backup size. This energy will mostly be determined by the write energy of the NVM, that can be even 100× that of the SRAM [YC16]. Our approach provides a reduction of the backup energy by decreasing the number of data transfers and by improving the speed of the process compared with a software based backup strategy.

In this section, we provide a simplified model to evaluate and compare the energy cost of some of the different system architectures introduced in Section 3.3.2. Results provided in Section 3.7 are based on this model. In particular, we derive the energy cost in terms

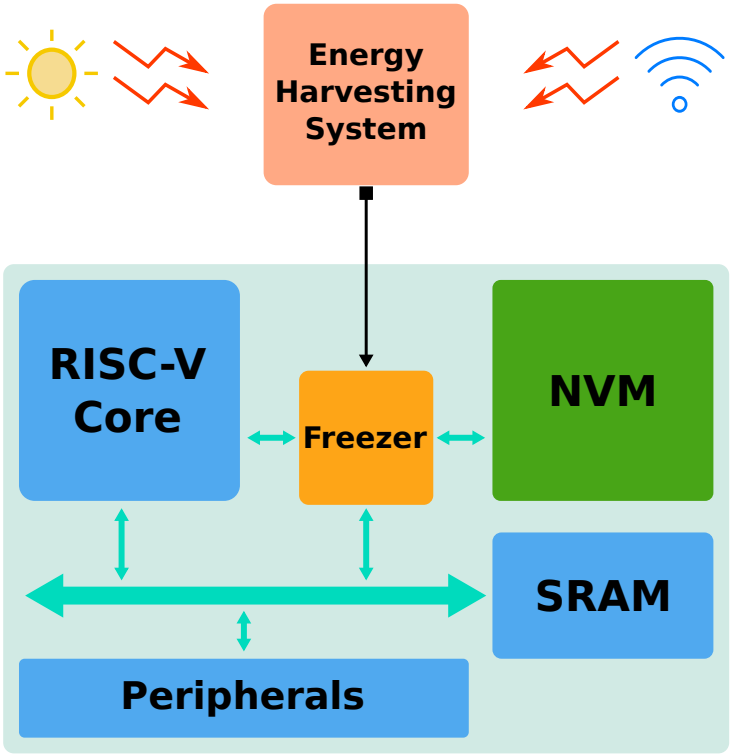


FIGURE 3.4 – General overview of a system implementing Freezer.

of memory accesses for the following types of memory models :

— SRAM + NVM for backup
— NVM only
— cache + NVM as main memory

For the *SRAM+NVM* architecture, we consider both a system which performs a full memory backup and a system with Freezer. For this system, the energy cost associated with memory accesses can be expressed as

$$E_{SRAM+NVM} = E_{prog} + E_{backup} + E_{restore} \tag{3.3}$$

where E_{prog} is the energy of the memory accesses needed for running the program.

$$E_{prog} = E_{sram/r} N_{load} + E_{sram/w} N_{store} \tag{3.4}$$

where $E_{sram/r}$ and $E_{sram/w}$ are the read and write energy of the SRAM, and N_{load} and N_{store} are the total number of load and store operations, respectively. The additional cost required by a platform with both SRAM and NVM are expressed in Eq. 3.3 by the energy

for the backup E_{backup} and by the energy for the restore $E_{restore}$, defined respectively as

$$E_{backup} = N_s(E_{sram/r} + E_{nvm/w}), \tag{3.5}$$

$$E_{restore} = N_r(E_{nvm/r} + E_{sram/w}). \tag{3.6}$$

The energy for the backup depends on the total size of the backup N_s and on the energy required for reading from SRAM $E_{sram/r}$ and writing to NVM $E_{nvm/w}$. N_s is the total number of saved words throughout the full execution. Similarly $E_{restore}$ can be expressed as the energy for a single transfer (read from NVM and write to SRAM) multiplied by the total number of restored words N_r.

For the *NVM-only* architecture there is no need to preform backup and restore operations, as everything is already saved in the NVM. In this case, the memory access energy is given only by the load and store operations performed for running the program. The energy cost for a purely non-volatile system that uses a NVM as its main memory is estimated by

$$E_{NVM} = E_{progNVM} = E_{nvm/r}N_{load} + E_{nvm/w}N_{store}. \tag{3.7}$$

The *cache+NVM* architecture comprises both an NVM as its main memory, and an SRAM-based cache to reduce the number of accesses to the NVM. This system uses a write-back cache controller that performs a flush of the dirty lines on NVM in case of a power failure. On a cache system, for every operation, the TAG memory is first read to verify if the required address is on the cache or not, then in case of a miss a read from NVM is executed. Moreover, simultaneous TAG and DATA memory reads are performed inside the cache to sustain high throughput. Finally, multiple data words may be accessed in parallel on N-way set-associative cache where only one word is useful. Therefore, the energy per read/write operation of this system is much higher that the one with tightly coupled memory (SRAM+NVM). The energy cost for a cache+NVM system is therefore estimated by

$$E_{cache} = E_{hits} + E_{misses} + E_{flushes} \tag{3.8}$$

where E_{hits} is the energy due to cache hits, E_{misses} the energy penalty due to misses, and $E_{flushes}$ the energy consumed with flushes. The first part of the energy cost E_{hits} is

$$E_{hits} = N_{hit/r}E_{hit} + N_{hit/w}(E_{hit} + E_{cache/w}) \tag{3.9}$$

where $N_{hit/r}$ and $N_{hit/w}$ are respectively the number of read and write hits, E_{hit} the energy for a single cache access and $E_{cache/w}$ the energy for a write operation inside the cache. E_{hits} therefore includes the energy due to read hits $N_{hit/r}E_{hit}$ and the energy due to write hits $N_{hit/w}(E_{hit} + E_{cache/w})$. E_{misses}, the energy due to the misses, is expressed as

$$\begin{aligned} E_{misses} = N_{miss}(E_{miss} + (E_{nvm/r} + E_{cache/w}) \times 8) \\ + N_{evict}E_{nvm/w} \end{aligned} \tag{3.10}$$

where N_{miss} is the total number of misses, E_{miss} the energy for a missing access, N_{evict} the total number of evicted words, $E_{nvm/r}$ and $E_{nvm/w}$ are the energy for reading and writing a word in the NVM. Eq. (3.10) shows that each miss causes the reading of a full block (8 words in our case) from the NVM. Moreover a missing access may also cause the eviction of a block from the cache resulting in writes to the NVM. $E_{flushes}$ is caused by the backup of the dirty lines before a power failure happens. This operation requires to scan all the cache lines and write back the dirty ones and is repeated before every power failure.

$$E_{flushes} = N_i N_{lines} E_{hit} + N_{flush} E_{nvm/w} \times 8 \tag{3.11}$$

where $N_{lines}E_{hit}$ represents the energy for reading all the blocks of the cache and N_i the number of interruptions. The energy due to the writes to NVM is expressed by N_{flush}, the total number of flushed blocks throughout all power failures, multiplied by the energy for writing 8 words to NVM.

3.4 Modeling of the Backup Strategies

By analyzing the memory access sequences, we can identify four main backup strategies. The *Full Memory Backup* strategy corresponds to the state of the art. In this chapter, we propose four backup strategies defined as *Used Address (UA)*, *Modified Address (MA)*, and *Modified Block (MB)*, a block-based evolution of the two previous strategies. The last strategy presented is an *Oracle* and cannot be implemented in a real system as it requires knowledge of the future. This oracle is however very useful for comparison, as it gives the optimal lower bound for the backup size. In the rest of the chapter, a *word* is defined as a 32-bit data.

3.4.1 Full Memory Backup

The first and simplest solution is to backup the full content of the memory at the end of each interval as it is proposed in [Bal+15]. For our study and fair comparison, we considered a slightly improved version of this strategy that saves only the data section of the program in pages of 512 bytes (128 words), thus not saving the full memory every time. As an example, if a program needs a 1000-byte data space, 1024 bytes (2 pages) will be saved in the NVM. With this approach, the backup size is a constant for all the intervals, equivalent to the number of pages to be saved.

3.4.2 Used Address Backup

The first strategy that we propose is the *Used Address (UA)* strategy. UA consists of keeping track of all the different addresses that are accessed (reading and writing) during an interval. When a power failure is detected every address that was accessed during that interval is saved in the NVM. In the UA case, only the memory locations that were used during the interval are going to be backed-up.

3.4.3 Modified Address Backup

If the initial snapshot of the program is stored in the NVM, the UA scheme can be improved by implementing the *Modified Address (MA)* backup strategy. MA only keeps track of the memory locations that are modified (written) during a power cycle. Then, before a power outage, only the words that were modified (write operation) are saved to the NVM. In practice this means saving only the addresses accessed by a store operation at least once during the interval. This number of addresses gives the size of the backup at the end of the interval. It may happen that the data written during execution do not modify the content of the memory. However, to keep the technique simple, we do not track the content of the memory but only the addresses where a write operation happens.

3.4.4 Oracle

The *Oracle* is defined as the strategy that saves only the words that are *alive*. An address is considered alive when it is going to be read at least once in any future interval. In other words, a written data is considered alive if it is read in any future intervals, before being modified by any other write at the same address. A word that will be overwritten

before being read is not considered alive and thus is not backed-up by the oracle. Fig. 3.5

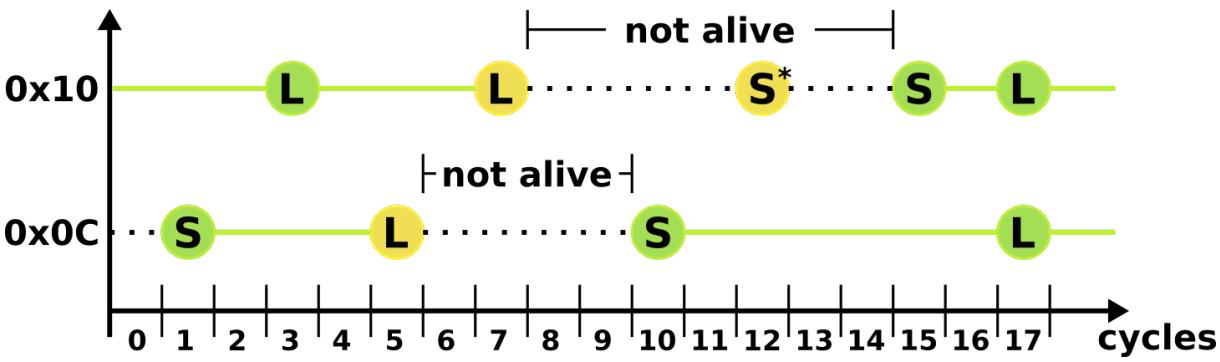


FIGURE 3.5 – Example of the aliveness of two addresses. L and S are respectively Load (L) and Store (S) instructions. The continuous green line indicates that the address is alive. The black dotted line is used when the address is not alive. The store on address *0x10* at cycle 12 (S*) does not make the address alive because it is followed by another store at cycle 15, that overwrites the value written by S*.

shows an example of two addresses changing between the *alive* and the *not alive* state as the execution progresses. In the example, address *0x0C* stops being alive after it is used by the load in cycle 5 and stays *not alive* for the period between the sixth and the ninth clock cycles. This happens because the Oracle knows that the value will be overwritten by the store executed at clock cycle 10. Therefore, between clock cycles 6 and 9, it does not consider *0x0C* as an alive address. For the same reason, address *0x10* stops being alive after the load in cycle 7 and is *not alive* in the time between cycle 8 and cycle 14. The store operation happening at cycle 12 does not change the state of the address because it is going to be followed by another store instruction that will discard this temporary update.

The *Oracle*, before the power failure, only saves the words that are going to be read during any further interval. Extending this oracle, we moreover define the *Oracle Modified (OM)* strategy that only saves the alive words that were modified in the current interval. As for the MA scheme, we can consider that a complete snapshot of the system memory is stored in the NVM at the beginning and during any previous interval. With the OM strategy, the data that will be read in the future are only saved if they were modified. If a data has been saved in the previous intervals and remained unchanged, it is not added in the snapshot of the memory to be saved before the next power failure. From now on, we will use Oracle to refer to the Oracle Modified when comparing with the other strategies.

3.4.5 Block-Based Strategies

Both the *Used Address* and *Modified Address* strategies can be implemented with different degrees of granularity. Tracking each individual word may require a very large memory to store the modified addresses, block-based strategy tries to trade-off between hardware cost and backup saving. Instead of considering single word addresses, the addresses can be grouped in blocks of N words and the scheme can be adapted to keep track of these blocks. Therefore the *Modified Block (MB)* strategy keeps track of the blocks that are modified during the interval. The backup size is given, for each interval, by the number of blocks that are accessed with one or more store operations. In Freezer, the modified blocks are tracked using corresponding *dirty bits*, which allows for the size of the associated tracking memory to be reduced by a factor equivalent to the block size. MB with blocks of $N = 1$ word corresponds to the MA strategy.

3.5 Trace Analysis and Improvement in Backup Size

In order to validate our approach, we analyzed the memory access traces of several benchmarks from a subset of MiBench (see Table 3.3 for a list of the benchmarks). The benchmarks were run on a cycle accurate, bit accurate RISC-V model [Rok+19], thus only two types of memory access are possible : load and store operations. The traces report the information about each memory access during the program execution. In particular, each trace records a timestamp (cycle count), the type of operation (ST or LD for store or load) and the address for every memory access. Table 3.1 shows an example of a memory access trace. The occurrences of power failures are simulated by dividing an access trace

Table 3.1 – Example of memory access trace.

interval	cycle	op	addr
i	...	...	...
i	90	ST	0x38aaad4
i	97	LD	0x2ba50
i	99	LD	0x2b06c
$i+1$	104	LD	0x2b954
$i+1$	109	LD	0x38aaad4
$i+1$	...	...	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	...	...	...

in n time intervals. Each interval i is composed of a given number of clock cycles N_{prog_i}, equal to the active time t_a of the interval i divided by the processor clock period. The cycle count reported in the trace is used to divide the execution of a benchmark in these n intervals. In the rest of the chapter, for simplicity without loosing generality, we divide t_{prog}, the time needed for running the whole program without interruptions, in n equal intervals of N_{prog} cycles.

In the example reported on Table 3.1, the interruption is placed after cycle *99*. This means that the load happening in cycle *104* is considered as being executed in the next interval $(i+1)$. This is a simple way to simulate a frequency of power failures every N_{prog} cycles ($N_{prog} = 100$ in this example). In practice, for our simulations we considered longer intervals, ranging from 10^5 to 10^7 cycles. As an example, considering a device running at 10 MHz, intervals of 10^6 clock cycles would correspond to a frequency of interruptions due to power failures of 10 Hz. In Section 3.7.2, we present an analysis of the impact of the interval length and of the variability of intervals duration on the reduction of backup size.

From these traces, the number of load and store operations per interval, as well as other memory access features, can be extracted. As an example, Fig. 3.6 shows the number of LD and ST in each interval during the execution of the FFT benchmark, with intervals of $N_{prog} = 10^7$ clock cycles. Considering the duration of the full execution of the FFT

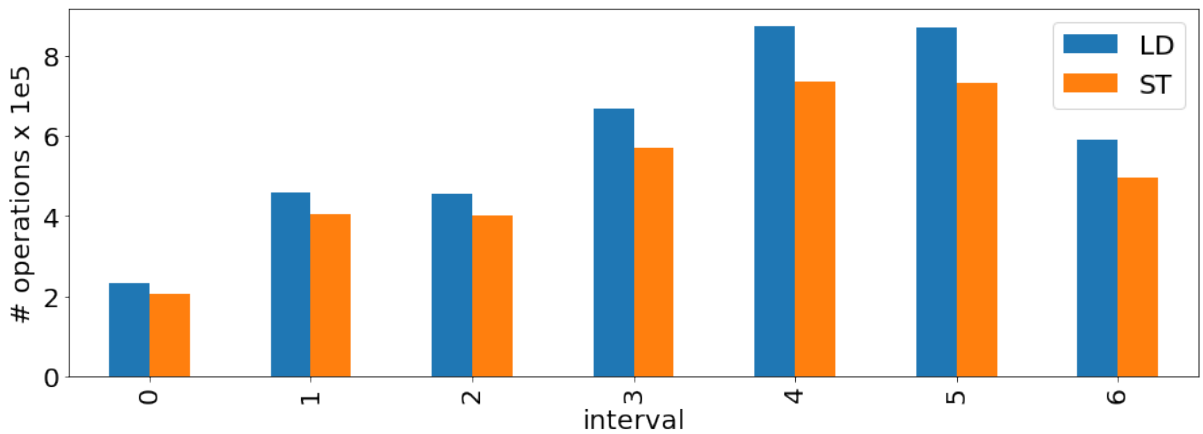


FIGURE 3.6 – Number of LD and ST operations per interval during the execution of the FFT benchmark with $N_{prog} = 10^7$ clock cycles.

benchmark on the target processor, $n = 7$ intervals can be simulated, ranging from interval 0 to interval 6 in the figure. These traces provide relevant information about the memory

access behavior of a given program. They will be used to compare the different backup strategies in Sections 3.6.3 and 3.7.

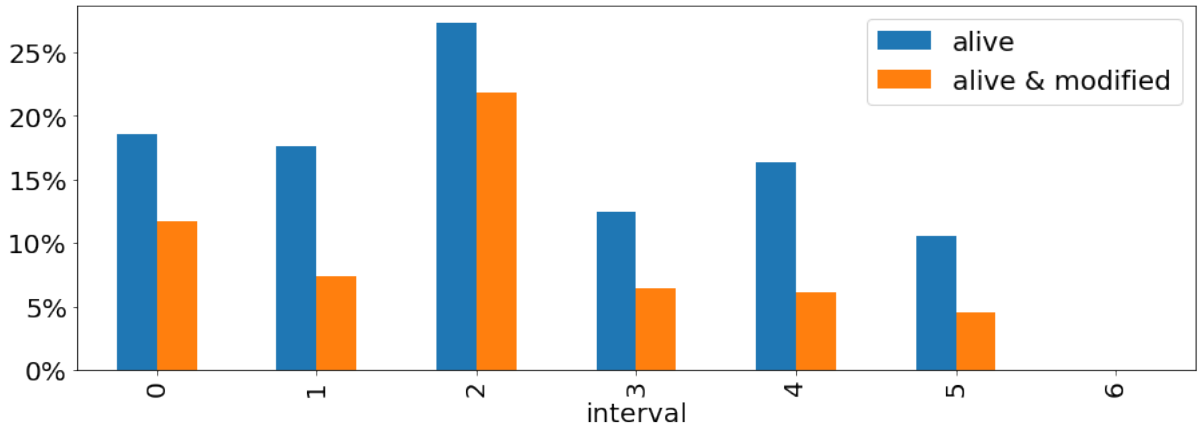


FIGURE 3.7 – Percentage w.r.t. full memory space of "alive" and "alive & modified" addresses per interval during the execution of the FFT benchmark with $N_{prog} = 10^7$ clock cycles.

Fig. 3.7 shows the fraction of *alive* and *alive & modified* addresses with respect to the total number of words addressed, for every interval of 10^7 clock cycles for the FFT benchmark. In the last interval no address is considered alive as the oracle knows that the program is going to terminate before the next power failure. The figure also shows that, even with a relatively small benchmark, the number of words that really needs to be saved is less than a quarter of the total. This motivates our work on the definition of new backup strategies to reduce the volume of data to be backed-up before a power failure. However, as already mentioned, the OM cannot be implemented in a real system as it requires knowledge of the future. It is however very useful for comparison as it gives the optimal lowest bound to the backup size.

Fig. 3.8 compares the average number of word saved per interval by the full-memory, UA, MA, and OM strategies for different benchmarks and with $N_{prog} = 10^6$ clock cycles. The figure shows the great potential of the proposed strategies w.r.t. state-of-the-art approaches. Fig. 3.8 also demonstrates that the MA strategy always outperforms the UA strategy in terms of number of saved words and it is the only technique that comes close to the performance of the oracle modified. Therefore, only the MA strategy will be considered in the rest of the chapter, as well as its extension to a block-based strategy presented in the following section.

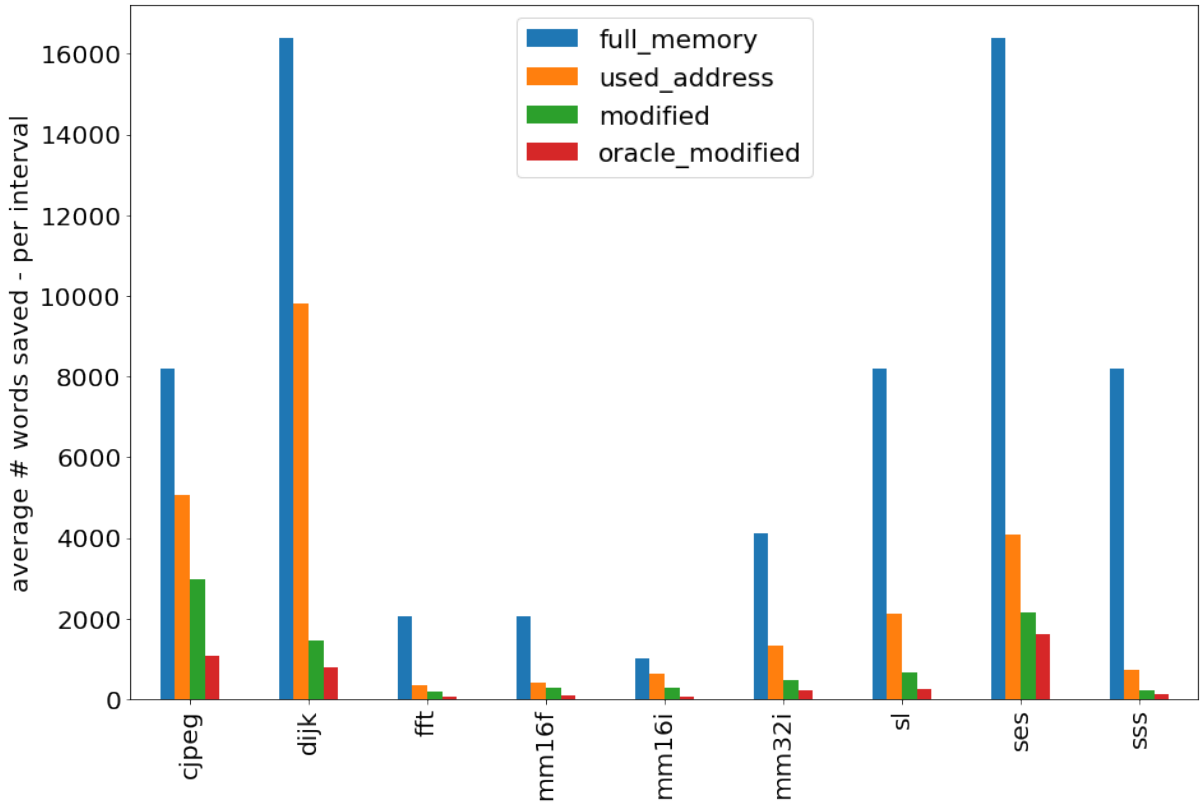


FIGURE 3.8 – Average number of words saved per interval by the different backup strategies – full-memory, used-address (UA), modified (MA), and oracle modified (OM) – during the execution of different benchmarks, with $N_{prog} = 10^6$ cycles.

3.6 Freezer

In this section, we present Freezer, a backup controller that implements the *Modified Block* backup strategy, and study the impacts of the block size in the MB strategy.

3.6.1 Freezer Architecture

Fig. 3.4 shows the system-level view of the *Freezer* architecture. The system is composed of four major components : the CPU, the SRAM used as a main memory, the NVM used for the backup, and the backup controller (Freezer). Freezer is itself composed of two main blocks : a controller implemented as a finite-state machine (FSM) for sequencing the operations and a small memory containing the dirty bits used to keep track of the blocks that need to be saved, as shown in Fig. 3.9. The Freezer controller is a stand-alone component, that does not need to be tightly coupled with the memories or with the core.

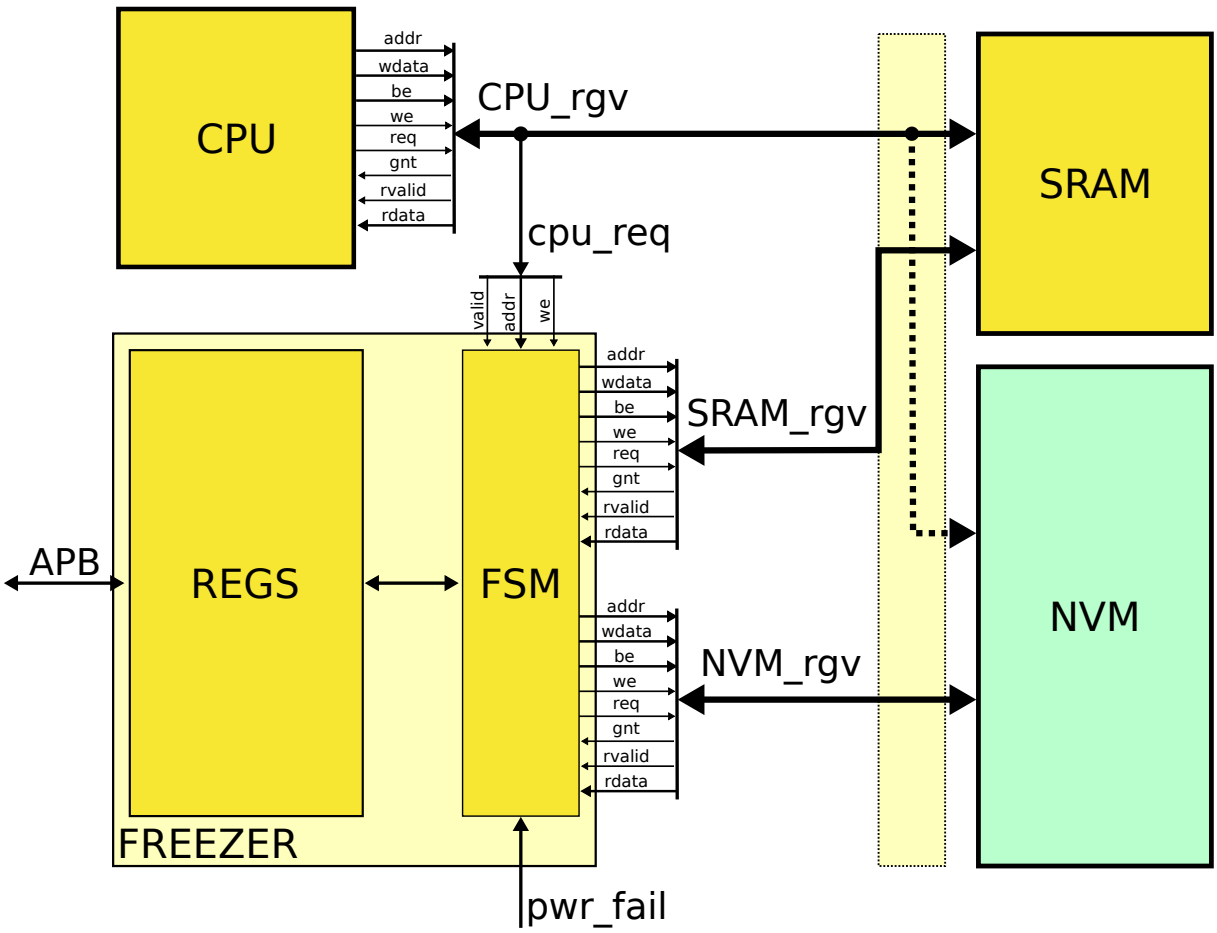


FIGURE 3.9 – Freezer internal architecture

It uses two handshake interfaces for the SRAM and NVM requests, allowing to tolerate variable access latency. Freezer can be directly connected to the control, address and data signals of both SRAM and NVM, using these handshake interfaces. Alternatively the SRAM and NVM interfaces can be arbitrated and share a single master port on the system bus. Moreover, Freezer is also connected to the request signals of the CPU to the SRAM, this allows Freezer to (i) spy the address of the SRAM accesses by the processor and (ii) manage the backup-to and restore-from-NVM phases in place of the processor. SRAM and NVM do not need to have two ports, CPU and Freezer accesses can be easily arbitrated as they never access the memory at the same time.

At run-time, Freezer checks the address of the store operations in the SRAM to dynamically keep track of the blocks that are modified. When a power failure arises, the CPU is halted and the controller starts transferring the modified blocks into the non-volatile

Algorithm 1: Freezer backup controller algorithm

```
Input: cpu_addr address generated by the CPU
Input: is_store = 1 if the operation is a store
Input: op_valid = 1 if the operation is valid
Input: pwr_fail = 1 if power failure is detected
Input: restore = 1 if resume after a power failure
Data: to_backup flag memory of 1-bit per block
if restore :
    for i ← 0 to SRAM_SIZE − 1:
        sram[i] ← nvm[i];
else :
    if not pwr_fail :
        if is_store and op_valid :
            block ← cpu_addr ≫ log2(BLOCK_SIZE);
            to_backup[block] ← 1;
    else :
        for b ← 0 to BLOCK_NUM − 1:
            if to_backup[b] :
                for a ← 0 to BLOCK_SIZE-1:
                    addr ← (b ≪ log2(BLOCK_SIZE))‖a;
                    nvm[addr] ← sram[addr];
```

memory. The words within a block are then stored sequentially in the NVM. The controller uses the information collected during the active time to determine which blocks to save. When performing this task, Freezer has access to both the SRAM and the NVM memory.

Algorithm 1 describes the behavior of the backup controller during the execution, backup, and restore phases. During execution, Freezer implements the *Modified Block* backup strategy. During execution, when there is no power failure (not *pwr_fail*) and there is a valid store operation, the controller records the blocks that are modified in a table (*to_backup*) implemented in a small memory, or in a register bank. When the *pwr_fail* condition is true, it enters in the backup phase and in a loop where, for each block, the *to_backup* memory is checked. If the block has to be saved, then a loop for every address of the block is executed, where a word is read from the SRAM and written in the NVM. This last loop can easily be pipe-lined such that an NVM write in an address can be executed in the same cycle with an SRAM read in the successive address. The same holds true also for the restore phase, that simply moves back the data from the

NVM to the SRAM. In this way, the backup controller is able to back up and restore one word every clock cycle. This should also lead to an additional speed-up, when compared with software-based backup loops executed on low-end micro-controllers, as in the case of [Bal+15].

In the hardware implementation, the process of checking the dirty bits can also be optimised. As an example, the scan of the *to_backup* memory to find the next dirty block can happen in parallel to the backup of the current block, which is a relatively long operation. Moreover, the *to_backup* memory can be organised as a matrix of dirty bits and the controller can check an entire row of dirty bits in parallel. This means that the *to_backup* memory can be scanned row by row. The sparsity of the dirty bits can also be exploited : skipping rows that have only clear bits (all zeros).

With these and other optimisations, the throughput of the backup operation can be sustained with little to no dead cycles. However these low level optimisations are outside the scope of this work and will not be investigated further.

3.6.2 Area and Power Results

As our algorithm is relatively simple, the controller itself introduces small area and power overheads. The major contribution in the area and power overheads is given by the *to_backup* dirty-bit memory, used to keep track of the blocks that have to be saved. Table 3.2 shows the number of bits and an estimation of the area of the *to_backup* memory for different block sizes, considering a 32KB SRAM. For these results, the *to_backup* memory

Table 3.2 – Number of bits and area estimation of the *to_backup* memory, implemented with standard cells in 28 nm FDSOI.

block size (32bit words)	2	4	8	16	32	64
# bits	4096	2048	1024	512	256	128
area [μm^2]	10838.11	5452.02	2748.94	1436.81	730.64	386.95

is synthesized with standard cells in a 28nm FDSOI technology using Synopsys Design Compiler (DC). Even when considering a fine granularity for the block size, the dirty-bit memory is small compared to the total size of the SRAM memory. As an example, for a block size of 8 words, the required 1024-bit memory is 256× smaller than the main SRAM memory. Moreover, by tuning the block size with larger blocks, the *to_backup* memory can be stored in a register file with a small increase in the backup size.

A non optimized version of the controller was synthesized from a C++ specification using Mentor Graphics CatapultHLS and Synopsys (DC) with the same 28nm FDSOI technology at $0.7V$. In this configuration, Freezer's controller achieves a dynamic power of $P_{active} = 6.8\mu W$ and a leakage power of around $P_{leak} = 40nW$ at 25°. With the same technology, we estimate a leakage of roughly $600nW$ for a register-based *to_backup* memory of 1024 bits. These synthesis results will be exploited in Section 3.7.6 to estimate the energy of a system implementing Freezer.

3.6.3 Impact of Block Size

In this section, we study the impact of the block size on the size of the backup provided by the MB strategy. The size of the *to_backup* memory depends on two parameters : the number of 32-bit words in each block, which determines the granularity of the backup strategy, and the total size of the SRAM. Therefore, it is possible to trade off an increase in backup size with a smaller area overhead of the *to_backup* memory. In Table 3.3, the backup size across a set of benchmarks is reported for different configurations of block granularity. The backup size is averaged on all intervals and normalized with respect to a block of one word (MA strategy). The interval is set to $N_{prog} = 10^6$ clock cycles. Increasing the block size has obviously an impact on the performance of the MB strategy,

Table 3.3 – Backup size relative to blocks of one 32-bit word (MA approach) for different benchmarks. $N_{prog} = 10^6$ cycles. The table also reports the average on all benchmarks.

block size N	2	4	8	16	32	64
susan_smooth_small (sss)	1.01	1.04	1.09	1.17	1.32	1.62
susan_edge_small (ses)	1.03	1.10	1.22	1.35	1.54	1.68
matmul16_float (mm16f)	1.11	1.24	1.47	1.77	2.23	2.54
qsort (qsort)	1.01	1.03	1.07	1.28	1.58	1.70
fft (fft)	1.10	1.22	1.44	1.79	2.45	3.22
matmul32_int (mm32i)	1.03	1.08	1.17	1.28	1.47	1.75
str_search (str)	1.02	1.06	1.12	1.17	1.28	1.55
cjpeg (cjpeg)	1.01	1.03	1.06	1.10	1.18	1.32
dijkstra (dijk)	1.06	1.18	1.31	1.36	1.45	1.55
matmul16_int (mm16i)	1.07	1.19	1.38	1.69	2.16	2.76
susan_edge_large (sel)	1.08	1.17	1.30	1.46	1.59	1.72
average (avg)	1.05	1.12	1.24	1.40	1.66	1.95

i.e., the average size of the backup required at the end of each interval. However, MB with a relative small size of block (up to $N = 8$) only increases the backup size by 24% in

average, while this block-based strategy can decrease the size of the *to_backup* memory by a factor of 8×. More comparisons and impact of the interval size are reported in the next section.

3.7 Results

In this section, the details of the experimental setup are explained and the results regarding the backup size (Sec. 3.7.1) and backup time (Sec. 3.7.3) are reported. The impact of the interval size is discussed in Section 3.7.2. For all the other results provided in this section, the interval is set to $N_{prog} = 10^6$ clock cycles. Moreover, a discussion about power, energy, and area of our approach is presented in Sections 3.7.4 and 3.7.6, while considerations on the impact of leakage are presented in Section 3.7.5.

3.7.1 Backup Size

For every interval, the backup size is computed considering the different approaches described in Section 3.4. Fig. 3.10 shows the backup size reported for every benchmark and for blocks of 1, 8, and 64 32-bit words. Blocks of size equal to one word corresponds to the MA strategy. The *OM* strategy is also reported to provide the optimal (non reachable) value. The backup size is averaged on all intervals and normalized against the improved Hibernus [Bal+15] approach, which saves the full memory used by the program in pages of 512 bytes.

As it can be seen from Fig. 3.10, our approach greatly reduces the average backup size per interval, reaching an 87.7% (more than 8×) reduction in average, with only a 7.5% distance from the *oracle modified*, when configured with a granularity of 8 words per block. This reduction in backup size can be directly converted into an energy saving in number of write to the NVM during the backup phase.

3.7.2 Impact of Interval Size

On a system powered with intermittent ambient energy, the time length of the intervals is mostly determined by the energy source and by the energy budget of the platform. If the energy source is relatively stable, the length of the power cycle increases, and so does the amount of computation that the processor manages to complete during one interval. This means that more memory accesses will be performed, thus we can expect

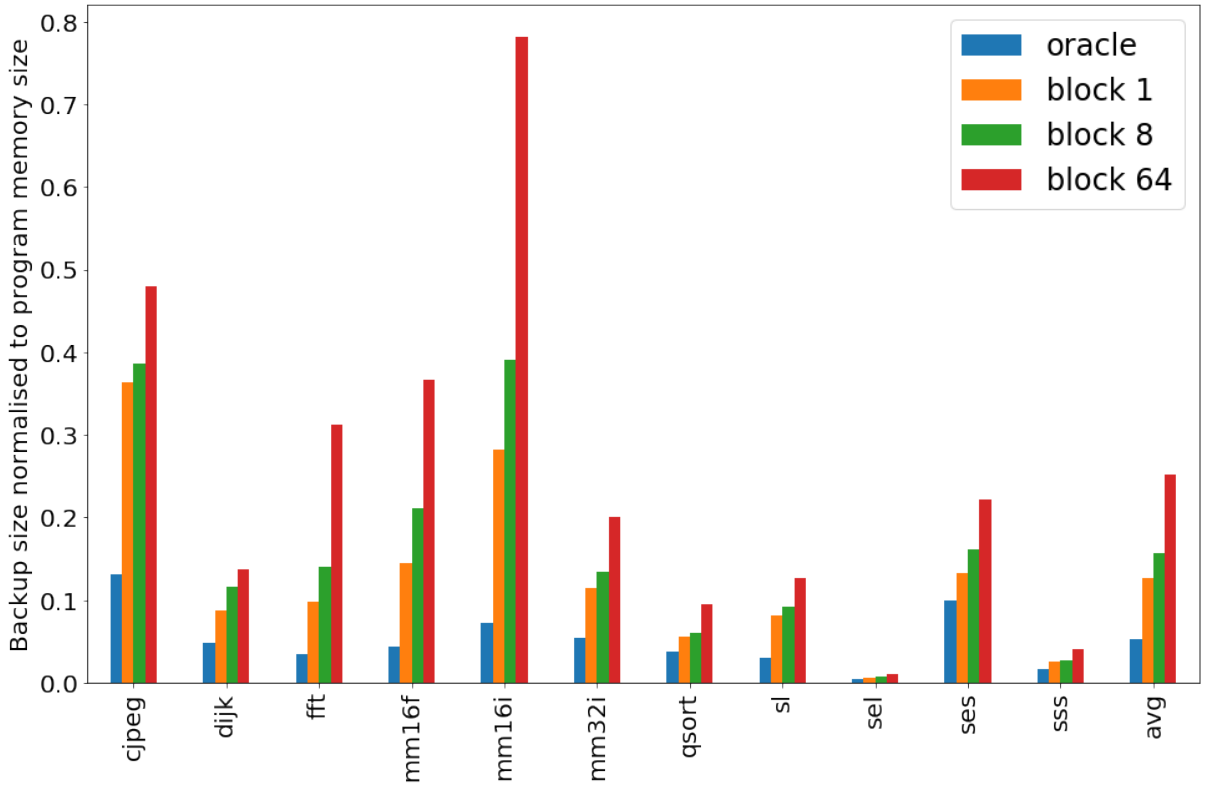


FIGURE 3.10 – Backup size normalized w.r.t. the program memory (improved Hibernus strategy) of Freezer implementing MB strategy with blocks of 1, 8, and 64 words. Lower bound in backup size of the oracle-modified is also reported.

the average size of a backup to increase. However, this also depends on the spatial locality of the application, and considering wider blocks could be beneficial for less intermittent sources. When the length of the power cycles decreases, the processor is interrupted more frequently, and the number of memory accesses is reduced. Therefore, the average backup size is further decreased. Fig. 3.11 shows the average reduction in backup size, across all benchmarks, for different lengths of the power cycles (interval size N_{prog} expressed in number of clock cycles), considering blocks of 8 words. As it can be seen, the backup size reduction is greater than 70% for all the interval lengths. Moreover, with shorter intervals, the reduction becomes greater than 90%. It must be noted also that, when the length of the interval is increased above 20 million clock cycles, the majority of the programs are able to run to completion before the first power failure occurs.

Due to the unpredictability of the energy source, an intermittently-powered system might also experience a wide variation between the time length of successive intervals. To

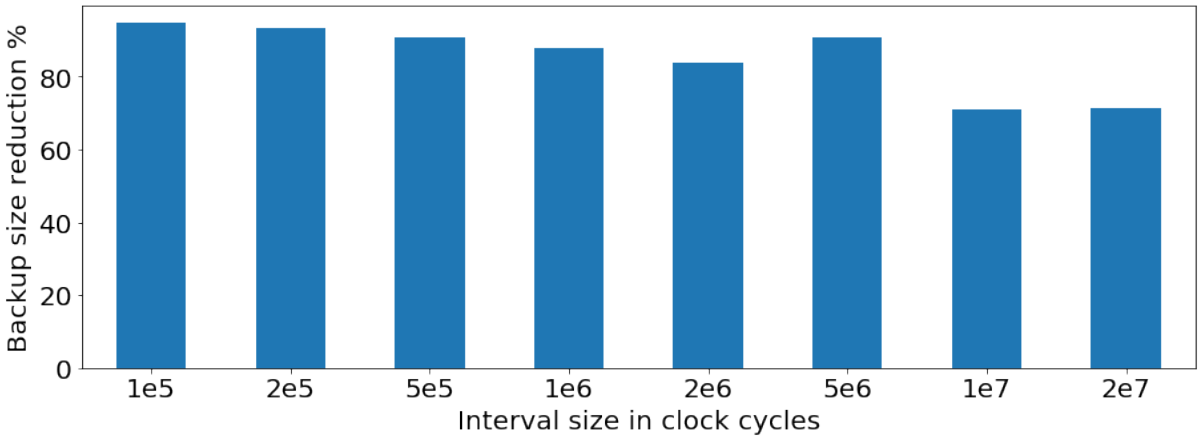
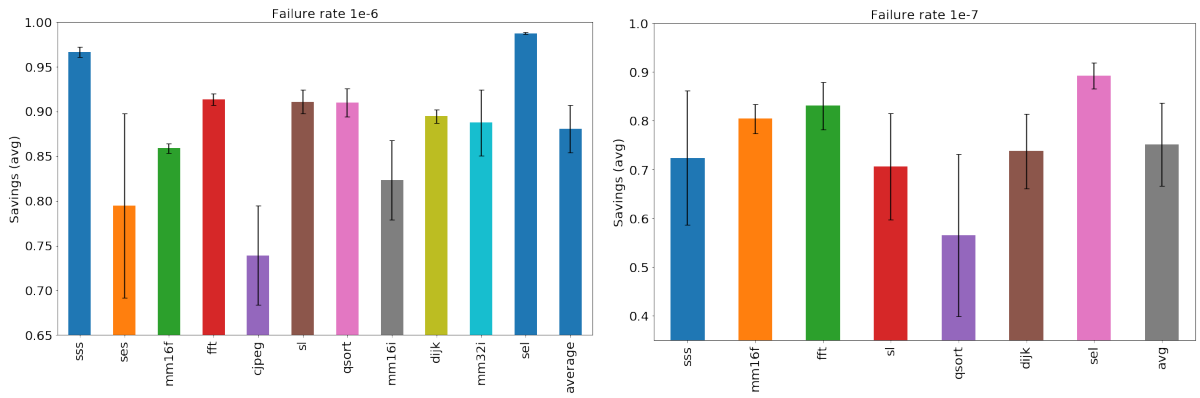


FIGURE 3.11 – Average backup size reduction with different interval size N_{prog} expressed in number of clock cycles.

better capture this behaviour, we model the occurrence of a power failure as a random variable distributed according to a binomial law. Power failure events in this model are considered independent of one another. At each clock cycle, there is a certain probability to incur in a power failure. For this experiment, we considered two values of one power failure every 10^6 cycles and one power failure every 10^7 clock cycles. Figures 3.12a and 3.12b show, for each benchmark, the average savings with relative standard deviation computed for 100 executions, considering blocks of 8 words. Our proposed method is robust to variability in the size of the intervals and it is able to achieve more than 83% and 88% savings on average when the failure rates are respectively 10^{-7} and 10^{-6}.



(a) (b)

FIGURE 3.12 – Average savings an std. deviation for 100 executions with power failures distributed following the binomial law.

3.7.3 Backup Time

The reduction in the backup size comes with a relative reduction in the save time. On top of that, thanks to the hardware accelerated backup process, our solution provides an additional improvement in terms of backup time. In particular, the backup process is managed directly by Freezer and can be further pipelined, so that each word can be saved in one clock cycle. Of course, the speed of this process is limited by the cycle-time of the slowest NVM memory. As our approach does not rely on any specific NVM technology, we considered the numbers reported in [Bal+15] for our comparison. In particular, we considered a clock frequency of 24 MHz for the normal operation using SRAM, and a clock cycle period of 125 ns (8 MHz) for the FeRAM.

Table 3.4 – Percentage reduction of backup time w.r.t. improved Hibernus (higher is better). Columns b_N provides results with our strategy using blocks of N words. Oracle modified and NVP [Liu+16] are also provided for comparison.

	b_1	b_8	b_64	oracle	NVP [Liu+16]
susan_smooth_small	99.80	99.78	99.68	99.87	39.25
susan_edge_small	98.93	98.69	98.20	99.20	42.58
matmul16_float	99.32	99.00	98.27	99.79	39.08
qsort	99.38	99.34	98.95	99.58	45.96
fft	99.58	99.39	98.64	99.85	39.64
matmul32_int	99.35	99.23	98.86	99.69	40.44
str_search	99.49	99.43	99.21	99.81	43.03
cjpeg	98.10	97.98	97.48	99.32	38.97
dijkstra	99.34	99.13	98.98	99.63	43.32
matmul16_int	99.23	98.94	97.88	99.80	29.94
susan_edge_large	99.93	99.91	99.89	99.95	45.78
average	99.31	99.17	98.73	99.68	40.73

Table 3.4 reports the improvement in backup time compared with a modified implementation of Hibernus that only saves the memory used by the program (in pages of 512 KB). Columns b_N provides results with our strategy using blocks of N 32-bit words. For the column related to non-volatile processor (NVP), we considered the backup time reported in [Liu+16] of 1.02 ms for 4KB, and scaled it for the memory size of our benchmarks, grouping the addresses in pages of 1 KB. With this configuration, our approach gives a two orders of magnitude improvement in backup time when compared to the software-based approach that saves the whole program memory. Moreover, Freezer provides a significant advantage also when compared with a fully non-volatile processor as [Liu+16], which only

provides an improvement of 40% when compared to the software-based approach.

This improvement in the backup time is also going to affect positively the total execution time, as expressed in Eq. 3.2. We considered a 24 MHz frequency for the volatile operations and an 8 MHz frequency for the FeRAM accesses. The active time is set to $N_{prog} = 10^7$ clock cycles at 24 MHz. We assumed an average off time equal to the active time. As reported in Table 3.5, our strategy achieves a 32% average decrease of the total execution time when compared with improved Hibernus. We also compared Freezer against approaches like QuickRecall [Jay+15] that runs the programs only using the NVM. In this case, the save and restore time are roughly zero (only the registers need to be saved), but the frequency of the core is limited to the frequency of the FeRAM. As a consequence, in most cases, the QuickRecall approach leads to longer execution time than Hibernus, whereas our solution performs always better and is very close to the Oracle.

Table 3.5 – Percentage reduction of execution time w.r.t. improved Hibernus (higher is better). Columns b_N provides results with our strategy using blocks of N words. Oracle and NVM-only solution of [Jay+15] are also provided for comparison.

	b_1	b_8	oracle	NVM only [Jay+15]
susan_smooth_small	20.75	20.75	20.76	-17.59
susan_edge_small	33.35	33.31	33.41	2.35
matmul16_float	9.90	9.88	9.93	-34.50
qsort	88.79	88.77	88.90	89.00
fft	10.68	10.67	10.70	-33.30
matmul32_int	15.32	15.31	15.35	-26.01
str_search	24.90	24.89	24.95	-11.05
cjpeg	27.93	27.91	28.13	-5.94
dijkstra	35.21	35.17	35.27	5.14
matmul16_int	8.72	8.71	8.75	-36.34
susan_edge_large	83.49	83.48	83.50	80.27
average	32.64	32.62	32.69	1.09

3.7.4 Energy Comparison with other Memory Models

We use Eqs. (3.3), (3.7), and (3.8) to compare the dynamic memory access energy of the different system configurations. Figures 3.13a and 3.13b show these dynamic energies normalised w.r.t. the system using Freezer, with RRAM and STT, respectively. For the cache+NVM architecture, four different cache sizes of 2KB, 4KB, 8KB and 16KB are reported. The three caches are all 4-way set associative with lines of 8 words (256 bits),

Table 3.6 – Energy and leakage power parameters used for memory access cost simulation. Read/write energy is reported in pJ per 32-bit word access.

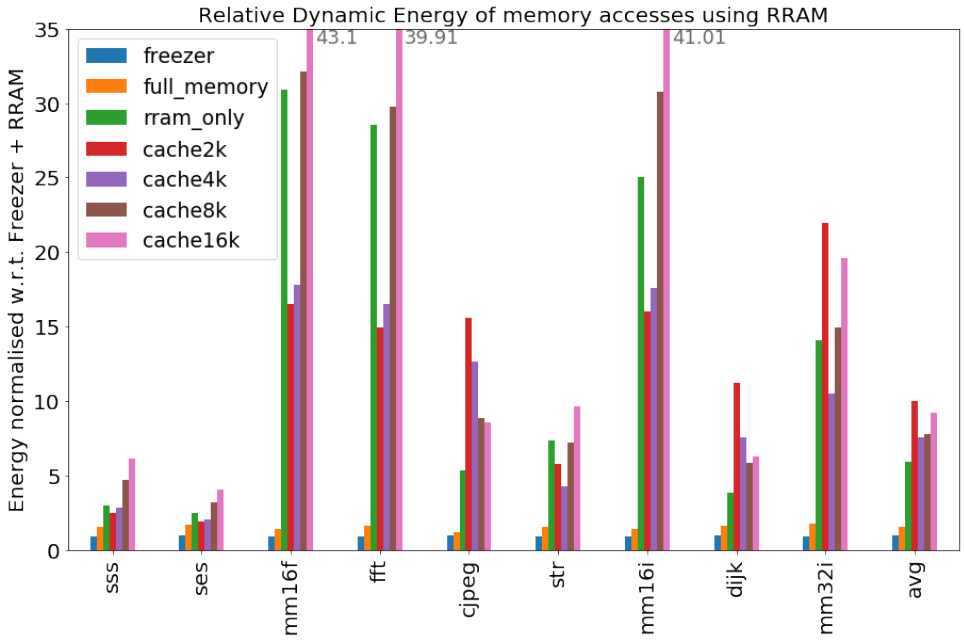
	SRAM				STT				RRAM		
Size [KB]	4	16	32	64	4	16	32	64	4	16	32
Read [pJ]	0.219	0.703	1.664	2.50	7.754	7.889	8.426	8.692	5.101	5.477	6.004
Write [pJ]	0.111	0.215	1.175	1.388	20.244	20.614	20.873	21.416	21.349	27.449	24.17
Leakage [μW]	0.78	2.16	3.58	7.16							

Table 3.7 – Cache Miss and Hit dynamic energy in pJ per 32-bit word access

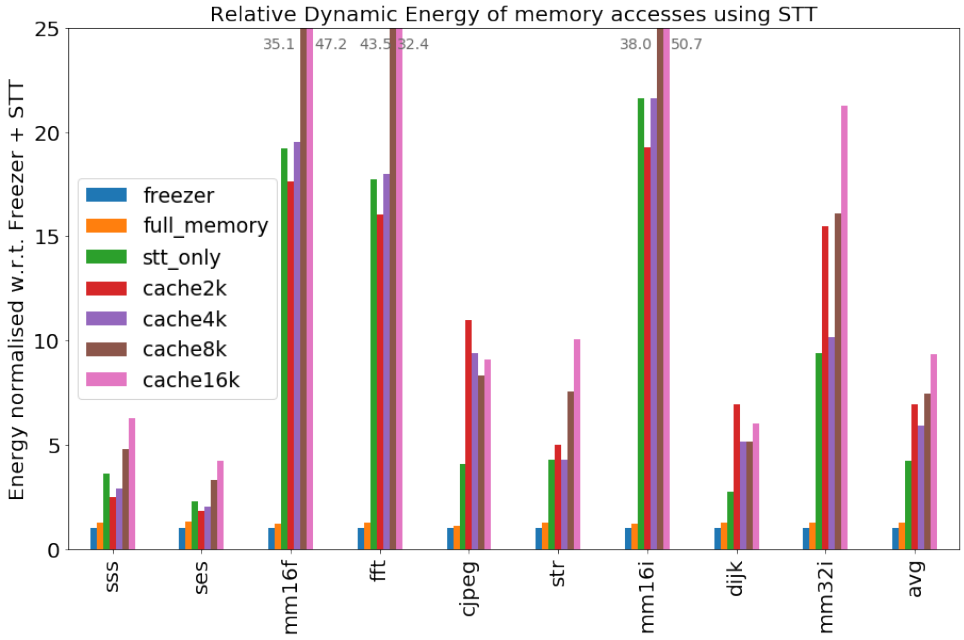
Size [KB]	Hit Energy [pJ] $E_{cache/r}$	Write Energy [pJ] $E_{cache/w}$
2	5.43	4.5
4	6.15	4.96
8	10.13	9.42
16	13.45	12.74

which is representative of this type of device. We considered blocks of 8 words also for the system using Freezer. The read and write dynamic energies per 32-bit word for the memories used in these comparison are reported in Table 3.6, and were obtained using NVSim [Don+12]. Table 3.7 reports the Hit and Write dynamic energy for the different cache sizes, obtained with NVSim. Miss energies were in all cases equal to Hit energies. As it can be seen from Fig. 3.13, our proposed approach provides a significant reduction in the energy due to memory accesses when compared with all the other methods. The memory access energy for a full-memory backup strategy is in average 1.26× of that required by Freezer when using STT and 1.65× when considering RRAM.

Being based on the same SRAM+NVM architecture, Freezer and full-memory backup strategies require the same absolute amount of energy for the execution of the program, i.e., the energy required for executing load and store operations is the same for the same benchmark. Moreover as the two strategies rely on a full memory restore after a power failure, they spend the same amount of energy for the restore memory accesses, for the same benchmark. Tables 3.8 and 3.9 show the energy decomposition, across all benchmarks, for Freezer and full-memory strategies when using STT NVM. The two tables show the clear advantage that Freezer brings in terms of backup energy, reducing its weight from an average 23.25% to an average 3.44% of the total memory access energy. Figures 3.13a and 3.13b also show that, due to the higher read and write dynamic energies, using the NVM as the main memory is often detrimental even when compared with full-memory ba-



(a)



(b)

FIGURE 3.13 – Memory access dynamic energy using RRAM (a) and STT (b) as NVMs for backup, with different system configurations.

Table 3.8 – Memory access energy percentage decomposition for Freezer using STT

Trace	backup	restore	prog. loads	prog. stores
sss	0.74	22.60	74.86	1.80
ses	5.97	29.30	59.23	5.49
mm16f	5.53	21.89	49.88	22.70
fft	4.72	28.28	46.12	20.88
cjpeg	7.53	16.30	57.64	18.53
str	2.62	23.86	43.65	29.87
mm16i	15.13	33.12	40.55	11.21
dijk	3.45	23.56	61.34	11.64
mm32i	4.46	26.86	60.42	8.26
avg	3.44	23.52	61.18	11.86

Table 3.9 – Memory access energy percentage decomposition for full-memory backup using STT

Trace	backup	restore	prog. loads	prog. stores
sss	21.39	17.90	59.29	1.43
ses	28.29	22.35	45.18	4.19
mm16f	21.66	18.15	41.36	18.82
fft	26.15	21.92	35.75	16.18
cjpeg	17.40	14.56	51.49	16.55
str	22.65	18.95	34.67	23.72
mm16i	31.33	26.80	32.81	9.07
dijk	23.60	18.65	48.54	9.21
mm32i	25.77	20.87	46.94	6.42
avg	23.25	18.69	48.63	9.42

ckup systems. Moreover when compared to Freezer, NVM-only systems require in average 6.19× and 4.22× more energy for RRAM and STT respectively.

As described in Section 3.3.3, the cache+NVM system uses the write-back policy and flushes the dirty lines in the NVM when a power failure arises. Thus the cache+NVM system shows a behaviour that is similar to the one of Freezer during power failures, but with higher energy per operation. There are however some major differences between a system that implements Freezer and a system with a write-back cache and a NVM main memory. First of all, Freezer is meant to be simple to reduce the energy overhead of tracking the modified blocks. Moreover, Freezer is able to track the full main memory and only needs to write on the NVM before a power failure happens. A write-back cache on the other hand might perform additional writes on the NVM at run-time. In fact if the

access causes a conflict, the cache will evict the conflicting line thus causing additional NVM writes. These additional writes may reduce the lifetime of the NVM due to the limited endurance of these type of memories.

When it comes to cache+NVM based systems, the size that in average provides the smallest energy is 4KB, with 2KB and 8KB caches performing better in some benchmarks. Access to smaller caches requires less energy, as shown in Table 3.7, but they might incur in the high cost of additional NVMs read and writes due to a larger number of misses and evictions. The increased number of writes to the NVM could also cause problems of endurance because of wear-out, that might prevent this solution to be applied for long-lasting operations. A larger cache can reduce the number of accesses to the NVM, up to the point where the cache is so large that it is able to buffer the full application. In these case, it is possible to obtain a number of writes to the NVM which is close to what Freezer achieves. However, this comes at the cost of having a large cache that is complex and energy hungry. Moreover, it is unusual to see a cache used in small low-power edge devices, where the system memory is embedded on chip and seldom exceeds 64KB. To summarize, for our set of benchmarks, the energy required by a 4KB cache + STT system is 5.9× w.r.t. Freezer, whereas the larger 16KB cache requires in average 9.3× more energy than Freezer.

3.7.5 Impact of Leakage Power

For a fair comparison, it is also important to study the impact of leakage power of the SRAM+NVM memory model, especially when compared to NVM-only architectures. Eq. 3.3 is therefore enhanced by considering the leakage power of low-power SRAMs of the appropriate size, as reported in Table 3.6. The leakage power of STT and RRAM is considered to be zero, which is obviously not the case for real designs. Table 3.10 reports for each benchmark the absolute dynamic energy of memory accesses for Freezer with both RRAM and STT as NVMs, equivalent to the Freezer blue bar in Figures 3.13a and 3.13b, respectively. The table also reports an estimation of the leakage energy due to the main SRAM memory obtained considering a $20MHz$ clock, and the total memory size of the benchmark. Table 3.10 shows that the leakage energy represents around half of the dynamic energy of memory accesses when using Freezer. Even accounting for the leakage of SRAM, the approaches based on SRAM+NVMs are still better than running an NVM-only system. Compared to full-memory backup which would consume roughly the same leakage energy, Freezer still benefits from the backup size reduction.

TABLE 3.10 – Backup energy using Freezer, leakage and memory size for different benchmarks, energy in [μJ], memory size in words of 32 bits.

Trace	mem_size [32-bit word]	E_freezer RRAM	E_freezer STT	E_leakage SRAM
sss	8192	11.0	11.0	6.1
ses	16384	3.3	3.2	1.9
mm16f	2048	2.5	2.4	2.0
fft	2048	3.4	3.4	3.7
cjpeg	8192	3.9	3.6	1.3
sl	8192	3.6	3.7	1.9
mm16i	1024	0.051	0.045	0.028
dijk	16384	51.0	50.0	27.0
mm32i	4096	0.58	0.56	0.41

Moreover, even accounting for the leakage of the NVM memories would not change the outcome of the analysis. In fact, when considering NVMs of the same size running for similar periods of time, the leakage due to the NVMs would be roughly the same for both SRAM+NVM and NVM-only architectures. Furthermore, an SRAM+NVM system would even be able to activate the NVM only during the backup and restore phases, reducing even more the impact of NVM leakage. In both cases, the SRAM+NVM architecture would still show an advantage.

3.7.6 Energy and Area Overhead Considerations

In this section, we provide insights about the overhead in energy due to our backup controller. The use of the Freezer hardware backup strategy in an energy harvesting platform will introduce a small overhead at run-time, but will also decrease the energy required for the backup and restore operations. We can account for the overhead and the reduction in the backup size by modifying Eq. 3.1 which becomes

$$E_c = E_s N'_s + E_r N_r + (P_{on} + P_{ovh}) \times t_{on} + P_{off} t_{off}, \tag{3.12}$$

where N'_s is the reduced backup size and P_{ovh} represents the overhead introduced at run-time. The energy required for moving the data (E_s for save and E_r for restore) is heavily dependent on the memory technology. However, software-based approaches introduces additional overhead. In our case, as a backup operation may require hundreds or even

thousands of transfers, we can approximate the energy required for saving one word as

$$E_s = E_{sram/r} + E_{nvm/w} \tag{3.13}$$

where $E_{sram/r}$ is the energy for reading a from the SRAM and $E_{nvm/w}$ the energy required for a write in the NVM.

The power overhead introduced by our strategy can be estimated as $P_{ovh} = \alpha \times P_{active} + P_{leak}$, where P_{leak} is the leakage power, which will be mostly determined by the *to_backup* memory, and P_{active} the active power. P_{leak} and P_{active} were provided in Section 3.6.2. P_{active} will be consumed whenever the processor performs a store operation and $\alpha = N_{store}/N_{prog}$ is the fraction of clock cycles spent performing store operations w.r.t. the execution of the program in the whole interval.

This overhead can be compared with the advantage gained in terms of save and restore energy. If we compare against a system that saves everything but does not introduce any overhead, we can estimate the maximum active time t_{on} after which the power consumed by the controller during active time becomes greater than the energy reduction obtained at backup time. t_{on} is constrained by the following inequation :

$$t_{on} \leq \frac{\delta E_s N_{tot}}{P_{ovh}} \tag{3.14}$$

where N_{tot} is the number of words to be backed-up without Freezer (full memory), E_s the energy required to back-up one word ($E_{sram/r} + E_{nvm/w}$), and $\delta E_s N_{tot}$ the energy saved during the backup operation. With Freezer, considering $\delta = 87.7\%$, $E_{sram/r} = 0.45pJ/bit$, $E_{nvm/w} = 100 \times E_{sram}$, we obtained for the two extreme configurations depending on the considered benchmark :

$t_{on} < 16.42s$ and $P_{ovh} = 1.18\mu W$ for *susan_smooth*, and

$t_{on} < 2.4s$ and a similar P_{ovh} for the *FFT* benchmark.

Both these t_{on} values allow for the programs to be executed completely and are well above the typical active time of intermittently-powered systems. Moreover, Eq. 3.14 is obtained by comparing our solution to a system that introduces no overhead at run-time and no overhead during the backup process, which would not be the case in real systems.

To give an idea of how Freezer would fit in a low-end IoT node, we can compare it with a ultra-low-power, size-optimised SoC, implemented with the same 28nm FDSOI technology node such as the one presented in [Bol+19]. In terms of area the SoC is

$0.7mm^2$, while its power consumption is 3μW/MHz giving at 48MHz a power consumption of 144μW. From these numbers we can see that Freezer, even with our non-optimised implementation, would lead to a small overhead. In particular, assuming blocks of 8 words, the area overhead of $2,748\mu m^2$ represents $\approx 0.4\%$. The power overhead during active time, considering the α of the FFT benchmark, could be as low as 0.82%.

3.8 Discussion About The Approach

Several studies have approached the problem of computing under intermittent power supply, providing a wide variety of different solutions. While software-based approaches try to solve the problem at the application level, hardware-based solutions try to provide platforms that implement the non-volatility in a way that is transparent to the programmer. The majority of the hardware solutions usually rely heavily on the underlying memory technology to accomplish the state retention. Even in [Hag+17], where no NVM is used, their technique relies on an ultra low-power retention SRAM.

Our approach moves away from this type of scheme and tries to solve the problem from a different standpoint, by providing hardware acceleration for the backup and restore procedures, and by exploiting run-time information to optimize the backup sequence. Moreover, this approach is agnostic with respect to the NVM technology, and opens a series of possibilities. Technologies such as hybrid nvSRAM, as the one used in [Liu+16], with circuit-level configurable memory, parallel block-wise backup and adaptive restore, may be exploited and enhanced by Freezer, thus achieving a faster and more energy efficient backup sequence thanks to the backup size reduction.

Furthermore, our approach could be extended to implement a programmable backup hardware accelerator, or to implement a dedicated ISA extension. This would provide programs with some levels of control on the save and restore procedures and allow for the hardware to exploit some of the information available to the program. As an example, a program may signal that a certain buffer or memory region is no longer used, allowing the controller to exclude it from the backup process. This would also make possible to integrate static analysis techniques such as the one presented in [Zha+15] and [Zha+17] on top of Freezer.

3.9 Conclusion

Applications that run under ambient harvested energy suffer from frequent and unpredictable power losses. To guarantee progress of computation in this circumstances, these applications have to rely on some mechanisms to retain their state. In this chapter, we propose Freezer, a backup and restore controller that is able to reduce the backup size by monitoring the memory accesses, and that provides hardware acceleration for the backup and restore procedures. The controller only requires a small memory to keep track of the store operations. Moreover, it can be implemented with plain CMOS technology and does not rely on complex and expensive hybrid non-volatile memory elements. Furthermore, Freezer is a drop-in component that can be integrated in existing SoCs without requiring modifications to the internal architecture of the processor. Our proposed solution achieve a 87.7% average reduction in backup size on a set of benchmarks, and a two orders of magnitude reduction in the backup time when compared with software based state-of-the-art approaches. Our analysis of memory access energy shows how architectures based on SRAM+NVM outperform cache+NVM and NVM-only architectures. In particular, our proposed solution requires on average 83.8% and 76.3% less energy for memory accesses than an NVM-only solution, and 87.2% and 83.0% less energy than a 4KB cache + NVM architecture, considering RRAM and STT, respectively.

Chapitre 4

Robust and Consistent Incremental Backup for Intermittently Powered Systems

Thanks to the advances in Non-Volatile Memory (NVM) technologies, environmental energy promises to be a viable solution to power future embedded and Internet of Things (IoT) devices. These devices can benefit from the persistence of emerging NVMs and use them to preserve the state of the computation across power failures. Such backups and restores of the state are critical operations whose interruption could jeopardize the forward progress or even lead to a corruption of the application state. To address this problem we propose two backup and restore algorithms that can, at each point, guarantee the existence of a consistent state of the system to revert to. Moreover, we show that thanks to this guarantee, we can further increase the efficiency of the system, allowing for a longer execution with the same value of capacitor, with both our proposed algorithms achieving more then 23% reduction in total energy and total execution time, when compared with a traditional *double-buffering* approach.

4.1 Introduction

Environmental energy harvesting is becoming a popular choice to power all those autonomous devices that, due to the long mission times, remote location or cost and size constraints, cannot afford to use a battery. Because of the intrinsic variability of environmental energy sources, these devices need mechanisms to tolerate sudden losses of power and to guarantee forward progress across power outages. In order to retain the state across power failures, these devices commonly seek to exploit emerging Non-Volatile Memory (NVM) technologies. In the literature, many solutions have been proposed to address the problem of forward progress under intermittent power. The most common

solution to this problem is to adopt some form of check-pointing strategy.

Check-pointing strategies can be grouped into two main categories : periodic check-pointing, and on-demand backup. In periodic check-pointing, the checkpoints are placed, usually at compile time, at predetermined locations during the execution, e.g., at the beginning of loops or function calls [RSF11]. In this chapter, we consider the more efficient scheme of on-demand backup, where a snapshot of the system is saved in NVM in response to a triggering event, such as the voltage falling below a threshold, which usually is an indication of an imminent power interruption. In both cases, when the energy becomes newly available, the system restores the volatile memory (SRAM) from the previous snapshot and resume the computation.

In the literature, many techniques have been proposed to carry out the backup and restore of the volatile system state. However the use of NVMs together with volatile memories in this context can give rise to inconsistencies. Two possible types of inconsistencies that may arise after a power failure can be identified : *NV-internal* and *NV-external* [RL14]. *NV-internal* inconsistencies may arise when data structures stored in NVM are partially updated before a power failure. As an example, a backup that is interrupted will result in an inconsistent state. On the other hand, *NV-external* inconsistencies can happen when the NVM is used to store run-time variables, as well as the backup data. In particular, they appear when, after a successful checkpoint, a variable residing in the NVM is updated, but a power failure happens before a new checkpoint is reached. The inconsistency arises because the volatile state is rolled-back to the previous checkpoint, but the variables in the NVM are not reverted to their previous state. While some works have been presented to address external inconsistencies, many of these techniques do not take internal inconsistencies into account, in particular those that may cause the corruption of the backup state.

These types of systems, usually rely on a storage capacitor to buffer the harvested energy, and to constitute an energy reserve to allow the backup operation, even when no energy can be harvested from the environment. The voltage across the storage capacitor is monitored, and when it falls below a given threshold, V_{backup}, a power failure is detected and the backup operation is triggered. Similarly when the capacitor voltage raises above a threshold, $V_{restore}$, the system can restore the state and resume the computation. An example of the evolution of the system state as a function of the capacitor voltage is depicted in Figure 4.3 and further detailed in Section 4.2.

In principle, a power failure could happen during the backup procedure, resulting in a

corrupted checkpoint. This event can result in a complete loss of progress, as in [Bal+15], and requires to restart the application. Many strategies presented in the literature assume that sufficient energy is available to carry out a full snapshot when the backup is triggered. This assumption of having enough energy for a complete backup, might be rendered untruth by some variability in the hardware platform, such as the value of the energy-storage capacitor, the backup/restore triggering threshold voltage levels or the leakage and power consumption of the device. This variability can be introduced due to factors such as manufacturing, aging, temperature or malfunctions. As an example, the value of the capacitor is usually defined within a tolerance range, with the tolerance for cheaper capacitors that can be as high as 20% of the nominal value. Moreover, during the lifetime of the device, the value of the capacitance might change due to aging or temperature variations. Aging and temperature can also affect the rest of the system, causing an increase in power consumption and leakage, that might drain the capacitor faster.

Failures during backup can cause the complete loss of progress and the corruption of the system state, if the backup strategy does not include any protection nor recovery mechanism. In order to maintain enough energy for a full backup, conservative approaches, as it is for some solutions proposed in the literature [PMS20], might require to oversize the buffer capacitor, or to set a higher threshold voltage for the backup. However, using a higher threshold voltage effectively reduces the available energy in the capacitor that can be used for computations. This means that the application will be interrupted more often, as when the backup threshold approaches the restore threshold, there is less time left for computations. Frequent interruptions will come with the additional overhead due to the time and energy spent for more backup and restore operations, and due to the wake-up energy and time required by the CPU to get back to the active state.

As a result, we can identify two different sources of overhead that can increase the total energy consumption. When V_{backup} is close to the minimum voltage allowed by the CPU (V_{fail}), the check-pointing process can fail due to the voltage dropping below V_{fail} before the backup is complete. These failures lead to significant energy overhead, as energy is wasted for re-execution of the lost progress. On the other hand, when V_{backup} approaches $V_{restore}$, more energy is available for backups, and less is available for execution. This reduces the probability of failure during backups, but it also introduces additional interruptions. The higher V_{backup} causes backup interruptions to be triggered more often than necessary, which wastes energy for unnecessary backup operations, as well as adding unnecessary wake-up (and restore) time and energy overhead.

These two opposite effects depend on where the value of V_{backup} sits between V_{fail} and $V_{restore}$ and are shown as a synthetic illustration of the phenomenon in Figure 4.1. The red line represents the energy overhead due to failures happening during check-pointing. These failures are more likely to occur when V_{backup} is close to V_{fail}, while they tend not to happen with larger values of V_{backup}. The green line in Figure 4.1 shows the effect of the increased number of interruptions as V_{backup} approaches $V_{restore}$. More interruptions due to higher V_{backup} increase the total energy since more and more frequent backup and restore phases occur. The combined effect of these two components is shown by the blue line in Figure 4.1, and creates a *U-shaped* curve on the total energy consumed by the intermittently-powered system.

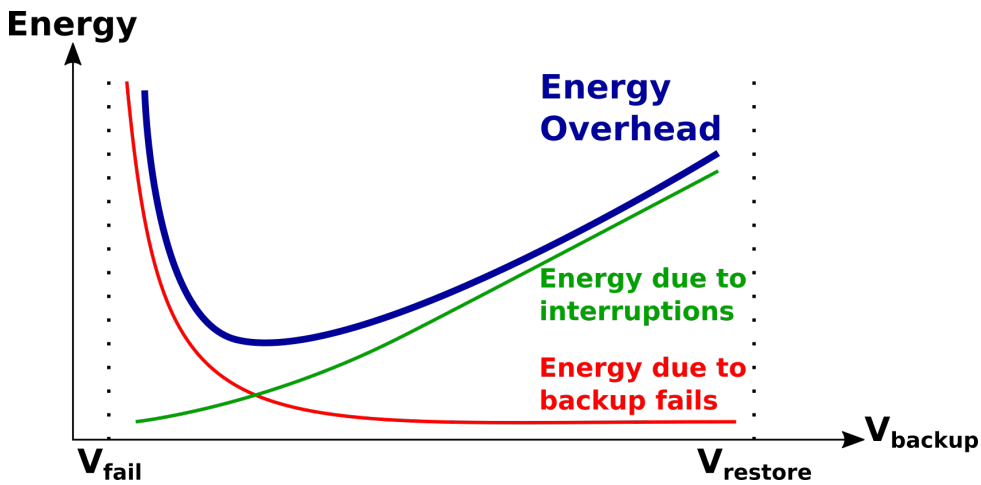


FIGURE 4.1 – Illustration of the total energy of an intermittently-powered system due to backup failures and interruptions.

Our proposed algorithm has the effect of moving the minimum point of the energy curve to the left, by reducing the overhead of failures and by reducing the probability of power failures happening. This enables lower V_{backup} to be used, which also lowers the energy overhead due to interruptions, by allowing more of the energy in the capacitor to be used for advancing computations between two power-off periods. As mentioned, reducing the value of V_{backup} will result in more energy available for the computations, without the need to increase the value of C_s.

In this chapter, we propose two *differential backup* algorithms that can always guarantee the existence of a consistent state to revert to, in case of a power failure during the backup process, without requiring to oversize the buffer capacitor. Moreover, we show that thanks to this guarantee, we can further increase the efficiency of the system, allowing

for a longer execution with the same value of capacitance. To evaluate our proposed algorithms, we moreover developed a model to estimate the probability of a backup failures. This chapter therefore presents the following contributions :

— We propose two backup and restore algorithms that are robust against failures during backup, and can revert to a consistent state in case of incomplete backup.
— We present a probabilistic model to evaluate the probability of incurring in a failure during backup.
— We evaluate our proposed algorithms with simulations driven by memory access traces issued from benchmarks and using our probabilistic failure model.
— We show that, thanks to our robust schemes, we can reduce the size of the capacitor and run with lower V_{backup}, which increases the time and energy available for computations.

The chapter is organized as follows. In Section 4.2, we describe the architectural model for the considered intermittently-powered system. Section 4.3 gives a brief overview of some related work. Section 4.4 motivates and details our two robust backup algorithms. In Section 4.5.2, we describe a model to evaluate the probability of incurring in a power failure during the backup. Section 4.6 presents the experimental setup and our simulation procedure, together with some results on the gain in memory and backup size, followed by a qualitative analysis of the overhead due to our approach. In Section 4.7, we compare the probability of failure and the system efficiency with other backup schemes before to conclude in the last section.

4.2 Architecture Model

The typical system architecture of an intermittent system powered through energy harvesting is depicted in Fig. 4.2. The device is powered with environmental energy extracted through an energy harvesting system. Examples of common sources of energy for this type of devices are vibrations, heat, ambient light and wireless signals. The harvested energy is buffered in the storage capacitor C_s to smooth out variation in the supplied power and to provide a minimal power reserve. Micro-controllers usually need a constant voltage to work properly. For this reason, the voltage across the capacitor is converted with a DC/DC converter to provide the SoC with the proper supply voltage. The voltage across the storage capacitor V_C is monitored by the power manager. When V_C falls below a certain threshold V_{backup}, a backup is triggered. When the voltage further falls below a

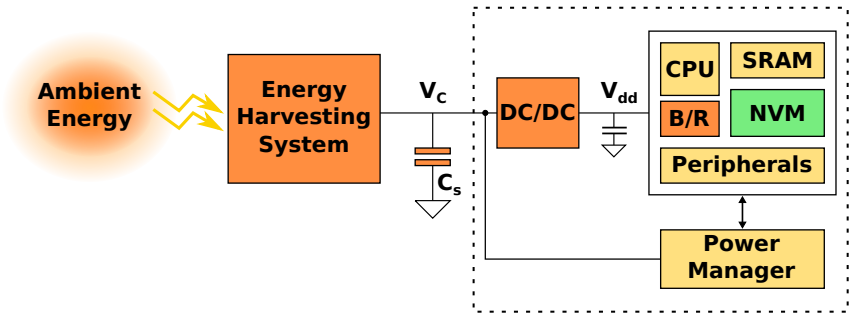


FIGURE 4.2 – System architecture of an intermittent system powered through energy harvesting

critical level V_{fail} outside of the range of the DC/DC converter, the processor can no longer function, thus a power failure happens and the system is powered-off. The power manager also triggers the restore operation when the voltage rises above a threshold $V_{restore}$, signifying that energy is newly available. Fig. 4.3 shows a simplified view of how the state of the system evolves depending on the voltage V_C. The possible states of the system can be *compute* (normal execution), *backup* (volatile data and state of the system are saved into the NVM), *restore* (state and data are restored from NVM to SRAM), and *off* (system is powered off), depending on the value of the voltage V_C in the storage capacitor. This voltage evolves over time according to the energy harvested from the environment, and the one consumed by the platform.

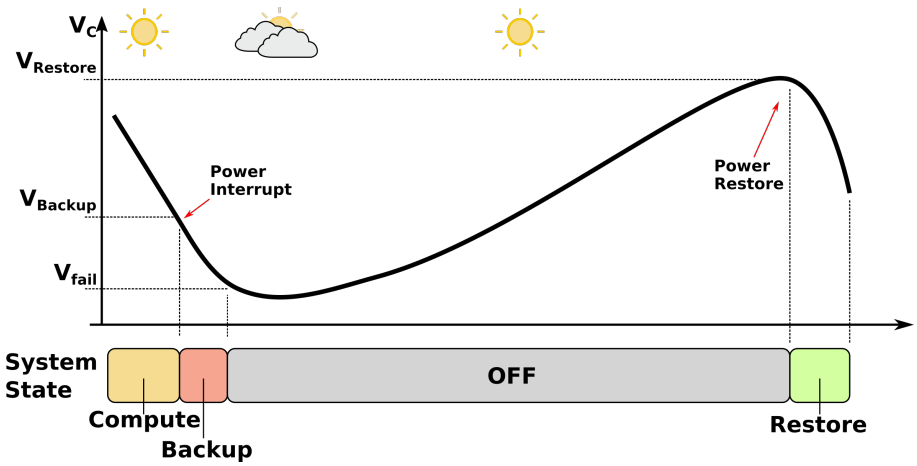


FIGURE 4.3 – Evolution of the system state as a function of the voltage across the capacitor.

The power manager does not need to be a dedicated hardware component, it could also be implemented as software running on the micro-controller. However, the platform

must be equipped with the circuitry for detecting power outages and trigger the backup and restore procedures. Such a circuitry could be based on dedicated voltage comparators [Wan+17], or use an analog-to-digital converter, which are common peripherals in commercially available devices [Ins21].

In this work we assume that the typical computing sub-system of Fig. 4.2, which comprises a power manager, a low-power micro-controller, a small amount of SRAM used as main working memory, and a non-volatile memory used for storing the backup. Moreover, we propose the introduction of an additional hardware component : a backup and restore controller that implements our proposed algorithms (the B/R orange block in Fig. 4.2). The main functions of this component are to track memory writes and to perform the memory transfers from SRAM to NVM for backups and from NVM to SRAM for the restores. To keep track of the modified blocks, the controller uses a small *bitmap* memory, in which every bit is uniquely mapped to a block in the SRAM. The bitmap is updated at run-time, similarly to what is proposed in [PMS20], or similarly to the dirty bits used in the cache lines of a cache memory. If a memory location within the block of data is updated, the corresponding bit in the *bitmap* will be set. As an example, a 16-bit bitmap could be used to map a 2KB memory, simultaneously tracking 16 blocks of 128 bytes, as shown in Fig. 4.4. Additionally, the proposed backup and restore operation

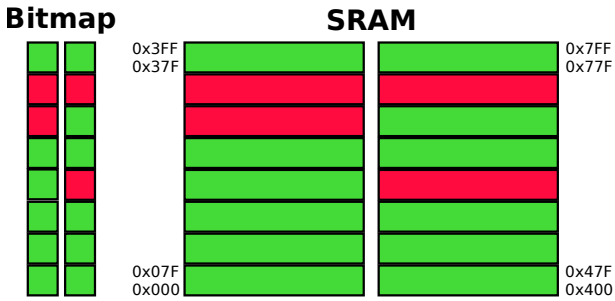


FIGURE 4.4 – Example of a 16-bit bitmap used to track writes in a 2KB memory

could be triggered by different events, other than the voltage crossing a threshold. As an example, an option could be to trigger the backup based on the expiration of a timer, changing the backup strategy from *On-Demand* (responding to imminent power failure), to a *Periodic Checkpointing* scheme.

4.3 Related Work

This section presents several works in the state of the art related to check-pointing for intermittently-powered systems. In the literature, many solutions and techniques have been proposed to tackle the problem of guaranteeing forward progress under intermittent power. We can categorize these approaches as either hardware or software based.

Hardware-based approaches exploit non-volatile and hybrid memories to implement fully non-volatile processors. Some techniques implement hybrid Flip-Flops based on Fe-RAM elements [Su+17b ; Wan+17 ; Liu+19]. Similarly, STT-MRAM and RRAM have been used to realize hybrid FFs in [Sak+14] and [Liu+16], respectively. In [Liu+16], even the main memory is hybrid, combining the classical CMOS-based SRAM design with RRAM cells. These techniques implement fully non-volatile processors that can achieve very fast wake-up and sleep time, thanks to the fast backup and restore operations brought by the hybrid memory elements. The drawbacks of these approaches are that these non-volatile processors are based on non-standard hybrid memory elements, which are commonly much larger than normal Flip-Flops, and can cause a very large area overhead. Moreover, while they have similar performance to their volatile counterparts, these devices still come with some speed and power overheads, due to the additional components required to access the non-volatile elements. To mitigate the area overhead and reduce the high peak current required for fully parallel backup and restore operations, Bartling *et al.* [Bar+13] proposed an NVP where the non volatile flip-flops are arranged in mini-arrays distributed throughout the processor. However, this design choice increases the backup and restore times with respect to other NVPs with fully parallel backup and restore operations.

Another commonly overlooked problem with these circuit/device-level solutions, is the fact that they do not offer any protection for the backup. In fact a single bit-flip in one of the non-volatile FFs could compromise the state of the application if no countermeasure is taken. Additionally, the emerging NVMs do not yet offer the same level of endurance as the SRAM. This can cause problems of reliability and expose the platform to wear-out attacks [CYL18a ; CYL18b].

On the other end of the spectrum, software-based solutions try to exploit the addressable NVM available in commercial devices [Ins21], to enable intermittent computations. In the software domain many techniques have been explored, from compile time checkpoint placement [RSF11], to on-demand full memory backup [Bal+15]. In [Cho+19], Choi

et al. present Elastine, a check-pointing scheme that exploits double buffering to backup the internal registers. Moreover they use a software-based page protection mechanism to track modified pages in the NVM during each interval and to save them on a shadow memory with a copy-on-write mechanism. After a failure, the check-pointed registers are restored and the NVM writes are discarded by restoring the content in the shadow memory. This avoids internal inconsistencies as both the NVM and the internal registers are restored to a consistent state.

The checkpoint stored in the NVM is also a vulnerable target in an intermittent system, that an attacker might want to corrupt to possibly cause unrecoverable errors. Cronin *et al.* proposed a *two-stage check-pointing scheme* to improve reliability and mitigate the threat from wear-out attacks [CYL18b]. The scheme uses checksum to detect errors in the checkpoint and keeping a copy of a previous checkpoint to revert to, in case of errors. They also propose to use *checkpoint scrambling* or *checkpoint encryption* to improve security, and they try to guarantee *checkpoint atomicity*. However, in order to guarantee *checkpoint atomicity*, their scheme still requires setting a high enough voltage threshold such that the platform has enough energy to carry out the full check-point of all the registers.

In [GGK17], a check-pointing scheme that uses Q-learning to compute off-line the optimal check-pointing strategy is presented. The approach considers the battery level, previous checkpoints and program forward progress. Moreover, the decision process is informed by the characteristics of the harvested energy, the processor power consumption and the check-pointing overhead. This scheme uses a shadow memory area in the NVM to store the snapshot of the registers and dirty cache lines. Besides, to avoid inconsistencies, the shadow memory is also used to checkpoint the modified blocks of NVM. This scheme uses a hardware scheduler to decide when to checkpoint and also requires up to 25KB of memory just to store the Q-table information.

All these schemes are based on forms of data replication, such as double buffering and shadow-memory [Cho+19 ; GGK17] or multiple check-pointing [CYL18b] to guarantee the consistency of the state. All these approaches use NVM as their system main memory, so that only the registers and the cache lines need to be check-pointed. However, this introduces run-time overhead, as the NVMs are slower and more energy hungry than traditional SRAM. In [PMS20], a differential backup scheme, based on saving only modified blocks of memory is proposed, the technique is implemented with a specialized hardware backup controller. A similar incremental backup approach is also proposed in [Ber+20], where instead of relying on a dedicated controller to perform the dirty memory tracking,

the Memory Protection Unit (MPU) available in many micro-controllers is used. The use of the MPU allows this solution to work on platforms without specialized hardware. However the backup granularity is limited by the number of regions supported by the MPU, moreover the use of the MPU to track dirty regions of memory introduces a run-time overhead due the interrupts generated by the MPU.

Both in [Ber+20] and [PMS20], the authors propose a form of incremental and differential backup. However, they do not provide protection against backup corruption. In this work, we propose two algorithms that can take advantage of the superior performance of SRAM, as demonstrated in [PMS20], while exploiting the NVM for guaranteeing a robust and consistent backup.

4.4 Robust and Consistent Backup

4.4.1 Full Backup with Double Buffering

The easiest way to avoid NVM internal inconsistencies due to partial or corrupted backups, is to use a double-buffering scheme to keep track of two different snapshots. This scheme extends the work in [Cho+19], which applies double-buffering to the register sate. To apply double-buffering to backup the main SRAM memory, the system needs enough space in the NVM to hold at least two full copies of the SRAM. The NVM is then divided into sections corresponding to the size of the SRAM. A pointer is used to point to the NVM section that holds the most recent backup. In the event of a power outage the SRAM is copied in the next available section of the NVM. This leaves untouched the previous snapshot of the system, which is still available in the preceding section of the NVM. In this way, when the snapshot is completed correctly, this section will become the most recent snapshot and it will be restored when the power resumes. Conversely, if the power fails before the backup is fully completed, the current backup is discarded and the previous snapshot will be restored. The NVM is used as a circular buffer, so that when the last section contains a valid snapshot, the next backup will be stored in the first section.

Double-buffering avoids overwriting the previous backup, thus guaranteeing that a valid snapshot is always available. This strategy however requires to have enough space on the NVM for two entire backups. Fortunately, this is rarely a problem, since it is common that the available NVM is several times larger than the available SRAM, in this kind of platforms. However, this scheme also comes with the same drawbacks as all the

other full-memory backup schemes. The backup process is in fact very expensive both in terms of time and energy, because it always copies the entire SRAM, even when there have been few changes. In order to improve this basic, naive backup and restore scheme, we propose two new algorithms that are shown to be more efficient in time, energy and resource usage, while also reducing the probability of a power failure.

4.4.2 Restore & Update Strategy

The first proposed algorithm follows the restore and update (RU) strategy. It divides the NVM into two sections : section **A** and **B**. Section **A** stores the last valid snapshot, which is a complete and correct state of the system. Section **B** is used during the backup process to store the block of SRAM memory that have been modified since the last safe restore. If the backup operation is suddenly interrupted by a power failure, the data in section **B** is incomplete. In this case, the system will restore the SRAM state from section **A**, which contains the previous fully consistent backup. On the other hand, if the backup is successfully executed, when the power resumes, the SRAM state is restored as follows : the last modified blocks are read from section **B**, while the blocks that remained unchanged are read from the previous snapshot stored in section **A**. Moreover, during the restore phase, the modified blocks read from section **B** are also written back to the section **A** of NVM to update the snapshot, thus creating, at the end of the process, a new complete valid state in section **A**. In the following, we describe in detail the steps of the algorithm during the *run-time*, *backup* and *restore* phases.

Run-time

During the run-time phase, while the CPU executes the program, the backup controller tracks the write accesses to the SRAM and marks the corresponding blocks as dirty, following Algorithm 2. Because the bitmap is managed and updated by the backup controller, there is no run-time overhead due to this operation on the CPU side.

Backup

When the voltage of the storage capacitor V_C reaches V_{backup}, the backup operation is triggered. During this phase, the bitmap pointing to the modified blocks is read. Each bit in the bitmap corresponds to a block in the main SRAM memory. All the blocks that are marked as modified (i.e., their corresponding bits are set to one), are copied to

Algorithm 2: Run-time block tracking of the RU strategy

Input: *cpu_addr* : address of the memory request from the CPU to the SRAM
Input: *store* : write enable signal for the SRAM, 1 for valid store operations
Data: $bitmap[Nblocks]$: array of dirty bits, one bit per block, size of $Nblocks$
Data: $BLOCK_SIZE$: size of a block in bytes

```
if store :
    block = cpu_addr ≫ log2(BLOCK_SIZE);
    bitmap[block] = 1;
```

section **B** of the NVM ($NVM.B$). Blocks that are not marked as dirty do not need to be saved, as they are already available in the previous snapshot saved in section **A** of the NVM ($NVM.A$). The bitmap containing the information of which blocks were modified, needs to be saved in the NVM as well. Therefore, when the transfer of modified blocks is complete, the bitmap is written in a dedicated area of the NVM ($NVM.bitmap$). To finish with the backup procedure, a *flag* value, signifying that the backup was successful, is written in a dedicated place of the NVM ($NVM.flag$). This flag will be checked at the beginning of the restore process to determine whether a valid backup was achieved. The backup procedure is formalized in Algorithm 3.

Algorithm 3: Backup algorithm of the RU strategy

Input: $SRAM$: SRAM memory
Output: $NVM.B$: section **B** of the NVM
Output: $NVM.bitmap$: section of the NVM to save *bitmap*
Output: $NVM.flag$: flag saved in the NVM to indicate a successful backup
Data: $bitmap[Nblocks], BLOCK_SIZE$

```
for i ← 0 to Nblocks − 1:
    if bitmap[i] :
        for w ← 0 to BLOCK_SIZE − 1:
            address = i + w;
            NVM.B[address] = SRAM[address];
NVM.bitmap = bitmap;
NVM.flag = BACKUP_SUCCESSFUL;
```

Additionally, one could think of computing and writing a CRC, or a cryptographic hash of the backup data (including the bitmap), to further protect the integrity of the backup. In this case, the checksum can be computed during the backup loop, pipelining the fetch of a block from SRAM, the computation of the checksum and the write in the NVM.

Restore

When the energy is newly available (i.e., when V_C is higher than $V_{restore}$), the controller starts the restore procedure, as described in Algorithm 4. First, the controller checks the flag in the NVM to know if the last backup was successful. If the flag is false, the last backup was not completed successfully, and the data in section **B** are corrupted. Thus, section **B** will be discarded and the controller will restore the SRAM only from the snapshot in section **A** of the NVM, which is a consistent backup. If the flag is true, the last backup was completed successfully and section **B** contains consistent data. Thus, the controller starts the restore sequence by first retrieving the dirty block bitmap from the NVM. Then the controller loops through all the blocks, checking if the corresponding bit in the bitmap is set. If a bit is set, the corresponding block has been modified during the previous interval. Therefore, its last value needs to be retrieved from section **B** of the NVM, and copied to the SRAM. If the bit is clear, the content of the block was not changed with respect to the last backup operation. In this case, the block is retrieved from section **A**. Additionally, every block read from section **B** is also written into the corresponding block in section **A**. This allows to update the content of the snapshot in section **A** with the changes recorded up to the last backup.

At the end of this procedure, the content of the SRAM has been restored and all the previously dirty blocks have also been copied from *NVM.B* to *NVM.A*. Thus, the primary area of the NVM now contains an updated and consistent version of the memory state. The controller then proceeds to clear the old dirty bitmap copy in the NVM and the flag. From this point, if no new backup is executed, the content of the memory can be safely restored from the snapshot section **A** of the NVM.

4.4.3 Cumulative Updates Strategy

The second proposed algorithm follows the cumulative updates (CU) strategy. As for RU, CU also divides the NVM into two main sections. The first section **A** initially holds the complete snapshot of the system. Moreover, the NVM also holds a bitmap of the modified blocks with the respect to the snapshot stored in section **A**. Besides, the modified blocks of the volatile memory are tracked at run-time by the controller using a dirty block bitmap. Finally, as in RU, at backup time, the dirty blocks are copied in a different section of the NVM, section **B**.

However, in CU, it is possible to successively backup the system by accumulating

Algorithm 4: Restore algorithm of the RU strategy

```
Input: NVM.bitmap, NVM.flag, NVM.B
Output: NVM.A : section A of the NVM
Output: SRAM
Data: bitmap[Nblocks], BLOCK_SIZE
if NVM.flag == BACKUP_SUCCESSFUL :
    bitmap = NVM.bitmap;
    for i ← 0 to Nblocks − 1:
        if bitmap[i] :
            for w ← 0 to BLOCK_SIZE − 1:
                a = i + w;
                SRAM[a] = NVM.B[a];
                NVM.A[a] = SRAM[a];
            bitmap[i] = 0;
        else :
            for w ← 0 to BLOCK_SIZE − 1:
                a = i + w;
                SRAM[a] = NVM.A[a];
        NVM.bitmap = 0;
        NVM.flag = false;
else :
    for a ← 0 to SRAM_SIZE − 1:
        SRAM[a] = NVM.A[a];
```

updates in section **B**, without previously having copied the modified blocks from section **B** into the section **A**. By updating the bitmap instead of updating the original snapshot, this algorithm reduces the number of NVM writes required with respect to the RU algorithm. However, in case of a backup interruption, this algorithm loses the progress and has to restore from the consistent backup, the one in section **A**. This effect can be mitigated by periodically synchronizing the snapshot in section **A** with the changes in section **B**. The number of intervals between two snapshot synchronizations N_{synch}, is a parameter that can be tuned to find a trade-off between a reduced number of NVM writes and the loss of progress in case of backup interruption due to a power failure. In case of a more stable power source, or when a larger capacitor is available, N_{synch} can be set to a higher value, reducing the overhead with respect to the RU strategy. On the other hand, N_{synch} can be reduced when the platform is more constrained and power failures are expected with a higher chance.

Run-time

As for the RU algorithm, the backup controller tracks the write accesses to the volatile SRAM memory at run-time and marks the corresponding blocks as dirty in its own bitmap.

Backup

The backup procedure is described in Algorithm 5. It starts by clearing the flag in the NVM indicating that the current cumulative update is being modified. As for the other algorithm, this flag can be a simple Boolean value. It will be used to guarantee the consistency of the backup. At backup time, the dirty blocks are copied in section **B** of the NVM ($NVM.B$). Blocks that are not marked as dirty do not need to be saved, as they are already available in the primary area of the NVM ($NVM.A$). After the modified blocks have been copied to section **B** of the NVM, the algorithm proceeds with the update of the dirty block bitmap stored in the NVM. This update consists of a bit-wise OR between the version of the bitmap stored in the NVM and the current run-time version of the bitmap with the newly modified blocks. Therefore, this bitmap is an accumulative version of the successive updates, which keeps track, in section **B**, of all modified blocks since the last update of section **A**.

Algorithm 5: Backup algorithm of the CU strategy

```
Input: SRAM : SRAM memory
Output: NVM.B, NVM.bitmap, NVM.flag
Data: bitmap[Nblocks], BLOCK_SIZE

NVM.flag = false;
for i ← 0 to Nblocks − 1:
    if bitmap[i] :
        for w ← 0 to BLOCK_SIZE − 1:
            address = i + w;
            NVM.B[address] = SRAM[address];
NVM.bitmap = bitmap ∨ NVM.bitmap;
NVM.flag = BACKUP_SUCCESSFUL;
```

After the bitmap in the NVM is updated, the backup phase can be concluded by clearing the run-time version of the bitmap in the controller, as all the modified blocks have been pushed to the NVM, and by setting the flag in the NVM to mark a successful backup. This flag can be enhanced by including some integrity protection, such as CRC code or a checksum of the whole backup. Moreover, also in this case, the backup could be

encrypted to improve the security of the application.

Restore

The restore phase for this algorithm is significantly simpler than in the RU case. It begins by reading the flag to check if the last backup was successful. If the flag is not set, the backup was interrupted and the system is in an inconsistent state. In this case, the volatile memory is restored from section **A** of the NVM, which contains an older but consistent state. This however results in the loss of the progress made so far. If, on the other hand, the last backup was successful, then the system can be restored from its last backup. In this case, the restore phase proceeds by retrieving from NVM the last updated version of the dirty blocks bitmap. Then, the SRAM memory content is restored block by block copying its content from section **B** when the corresponding dirty bit is set, and from **A** when the bit is clear. After this process, the system can resume the execution.

4.5 Robustness Against Power Failures

4.5.1 Robust Backup and Restore Algorithms

The steps in the backup and restore procedures of the two algorithms are organised in such a way that, if a power failure occurs, especially during the backup procedure, the system can always recover a consistent state, thus avoiding to restart from the beginning of the execution. In particular, for the *Restore and Update* algorithm, if the backup process is stopped due to a sudden loss of power, the flag would not be set. This means that, when the platform restarts, the content is retrieved only from section **A** of the NVM, ignoring any incomplete backup data present in section **B** and the invalid bitmap stored in the NVM. Moreover, the backup area and the dirty bitmap area in the NVM, will be replaced by the next successful backup.

In a similar way, a power failure happening during the restore process (which is much less likely to happen) does not leave the platform in a corrupted state. In particular, even an interruption while modified blocks are being copied from **B** to **A**, would not result in system corruption. In this case, only the dirty blocks of **A** are updated, whereas the clean blocks are left untouched. Thanks to the dirty bitmap in the NVM ($NVM.bitmap$), the unchanged blocks of **A** with the modified blocks saved in section **B** always represent a consistent state of the system. This allows to replay the restore procedure until the end is

reached and the flag is clear, at which point section **A** would contain a full and consistent snapshot of the system.

For the *Cumulative Updates* algorithm, the main difference is in the case of a loss of power during the backup phase. In this case, the NVM flag will be cleared at the beginning of the backup phase as the $NVM.B$ would be inconsistent. Therefore, the system should restore the snapshot in $NVM.A$, which remains consistent, while all the accumulated intermediate backups will be lost. To overcome this limitation, the solution is to update (*synchronize*) the $NVM.A$ section with the changes stored in $NVM.B$. The snapshot can be synchronized periodically, e.g., once every N_{synch} successful backup operations. This solution limits the maximum amount of lost progress in case of a backup interruption, as the snapshot in $NVM.A$ will be at most N_{synch} backup periods behind the current state in $NVM.B$. Another possibility is to update the snapshot in $NVM.A$ when the number of modified blocks saved in $NVM.B$ surpasses a certain threshold. This constraints the cost of each synchronization, which are expensive operations involving reading and writing from NVM. Moreover, this strategy exploits the locality of the applications to delay the synchronization. In fact, modifications are expected to often hit the same blocks, thus the number of dirty blocks should grow relatively slowly. A more involved approach would be to change the frequency of the synchronizations depending on the condition of the platform, such as the amount of harvested energy or the frequency of power outages.

4.5.2 Probabilistic Failure Model

In this section, we present a model to estimate the probability of a power failure during the backup phase. A power interrupt is triggered when the voltage across the capacitor reaches a defined threshold (V_{backup}). After this point, the platform only relies on the small amount of energy stored in the capacitor E_{Cap}, to carry out the backup operation. Due to uncertainties and variability in the value of the capacitor and in the measurement of V_{backup}, or due to a power consumption which is higher than expected, the remaining energy might not be enough to safely complete the backup operation. When this happens, the backup is interrupted abruptly by a power failure, and the partially saved data need to be discarded.

To model the probability of experiencing a power failure during the backup operation, we consider the energy balance of the system. In particular, the available energy in the capacitor, E_{Cap}, must be larger than the energy required for the backup, E_{backup}. This

means that the difference between this two values ΔE must be positive :

$$\Delta E = E_{Cap} - E_{backup} \geq 0 \tag{4.1}$$

Thus, a power failure will happen when $\Delta E \leq 0$. We can then model the probability of experiencing a power failure as the probability that the remaining energy ΔE is less than zero : $P(\Delta E \leq 0)$. To compute this probability, we assume that the value of ΔE is normally distributed, with a mean value ΔE_μ and standard deviation $\sigma_{\Delta E}$.

$$\Delta E \sim \mathcal{N}(\Delta E_\mu, \sigma_{\Delta E})$$

To compute the mean and standard deviation of ΔE, we need to model the energy available in the capacitor E_{Cap}, and the backup energy E_{backup}. The energy available in the capacitor depends on the threshold voltage V_{backup} and on the minimum working voltage V_{fail} under which the system will be in off state, and is expressed as

$$E_{Cap} = \frac{1}{2}C(V_{backup}^2 - V_{fail}^2). \tag{4.2}$$

To estimate the energy consumption of a backup operation, we assume a simplified linear model, that depends on the amount of words N, that needs to be saved :

$$E_{backup} = N \times 3E_{cycle} \tag{4.3}$$

We considered three clock cycles per word to complete a sequential backup, which would be in line to the values reported in [Cho+19], where the DMA is used. Moreover, we assume that each clock cycle consumes, in average, a given amount of energy E_{cycle}. Using Equations (4.2) and (4.3), the mean value ΔE_μ can be computed, given the following :

- C_s the nominal value of the capacitor,
- $V_b = V_{backup}$ the threshold voltage,
- $V_f = V_{fail}$ the minimum running voltage,
- $E_c = E_{cycle}$ the energy per clock cycle,
- N the amount of memory to backup.

To each of these values is associated an uncertainty. We can then compute $\sigma_{\Delta E}$ using the

formula for the propagation of uncertainties as

$$\sigma_{\Delta E} = \sqrt{\left(\frac{\partial \Delta E}{\partial C_s}\sigma_{C_s}\right)^2 + \left(\frac{\partial \Delta E}{\partial V_b}\sigma_{V_b}\right)^2 + \left(\frac{\partial \Delta E}{\partial V_f}\sigma_{V_f}\right)^2 + \left(\frac{\partial \Delta E}{\partial E_c}\sigma_{E_c}\right)^2}, \tag{4.4}$$

where Eq. 4.4 assumes, as a first approximation, that the variables are independent and where σ_{C_s}, σ_{V_b}, σ_{V_f}, σ_{E_c} are the uncertainties associated to each variable. The values of components, such as the capacitor C_s, can be modelled as a random variables distributed according to a normal distribution, where the mean of the distribution is the nominal value, and the uncertainty is the standard deviation. This variation is related to effects such as aging, temperature, and manufacturing tolerances that can affect the value of the capacitor. On the other hand, the value of V_{backup} is decided by the programmer or the designer of a transient system, but it is ultimately the result of a measurement using a circuit like an ADC. This measurement has an uncertainty that depends on the type of ADC and on its resolution (i.e. the number of bits). Similarly also E_{cycle} and V_{fail} will depend on the processor, on variations in the manufacturing process, and on external conditions such as aging and temperature [LHE14].

Using this model we computed the probability of experiencing a backup interruption, given different values of C_s and V_{backup}. We set E_{cycle} to be 370 pJ/cycle, while the value of V_{fail} was set to be $1.8V$. These values are extracted from the datasheet of a MSP430 microcontroller [Ins21], a platform commonly used for intermittently powered devices. For the backup energy E_{backup}, we considered a system performing a full memory backup [Bal+15], such that N is the total number of words. Using $N = 8192$ and the values previously reported, we obtained a map of the probability of backup failure as shown in Fig. 4.5. In Fig. 4.5, V_{backup} is expressed as the difference $\Delta V = V_{backup} - V_{fail}$ with the fixed value of $V_{fail} = 1V$ and is plotted as a function of the storage capacitor C_s. Larger C_s would provide more energy to the system but would also increase its form factor and cost. Fig. 4.5 shows how different values of C_s and V_{backup} correspond to a probability of failure during the backup, considering the different uncertainties of the system. For the probabilities depicted in Fig. 4.5, we set the relative uncertainty on the value of C_s to be 20%, as this is often the tolerance for cheaper ceramic capacitors, that can even be made worse due to temperature. For the voltages V_{backup}, V_{fail} we use a relative uncertainty of 2.5%, as this is a possible value for the total error for the 12-bit ADC available in the commonly used MSP430 MPU [Ins21]. However, considering an ultra-low power platform, this value could be worse especially if a smaller number of bits is used. Finally we assumed a 5%

uncertainty on the value of E_{cycle}.

As it can be seen from Fig. 4.5, obtaining a low probability of backup failure, say less than 1% with a small value of C_s, implies a significant increase in the value V_{backup}. This is equivalent to spend the majority of the energy stored in C_s for the backup/restore

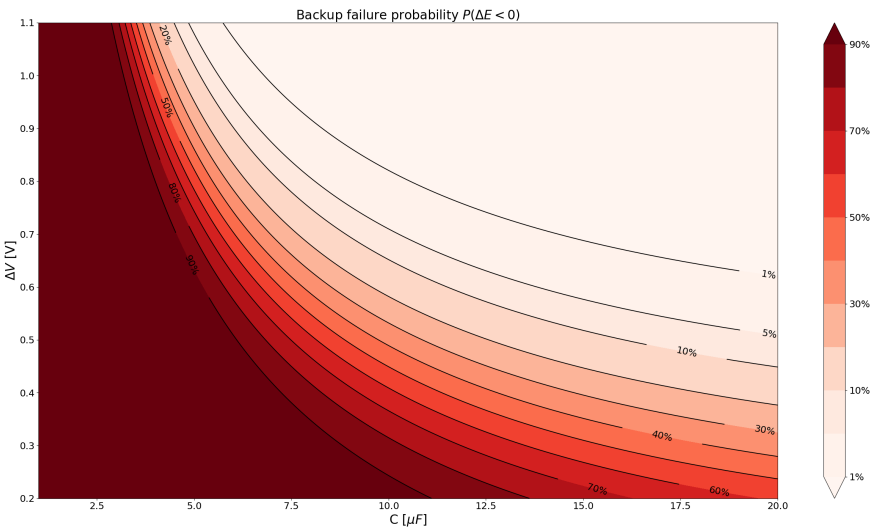


FIGURE 4.5 – Probability of backup interruption, given capacitance and threshold values. With N=8192 words (32kB of data), $V_{fail} = 1.8V$, $E_{cycle} = 370pJ/cycle$, $3\sigma_C = 20\%$, $\sigma_{V_b} = 2.5\%$, $\sigma_{V_f} = 10\%$, and $\sigma_{E_{cycle}} = 5\%$

operations, and not for the essential execution of the application. In particular, when the harvested energy is very small and cannot directly sustain computations, this energy needs to be accumulated in the capacitor. In these cases, the available energy for computations, during an interval, will be provided by the capacitor discharging from $V_{restore}$ to V_{backup}. Remind that $V_{restore}$ is the threshold voltage at which the platform is powered on again and the restore operation is started. The value of $V_{restore}$ cannot exceed the breakdown voltage of the capacitor, moreover it cannot be made too large or the charge time of the capacitor will increase. On the other hand, using a larger capacitor might provide additional energy storage that can be used for advancing computations, but could lead to an increase in the cost, and potentially in the size or volume of the platform. Thus, reducing the value of V_{backup} will result in more energy available for the computations, without the need to increase the value of C_s.

Fig. 4.6 reports the average probabilities of a backup interruption as a function of V_{backup} for five different backup strategies : Cumulative Update (CU), Restore and Update

(RU), full memory backup (FU), Differential (DI) and full memory Double Buffering (DB). The figure reports the probabilities for different benchmarks from MiBench (refer to Section 4.6.1 for details on the benchmarks) with different values of capacitor. As it can be seen from Fig. 4.6, our proposed approaches CU and RU always provide a reduction in the probability of backup interruption, thanks to the smaller size of the backup. This allows our strategies to operate with smaller capacitor values, even when the full memory backup is not possible. Alternatively, CU and RU can achieve the same probability of backup failure with smaller V_{backup}, effectively increasing the amount of energy per interval that can be spent for computations.

4.6 Results

In this section, we analyze the overheads of differential backup strategies and present the results of our simulations. For our simulations, we used the model proposed in the previous section to compute the power failure probability, and we combined it with a trace-based simulation. Section 4.6.1 presents the experimental setup as well as the simulation procedure used to provide the reported results. Then, in Section 4.6.3, we present a qualitative and quantitative analysis of the memory footprint and average backup size in order to compare the proposed robust algorithms with different strategies from the state of the art. Finally, Section 4.6.4, describes a qualitative analysis of the hardware overhead of our robust algorithms.

4.6.1 Experimental Setup and Simulation Procedure

To evaluate our proposed approach, we simulated the different backup strategies using memory access traces of different benchmarks. The benchmarks used are a subset of the MiBench benchmark suite : cjpeg, fft, matmul, susan_edge, susan_smooth, ssearch. The memory access traces are produced by running the benchmark with a cycle-accurate bit-accurate RISC-V processor simulator [Rok+19]. Furthermore, to simulate an intermittently-powered scenario, we assumed that the harvester is not able to supply enough energy to power the platform. Following the scheme defined in Section 4.2, the energy required for computations is first collected in the capacitor. When the voltage reaches $V_{restore}$ the platform wakes up. The wake-up time of the processor depends on the specific device. In our simulation, we rely on a setting equivalent to an MSP430 micro-

controller [Ins21], a popular device used for low-power, energy-harvesting applications. 0.5 ms is reported as the typical value for the wake-up time of an MSP430. The wake-up phase also consumes some energy from the capacitor and the MSP430 datasheet [Ins21] reports a typical charge of $\sim 120nAs$, consumed during the transition from reset to the active state.

After the wake-up, the memory is restored from NVM and the execution starts and continues until the voltage across the capacitor reaches V_{backup}, as shown in Fig. 4.3. For each simulated interval, the number of clock cycles available for the execution depends on the energy available in the capacitor between $V_{restore}$ and V_{backup} ($E_{V_r-V_b}$). To obtain the energy which is spent on actual computation, we subtract from $E_{V_r-V_b}$ the energy required for the wake-up and the energy required for the restore from NVM and backup to NVM. This give us the execution energy E_{exec} for the current interval. The energy consumed for the restore is computed based on the number of words that are restored, which is a constant equal to the full memory size, for the *Full Backup*, *Double Buffering* and *Differential* strategies. For the proposed *RU* and *CU* strategies, the energy of the restore phase will be slightly larger with respect to a plain *Differential* strategy. For *RU*, the energy of a normal restore (*i.e.*, when the previous checkpoint is valid) is the constant full memory size, plus the energy for updating the dirty words in the NVM, (which should be comparable to the energy of the previous backup) as described in Section 4.4.2. For *CU*, the restore energy is most of the time similar to the other strategies (which depends on the size of the memory), except when a synchronization is executed (see Section 4.4.3).

To obtain the number of clock cycles of actual execution, we divide the E_{exec} by the energy per clock cycles consumed by the CPU E_{CPU}. The execution of the program is then advanced by this number of clock cycles in the processor simulator, after which the voltage across the capacitor reaches V_{backup}. Then, for each backup strategy, the size of the backup, i.e. the number of words that needs to be saved, is computed. The backup size is constant for full-memory backup strategies, whereas it depends on the number of modified memory regions for differential backup strategies. The trace can be used to compute this number, and the total size of the memory used by the program. The energy for the backup can then be deduced from the traces.

At the end of each simulation cycle, the probability of power failure during the backup is computed. The probability depends on the backup size, the value of the capacitor C, the threshold voltage V_{backup}, the minimum working voltage V_{fail} and the energy per clock cycles during the backup operation $E_{backup/cycle}$. This probability is computed according

to the model presented in Section 4.5.2.

4.6.2 Probability of backup interruption

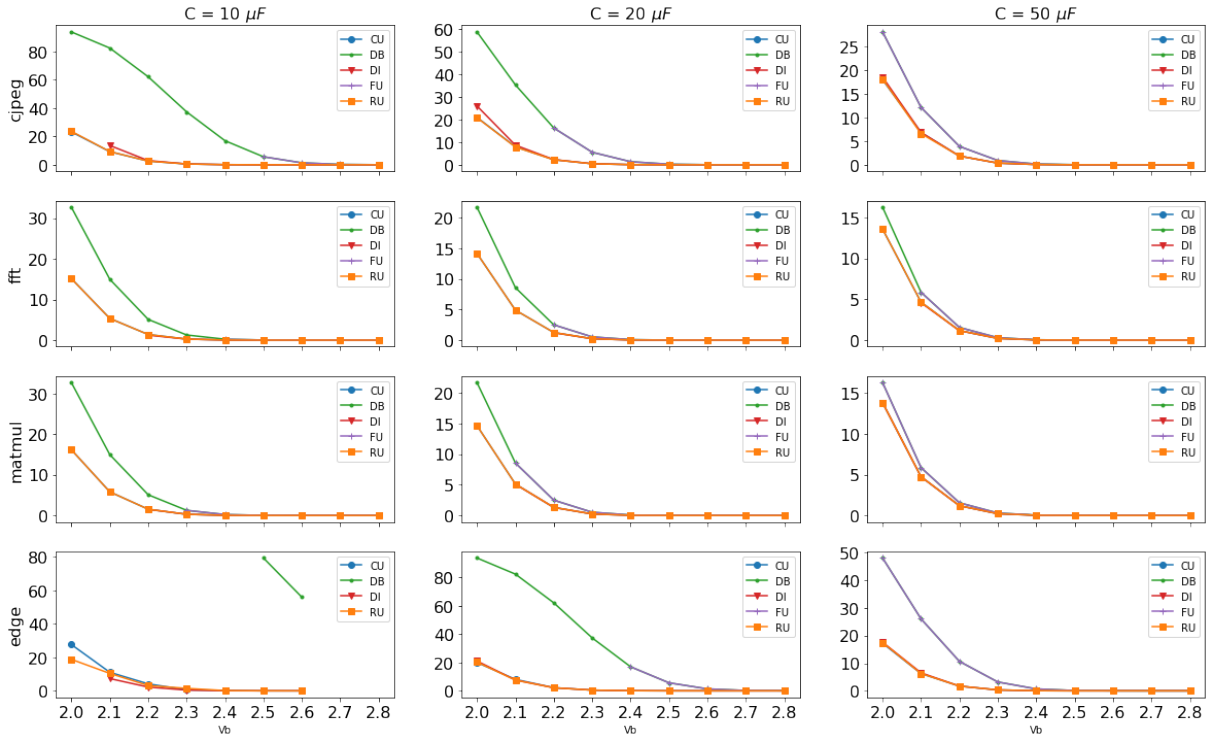


FIGURE 4.6 – Probability of backup interruption simulated for different benchmarks and different capacitor values, and for five different strategies : Cumulative Update (CU), Restore and Update (RU) and full memory Double Buffering (DB). Differential (DI) and full memory (FU) backup strategies are only reported when the benchmarks were able to run to completion.

The probability of backup interruption for the different strategies shown in Fig. 4.6 mainly depends on the backup size, which is determined by the benchmark considered. Therefore, the curves for the different differential backup strategies are often overlapping ; the full backup and double buffering curves also overlap each other. As an example, this is the case for most of the benchmarks with $C = 50\mu F$, where due to the abundance of energy storage, all benchmarks are able to complete their execution for almost all strategies, and only two lines are visible : one for differential strategies (CU, RU, DI), and one for the full memory strategies (DB and FU). One exception is the *fft* benchmark,

where at $C = 50\mu F$, the FU strategy is not able to complete with $V_{backup} < 2.1V$. For $C = 10\mu F$ and $C = 20\mu F$, this effect is also present for the other benchmarks. As an example, in *cjpeg* with $C = 10\mu F$, the FU strategy is only able to complete when $V_{backup} \geq 2.5V$, while for the *matmul* benchmark with the same C, FU only works when $V_{backup} = 2.3V$. The probability of power failure can also limit the unprotected differential strategy (DI) with smaller capacitors. Figure 4.6 shows this feature with $C = 10\mu F$, since DI is unable to complete the *cjpeg* and *edge* benchmarks for $V_{backup} < 2.1V$.

4.6.3 Analysis of Memory and Backup Data Size

In this section, we present a qualitative and quantitative analysis of the memory footprint and average backup size for the five different strategies to highlight the advantages of the proposed robust algorithms.

Table 4.1 – Comparison of the required size for the NVM, and the progress loss in case of backup corruption.

Algorithm	NVM size	Progress loss
Full Backup	1×	all
Differential	1×	all
Double Buffer	2×	1 interval
RU Strategy	$> 1\times, \leq 2\times$	1 interval
CU Strategy	2×	N_{synch} intervals

Table 4.1 summarizes the loss of progress in case of backup interruption and the required size of the NVM, for each of the strategies. *Full Backup* (FB) refers to a full memory backup, such as the one presented in [Bal+15], where a failure during the backup process results in a complete loss of progress. Similarly, the *Differential* (DI) backup algorithm refers to a differential backup strategy such as the one presented in [PMS20], which also does not protect the backup phase from unexpected interruptions. With DI, an interruption during backup would results in the complete loss of progress and in a corruption of the system state which cannot be recovered, as in FB. The *Double Buffer* (DB) algorithm alternates the use of two backup sections, such that if a power failure interrupts a backup, the system can always restore the previous valid backup [Cho+19]. Similarly, the *Restore & Update* (RU) algorithm also keeps the most recent snapshot

intact during the backup procedure, this means that the lost progress is limited up to the previous backup. However, because RU is a differential backup algorithm, the backup size is greatly reduced, as shown in [PMS20 ; Ahm+20 ; Ber+20]. This comes with a reduction of backup time and thus greatly lowers the probability of depleting the energy in the middle of the backup phase. For the same reason, the *Cumulative Update* (CU) algorithm also lowers the probability of a backup interruption. However, CU trades a reduction in the number of NVM writes during the restore phase, with a reduced protection against the loss of progress. The progress lost in case of backup interruption will depend on when was the last synchronization of the state performed (N_{synch} intervals in the table).

Because FB and DI backup strategies do not provide backup robustness, the NVM can have the same size as the volatile SRAM memory and there is therefore no overhead in the NVM size (1× in Table 4.1). With the *DB* strategy, only the progress made since the previous snapshot is lost in case of a backup interruption. However, DB always requires that the capacity of the NVM is at least twice (2×) that of the volatile memory. Our RU algorithm does not save the same amount of blocks every time as it only saves the modified blocks in section **B**. However, in the worst case, it is possible that the size of the backup amounts is equal to the full volatile memory. Therefore, the available NVM should also be at least 2× the size of the volatile memory. Furthermore, with our CU algorithm, there is the possibility of setting a threshold on the maximum number of dirty blocks allowed before triggering a new backup, thus limiting the maximum number of blocks in section **B**. It is then possible thanks to CU to use an NVM whose size can be tuned from slightly larger than the SRAM up to twice its size.

Then, Table 4.2 reports the mean, minimum and maximum average backup size per intervals that were obtained through our simulations for four different benchmarks. In the table, the backup size for the DB strategy corresponds to the total capacity of the volatile memory, and can therefore be considered as an upper bound used for reference. The FB strategy is not reported in Table 4.2, its backup size being equal to that of FB, as FB always copies the entire state of the volatile memory.

The backup size for the differential backup algorithms depends on the granularity of the memory blocks that are tracked. In our simulation, we used blocks of 8 words (each word is 4 bytes) for all the differential strategies. In [PMS20], a differential backup scheme with blocks of 8 words is shown to provide a backup size of around 20% of the volatile memory state in average. In our simulations, we obtained similar results, with backup

Table 4.2 – Average, minimum and maximum values of the backup size in number of words for various benchmarks from the MiBench suite and for different strategies.

Bench	Strategy	min	average	max
cjpeg	DB	8192.00	8192.00	8192.00
	CU	267.47	1418.58	3320.08
	DI	210.31	1531.59	3461.36
	RU	253.31	1401.36	3285.63
fft	DB	2048.00	2048.00	2048.00
	CU	181.30	252.74	297.16
	DI	106.74	234.29	282.58
	RU	180.84	251.54	296.18
matmul	DB	2048.00	2048.00	2048.00
	CU	189.15	366.27	448.99
	DI	185.18	365.81	473.90
	RU	186.73	365.19	449.79
edge	DB	16384.00	16384.00	16384.00
	CU	70.14	1027.74	3206.42
	DI	70.76	1009.49	3240.07
	RU	69.52	988.57	3265.04

sizes on *cjpeg* ranging from about 7%, up to 38% of the total memory, depending on the simulation parameters. In particular, with a higher harvested energy per clock cycle, and a larger capacitor, the application is interrupted less often, leading to longer intervals and thus larger backup sizes. This happens because, with longer uninterrupted executions, more regions of the memory are modified during an interval, thus increasing the backup size. We simulated these strategies using different configurations, changing the value of the capacitor C_s, the energy input from the harvester $E_{harvested/cycle}$ and with memory access traces extracted from different benchmarks. For each configuration we run the simulation with values of V_{backup} ranging from 2 V up to 2.9 V, with 0.1 V increments.

As shown in Table 4.2, our proposed strategies CU and RU provide the same average backup size as the similar, non-robust differential strategy (DI), which can still be considered as the reference in current related work.

4.6.4 Overhead Analysis

While full memory backup strategies, with or without Double Buffering (DB) can be implemented via software on platforms equipped with addressable NVMs, implementing differential strategies is more challenging. In [Ahm+20], a software-based system to implement differential backup is presented. The proposed approach relies on code instrumentation and pre-processing to mark and identify changes in the application state, and record them in main memory. This approach greatly reduces the checkpoint overhead with respect to approaches like Hibernus [Bal+15]. However it results in run-time overhead due to the insertion of additional code, required to track memory writes. In [PMS20], a hardware module to support differential backup is presented. The approach does not introduce a big hardware overhead as the proposed controller only requires a small memory bitmap, which could be implemented with registers, to track modified regions of memory. However, the proposed approach lacks the robustness against power failures. Thus, the benefit of differential backup cannot be fully exploited, and the capacitor and threshold voltage values needs to be set pessimistically to high values, like the ones for a full memory backup.

Our proposed approach also uses a small memory or register file, to mark dirty regions of memory. However, the proposed controller implements the robust backup algorithms introduced in Section 4.4, providing a strong guarantee against the corruption of the backup state. As the algorithms can be implemented with simple state machines, the main overhead introduced by our algorithms consists in the bitmap memory. Depending

on the desired granularity, and on the size of the memory, this bitmap can be composed of few flip-flops or memory bits. Following the example of Fig. 4.4 where a 16-bit bitmap is used to track changes in a 2KB memory (2048 bytes) composed of 16 blocks of 128 bytes, only 16 flip-flops can be used for tracking the memory sections. For the more relevant case of a larger 64KB memory, also decomposed in blocks of 128 bytes (i.e. 4 32-bit words), a 512-bit memory or register file is needed, which corresponds to a small area and energy overhead compared to the overall system.

4.7 Exploiting Robust Backup to Rise System Efficiency

The use of a robust backup and recovery system not only serves to preserve progress made in the face of unexpected backup failures, but can also be beneficial in improving overall system efficiency. Many non-robust backup systems, such as the ones proposed in [PMS20 ; Bal+15], need to assume that the capacitor holds enough energy to allow a complete backup of the system state. This assumption imposes constraints on the design of the platform, possibly requiring to oversize the storage capacitor or to increase the threshold voltage V_{backup}. Moreover, this assumption might become false during the lifetime of the device, not only due to variability and environmental conditions, but also to aging of the devices. The use of a robust backup system allows these constraints to be relaxed. As an example, a differential backup scheme, such as the ones proposed in [PMS20 ; Ahm+20 ; Ber+20], normally results in a significant reduction of the backup size. However, because a corruption of the backup cannot be recovered, it still needs to make the worst case assumption of a full memory backup when it comes to the capacitor size and the backup threshold voltage as is the case for [PMS20]. On the other hand, our proposed algorithms can be used with more optimistic setups. Because of the backup robustness, lower thresholds voltages, or smaller capacitor values could be used, accounting for the average energy consumption instead of the worst-case full backup scenario.

Fig. 4.7 compares the total energy consumption as a function of V_{backup}, for both protected (RU, CU, DB) and unprotected (DI, FU) strategies, using a $10\mu F$ storage capacitance and for four different benchmarks. The parameters for this simulations are reported in Table 4.3. As it can be seen from Fig. 4.7, our proposed approaches CU and RU provide a minimization of the total energy, and allow the platform to run on with lower V_{backup}. Only on benchmarks with small memory footprint, such as the FFT, the

Table 4.3 – Parameters of the simulation.

V_{fail}	1.8 V
$V_{restore}$	3.3 V
E_{CPU}	190 pJ/cycle
$E_{backup/cycle}$	370 pJ/cycle
$Q_{wake-up}$	123 nAs
$t_{wake-up}$	0.5 ms
$3\sigma_C$	20%
$\sigma_{V_{fail}}$	10%
$\sigma_{E_{backup/restore}}$	5%
$\sigma_{E_{CPU}}$	5%

Double Buffering (DB) can also provide a slight advantage for small value of V_{backup} over an unprotected differential strategy (DI) or even over CU. This happens when the cost of re-execution due to failures during backup outweighs the cost of a larger backup size paid by DB. This shows the importance of having a protection mechanism, that allows to recover from potential corruption of the backup state.

At the same time, Fig. 4.7 shows that for larger benchmarks such as *edge* or *smooth*, full backup strategies such as DB and FU are not really viable, or are always worse than differential strategies. FU is not plotted on Fig. 4.7 for *edge* and *smooth*, since its total energy is largely out of the scale. This happens when the capacitor is too small to sustain full memory backup plus the execution, limiting the viable range of V_{backup} for DB and FU strategies. In particular the full backup (FU) strategy, requires much larger V_{backup} to avoid to infinitely restart from the beginning of the program, as no progress can be made in case of failure. Additionally, for the *edge* benchmark and the given configuration, it is impossible for FU to execute to completion within the limit of 1000 intervals. Due to its large backup size, Double-buffering (DB) has the same high probability of failure as FU, but it can work with lower V_{backup}, since a power failure only makes the system restart from the previous safe snapshot.

As already mentioned, DB can have a minor advantage even over CU for very small benchmarks such as *fft* and only for very low V_{backup}. This is due to CU losing N_{sync} intervals in case of failures, whereas DB only looses one interval. For these simulations the synchronizations happens every $N_{synch} = 5$ backups. This parameter provides a trade-off between the energy overhead during the restore phases, and the energy due to the penalty of a backup power failure. Setting a lower value when working with small threshold

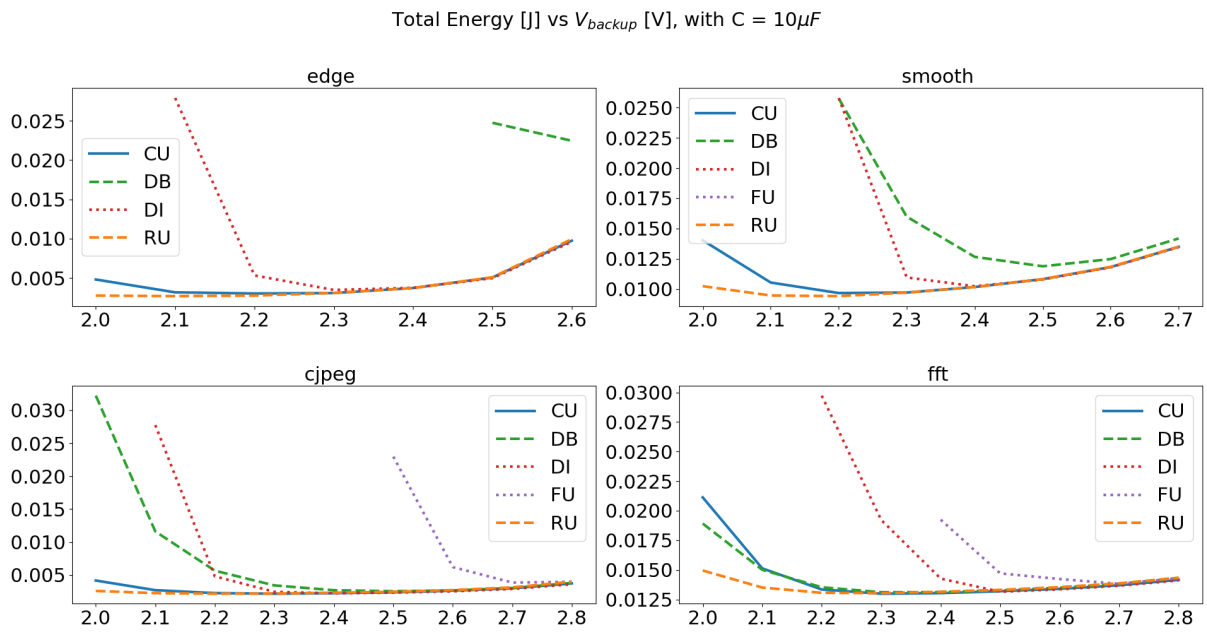


FIGURE 4.7 – Total energy consumption $[J]$ under different V_{backup} $[V]$ with constant harvested energy $E_{harvested} = 2pJ/cycle$, storage capacitor of $10\mu F$, for both protected (RU, CU, DB) and unprotected (DI, FU) strategies, and for four different benchmarks.

voltages would result in CU behaving similarly to RU, with exactly the same behaviour for the case of $N_{synch} = 0$. This effect is shown in Fig. 4.8, where the total energy of backup and restore as a function of V_{backup} is reported. On the other hand, increasing N_{synch} when the probability of backup interruption is low, such as with higher V_{backup} or larger capacitor, gives an advantage to CU over RU. However this advantage is only significant for some smaller benchmarks (e.g., *fft*), while there is almost no advantage for larger benchmarks such as *susan smooth*. This is mostly due to the large restore size of this benchmark, which makes the advantage of higher N_{synch}, less frequent synchronizations, negligible. Moreover, Fig. 4.8 shows that even for more sensitive benchmarks, such as *fft*, there is not a great advantage in increasing N_{synch} above a certain value. Additionally, while tuning N_{synch} does bring a measurable advantage at low V_{backup}, with higher V_{bakup}, the impact on the total energy, even for benchmarks such as *fft*, is small. In fact, the total energy greatly depends on the energy for execution and on the wake-up and restore energy, which at high V_{backup} are not influenced by N_{synch}. A hybrid approach would be to tune the N_{synch} parameter depending on the probability of failure. Setting lower values of N_{synch} for more constrained systems, and higher values when the probability of interruption is

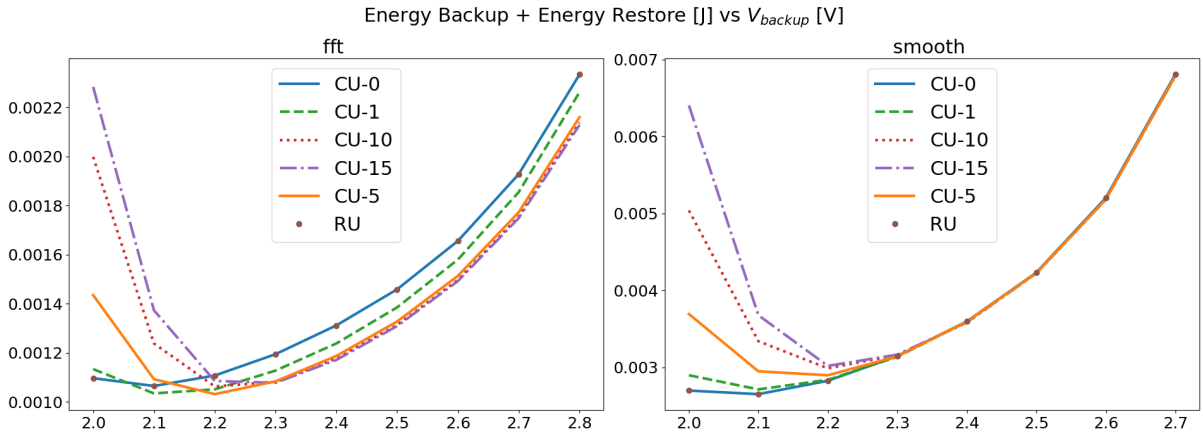


FIGURE 4.8 – Sum of backup and restore energy $[J]$ as a function of V_{backup} $[V]$ for RU and CU with different N_{synch} for fft and smooth benchmarks.

very low.

Fig. 4.9 shows the number of interruptions as a function of V_{backup} for the same set of benchmarks and with the same parameters. An increased number of interruptions lead to higher energy overhead due to more backups and restores, as well as the additional wake-up energy and time.

As shown by Figures 4.7 and 4.9, our two proposed schemes RU and CU can lead to significantly lower thresholds (or similarly to smaller capacitance), and therefore to a more efficient usage of the energy stored in the capacitor. In Table 4.4, we report the minimum values of the total energy consumption, as represented in Fig. 4.7, for a larger set of benchmarks. In the table, the best values are highlighted in bold. As it can be seen, our proposed approach always minimizes the total energy consumption of the system.

As reported in Table 4.4, our proposed strategies CU and RU respectively achieve, in average, a 23.2% and 23.7% reduction in the minimum total energy with respect to a double-buffering approach (DB).

Table 4.5 reports the best total running time, for each strategy, for a subset of benchmarks, considering a storage capacitor C_s of $10\mu F$ and an harvested energy of $E_harvested = 2pJ/cycle$. For the total run-time, the trend is very similar to the total energy. With our proposed strategies, the minimum total running time can be reduced by, in average 23.3% (for CU) and 23.8% (for RU), when compared with the DB strategy.

TABLE 4.4 – Minimum Total Energy [J], with $C_s = 10\mu F$ and $E_{harvested} = 2pJ/cycle$, best values in bold.

bench	cjpeg	edge	fft	matmul	smooth	ssearch
CU	0.002142	0.003016	**0.012980**	**0.007065**	0.009664	0.003122
DB	0.002528	0.022457	0.013112	0.007100	0.011877	0.003778
DI	0.002207	0.003467	0.013146	0.007158	0.010225	0.003349
FU	0.003835	NaN	0.013815	0.007328	0.018302	0.005542
RU	**0.002139**	**0.002680**	0.013031	0.007139	**0.009409**	**0.003084**

TABLE 4.5 – Minimum total running time [s], with $C_s = 10\mu F$ and $E_{harvested} = 2pJ/cycle$, best values in bold.

bench	cjpeg	edge	fft	matmul	smooth	ssearch
CU	67.55	95.01	**409.94**	**223.13**	304.86	98.48
DB	79.83	709.12	414.24	224.30	375.16	119.33
DI	69.59	109.19	415.19	226.08	322.47	105.63
FU	121.14	NaN	436.55	231.52	578.14	175.05
RU	**67.48**	**84.43**	411.54	225.45	**296.82**	**97.27**

4.8 Conclusion

Systems powered through environmental energy need to be resilient to unpredictable power outages. To sustain computation these systems often rely on check-pointing mechanisms to preserve the state in a non-volatile fashion. Many works proposed in the literature focus only on the execution of the backup, assuming that sufficient energy is available when the backup is triggered. Others have proposed more robust strategies, but rely only on the use of NVM thus introducing significant run-time overhead. In this work, we present two robust and efficient backup and restore algorithms for intermittently-powered systems. Our approaches are more efficient than a normal full memory backup, while also guaranteeing at any point in time the existence of a correct backup state that can be reverted. Moreover, we show how this stronger guarantee can improve the efficiency of the system, enabling it to execute more instructions before a backup is performed. Quantitatively, both of the proposed approaches achieve an average reduction in the total energy of more than 23% when compared to a *double-buffering* approach, as well as reducing by a similar amount the total run-time of the application. Alternatively, our techniques can be used to reduce the size of the capacitor, and thus the cost and bulk of the device, while retaining the same level of forward progress.

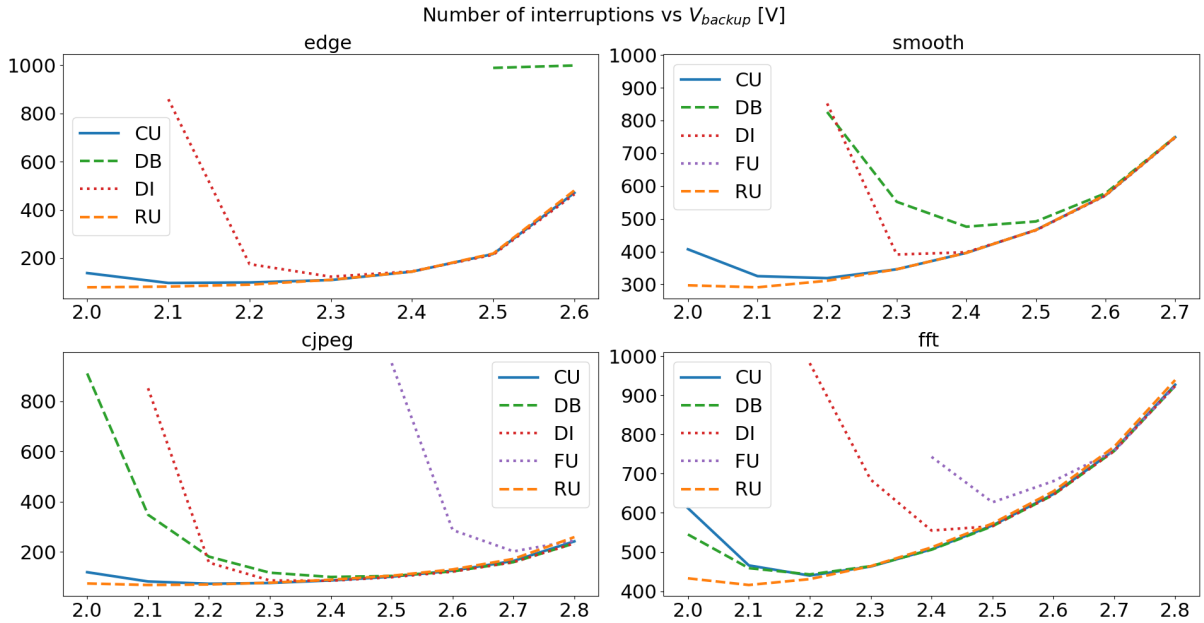


FIGURE 4.9 – Number of interruptions as a function of V_{backup} $[V]$ with constant harvested energy $E_{harvested} = 2pJ/cycle$, storage capacitor of $10\mu F$, for both protected (RU, CU, DB) and unprotected (DI, FU) strategies, and for four different benchmarks.

Chapitre 5

Bloom-Filter Based Memory Write Tracking for Differential Backup

Differential backup strategies save the difference between the current state and the previous backup, to minimize the amount of data that needs to be saved. This type of strategy can be useful in the context of intermittently-powered systems, that need some form of check-pointing to preserve computation across power interruptions. However, in order to apply a differential backup strategy, the platform needs to keep track of the changes in the state of the system memory. A simple approach to track memory writes is to use a bitmap, where each bit marks whether a section of memory has changed or not, as described in Chapter 3. This approach works better when the size of each memory section is small (e.g. 8 words), giving a finer granularity to track the changes. When the size of the system memory is small enough, this fine granularity is achievable with a relatively small number of bits. However, this approach is difficult to scale to larger memories, as it requires either a large bitmap or to increase the granularity of the differential tracking. This chapter presents an optimized differential backup scheme that uses approximate membership data structures, such as Bloom filters, to track the memory changes. This approach achieves better results than a comparable plain bitmap scheme, while enabling the use with larger main memories.

5.1 Introduction

In recent years, the use of environmental energy harvesting has been explored as a source to power edge devices. In particular, energy scavenging from the environment can enable battery-less operation for some of these systems. However, power from the environment is usually scarce, and unpredictable in nature. This means that these types of systems are transiently powered, and need to be able to withstand power outages without losing progress.

To enable these intermittent systems to operate across unexpected power failures, emerging Non-Volatile Memories are used to create Non-Volatile Computing Systems. One approach to enable intermittent computing is to make the whole processor non-volatile and use an NVM as the main memory. This approach however comes with a penalty in terms of both performance and energy consumption at run-time, as NVMs are normally slower and require more energy per access with respect to SRAM, especially when it comes to memory writes. Moreover most NVMs do not yet have the same level of endurance of SRAM, making them less suitable as a system memory replacement.

Another common approach is to use SRAM as a system memory and rely on some backup mechanism to save data on the NVM. To execute the backup, there exist both static and dynamic techniques. With static techniques such as periodic check-pointing, *check-points* are scheduled at compile time based on some heuristics, such as after function calls or at the end of each loop iteration, as proposed in [RSF11]. Other approaches also include the use of timers, also proposed in [RSF11], and the use of stack size analysis, to place backup points in places where the size of the stack, and thus the size of the backup, is minimized [Zha+15 ; Zha+17]. On the other hand, dynamic techniques implement *on-demand backups*. Approaches such as Hibernus [Bal+15 ; Bal+16] try to implement a more reactive system where the backups are only executed before a power-failure, in response to a triggering event. Contrary to static methods, approaches such as Hibernus execute full memory backups, as it is difficult to optimize the backup size, when backup events are unpredictable.

To optimize backup size in an on-demand scheme, we proposed Freezer [PMS20], which is detailed in Chapter 3, where a simple hardware module is used to spy the CPU's memory accesses, and track the modified memory blocks. A similar approach is presented in [Ber+20], where instead of using a dedicated hardware component, the memory protection unit (MPU) of ARM processors is used to track dirty pages. Similarly in [Sak+14], each internal register is paired with an additional dirty bit that is used to determine which register needs to be saved. While the use of dirty bits to implement differential backup schemes is effective, this approach is difficult to scale to larger memory sizes without losing in granularity.

In this chapter, a study of differential backup techniques using approximate membership data structures is presented. In particular, the use of Bloom filters and their integration with direct-map bitmaps, to track dirty memory blocks for a differential backup algorithm is investigated. Bloom filters are compact data structures that, using hashing

mechanism, can answer set membership queries. Finally a hybrid hierarchical strategy, that uses a bitmap to track writes at a coarse granularity (page level) with a Bloom filter used to track at a finer granularity (1-4 words), is presented. For each of the presented strategies, a number of parameters, such as the total number of bits, the number of keys, the level of granularity, etc., are evaluated, using a memory access trace based simulation. The use of plain Bloom filters, as well as Bloom filters with a false positive free zone (EGH filters) [Kis+18], is analyzed here.

The result of this simulation shows that the proposed hybrid approach can, for the same HW complexity (number of mapping bits), significantly increase the backup efficiency (reduce the useless backup data) with respect to a plain bitmap approach as presented in [PMS20]. Moreover, when considering larger main memory, the hybrid scheme can achieve smaller backup size with a reduced bitmap size with respect to the Freezer approach with a fine granularity of 8 words, which would require much larger bitmap to track a relatively large main memory.

5.2 Background and Related Work

This section presents a brief overview of the basic concepts and the related works on the subject of incremental backups for transiently-powered systems. Additionally, a brief and non comprehensive introduction to Bloom filters and other similar data structure is given.

5.2.1 Incremental Backup

Transiently-powered systems can adopt two main strategies to preserve their state across multiple power failures : static or periodic check-pointing, and on-demand backups. Contrary to static or periodic check-pointing, where the backup positions are predetermined, on-demand check-pointing is a backup strategy that defers the execution of the backup to the moment when a power outage is signaled. This allows for a more reactive system, and normally avoids the costs of roll-backs, as the backup is executed just before the power failure. However, this approach does not allow the same types of backup size optimization that are possible with static techniques.

Incremental or differential backup is a strategy that can be used to reduce the size of the backup in on-demand strategies. With incremental backup, the idea is that only what

has been modified since the last backup needs to be saved. This requires either computing or keeping track of the differences between the backup state and the current state of the system. While this is common in other domains, it is more challenging to achieve with the limited computing resources and energy available to energy-harvesting, intermittent systems.

Different forms of differential backup for intermittent systems, have been presented in the literature in recent years. In [Sak+14], a Non-Volatile Processor (NVP) based on STT-RAM is presented. The proposed system features hybrid non-volatile flip-flops that enable fast save and restore of the processor registers. To avoid unnecessary backups each register is paired to an additional dirty bit, which marks whether the current volatile content needs to be saved. This is effectively implementing incremental backups at the register level. One problem with this approach is that it requires exotic components such as hybrid magnetic flip-flops, and that adding an additional bit for each register incurs in a large overhead in terms of area, making this approach difficult to scale. Moreover, this method does not cover the main memory, which is fully non-volatile in the proposed NVP.

Extending incremental backup to the main memory is more challenging but solutions have been proposed both at the hardware level and at the software and/or operating-system (OS) level. Freezer is a hardware module that is able to spy the memory access of the CPU at runtime, and keeps track of the modified blocks using a bitmap [PMS20]. At backup time, the bitmap is used to copy from SRAM to the NVM only the blocks from the main memory that were modified during the execution. A similar approach is presented in [Ber+20] where, instead of a dedicated hardware component, the memory tracking is performed by exploiting the memory protection unit (MPU) of ARM processors. With respect to Freezer, this approach is easier to adapt to available off-the-shelf micro-controllers. On the other hand, Freezer is a custom component designed for the task, thus it offers better granularity and lower power consumption with respect to using the MPU. However, the use of a bit for each block of 8 words, as it is done in Freezer, is difficult to scale to larger memory sizes. For this reason, to improve the granularity, without dramatically increasing the bitmap size, we investigate the use of hashing and approximate membership data structures such as Bloom filters.

5.2.2 Hashing and Bloom Filters

Bloom filters [Blo70] are a space efficient data structure that is used to answer set membership queries. The Bloom filter offers two main operations : *insertions* and *queries* [Kis+18]. In their simplest form, Bloom filters consist of a bit array B of length m that is used to represent the set, and k hash functions, with $k << m$, that are used to map elements of the set to k different numbers, that are used to index the bit array [Kis+18]. To insert an element x in the set, k hash functions $h_1(x), \ldots, h_k(x)$ are k computed, and the resulting indices are used to set the corresponding bits in the array [Kis+18] :

$$B[h_i(x)] \leftarrow 1, \forall i \in 1 \ldots k$$

Figure 5.1 shows an example of an element x being inserted in a bit array B, using $k = 2$ hash functions h_0 and h_1.

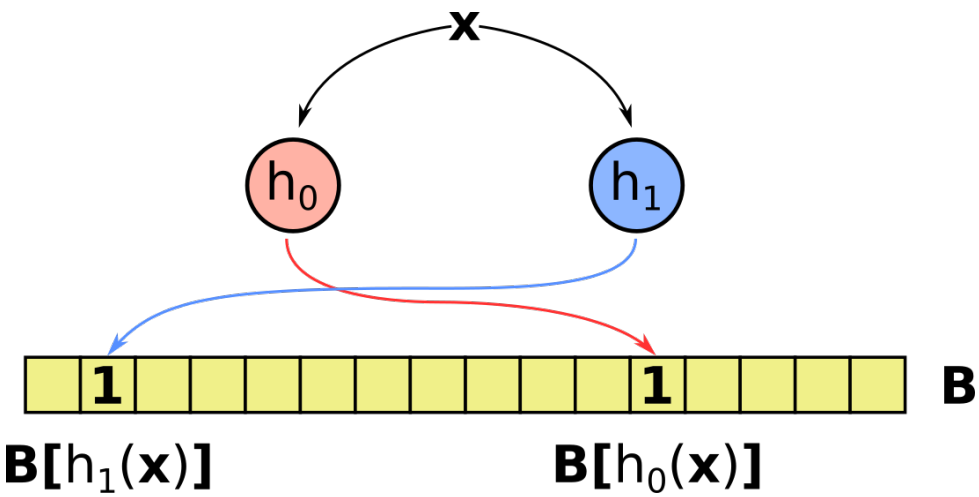


FIGURE 5.1 – Insertion of x in a 16-bit filter B with two hash functions h_0 and h_1.

To perform a query, i.e., to check if an element x is in the set S, the k hash functions are computed, and each corresponding bit in the array is checked to be one :

$$x \in S \implies B[h_i(x)] = 1, \forall i \in 1 \ldots k$$

If any of the checked bits is zero, then it is sure that the element x does not belong to the set S. However, because of the possibility of collisions, the membership of an element cannot be determined without uncertainty. This means that probability of a false negative is null. However, the probability of a false positive is not null, but can be bounded and

Algorithm 6: Xorshift-64 based hash function

Input: *cpu_addr*, address generated by the CPU

Output: x, 64 bit hash value

$x \leftarrow (1 << 32) \cup cpu_addr;$
$x \leftarrow x \oplus (x \ll 13);$
$x \leftarrow x \oplus (x \gg 7);$
$x \leftarrow x \oplus (x \ll 17);$

controlled by properly setting the filter parameters, and is the subject of many studies in the state of the art [Kis+18].

Many variants of the Bloom filter have been proposed in the literature, to improve performance or to adapt-it to different work-cases. As an example, an *Invertible Bloom Lookup Table* is presented in [GM11], that allows to list the inserted elements. In [Kis+18], a Bloom filter with a false positive free zone is presented. The proposed filter uses results from the context of *Combinatorial Group Testing* to build a bloom filter that, as long as the number of inserted elements is less than a number d, is guaranteed not to have false positives. The proposed filter, called EGH filter, also allows to list the inserted elements.

In this work, we seek to use the Bloom filter to store information about modified memory addresses. This means that other than insertion and query, a desirable operation would be to retrieve the list of inserted elements. However, because of the relative complexities of approaches such as [Kis+18] and [GM11], these filters might not be suitable for our context. The EGH filter [Kis+18] would require the hardware implementation of complex blocks such as modulo and multiplication operations. The invertible Bloom lookup table [GM11] would require additional memory for storing the filter. Instead, to enable a small and fast hardware implementation, we explored the use of a simple filter, and we use the *"xorshift"* function as the hash function for our Bloom filter [Mar03]. In particular we use a single round of the *xorshift-64* generator as a simple hash function. The code of this hash function is reported in Algorithm 6. As it can be seen from Algorithm 6, the cpu address (*cpu_addr*) is the input of the hash function. The input for the function is the 18-bit address arriving from the CPU, 20 bits are necessary for a 1MB address space, but the first 2 bits are discarded as tracking is done at the word level. The address is then extended in a full 64-bit word, by concatenating a 1 in the 32 MSBs. The 64-bit output is computed with three consecutive *xor* and *shift* operations. To obtain multiple keys for accessing the Bloom filter, the 64-bit output is simply divided. As an example, to obtain

two 12-bit keys for a 4096-bit Bloom filter, one could take the 12 LSBs from the output for the first key, and then the successive 12 bits for the second key.

5.3 System Model

The system model for this study is the same system model considered for Freezer in Chapter 3. We suppose that the platform is composed of a main CPU or micro-controller, with access to an SRAM used as the main memory, and with an NVM memory used for backup. We propose the insertion of a hardware module that sits besides the CPU core, and that at run-time, spies the memory accesses (similar to Freezer). At backup time, the proposed hardware module uses the information collected at run-time to copy from SRAM to the NVM all the modified blocks of memory. Similarly, at restore time the blocks moves back the content of the backup from NVM to SRAM. The main difference with Freezer in Chapter 3 is the insertion of the Bloom filter to increase efficiency and therefore to be able to deal with larger memory sizes, as described in the following section.

5.4 Filter-Based Memory Write Tracking

In this section, we present the different architectures that we explored to implement and optimize the memory write tracking for the backup and restore controller.

5.4.1 Plain Bloom Filters

The first architecture considered for this study is the use of a plain Bloom filter, instead of the bitmap that is used in Freezer. While the replacement of a bitmap with the filter seems straightforward, it brings several challenges. First of all, the Bloom parameters must be chosen to reduce the false positive rate. At the same time, because we want to implement this in a hardware module that operates in a very constrained environment, the filter needs to be designed considering the complexity of the hash function and the delay to access the bit-array. For this reason, limiting the number k of hash functions can be helpful, as it reduces the number of simultaneous accesses that need to be done to the bit array for each insertion and query operation. Another approach is to divide the bit array into parallel register blocks, so that multiple read accesses can be executed in parallel. This approach is however still limited by the possibility of conflicts, as two or

more hash functions could end up producing indices that access the same register block, thus creating contentions.

Another problem with the plain substitution of the bitmap with the Bloom filters, is that it does not allow for a fast retrieval of the modified words. In fact, for a direct map bitmap, every bit corresponds to a block of words, which allows to quickly retrieve the address of the modified blocks, simplifying the backup procedure. On the other hand, there is no easy way to invert the function of the Bloom filter, and extract the addresses from the bit vector. This means that, to perform the backup, the whole address space must be checked against the filter, and only those addresses that result in the filter should be considered for the backup. This adds overhead in the backup procedure by adding unnecessary checks in the filter. Moreover, it also increases the chance of collisions, as the whole universe (the address space) must be tested.

To mitigate these shortcomings, registers could be added to the hardware block to hold the boundaries and limit the search space. As an example, the simplest case would be to track the maximum and minimum written addresses using two registers. Then, the search in the filter could be limited within these two boundaries. With the addition of more of these registers, the address space could be further divided into more regions, and only addresses within those regions would be checked at backup time. This registers could be filled automatically by the hardware, or their value could be set by the programmer.

5.4.2 Parallel Bloom Filters

Another option to improve the search time during the backup operation is to use the available number of bits to implement multiple Bloom filters that could be accessed in parallel. The memory space is then divided into regions. A memory address is therefore composed of a region number (MSBs of the address) and the address inside the region (LSBs of the address). The filters could share the same hash functions, which would be computed only on the LSBs of the address, while the higher bits would be used to identify the region and thus the corresponding filter. This allows to check the same LSBs of the address in multiple regions at the same time, effectively dividing the size of the universe, with respect to the plain Bloom filter approach.

The main drawback of this approach is that by dividing the total available number of bits between multiple Bloom filters, the chances of collisions within those filters increase. As the objective is to embed the tracking hardware in the SoC, only a limited number of bits is available for the Bloom filters and, dividing them between multiple Bloom filters,

makes them smaller and less effective. Moreover, depending on the access patterns and on the locality of the application, the memory accesses might be confined to only few regions in between backups. When the accesses hits mainly one region, the corresponding Bloom filter would be saturated, while the others will remain mostly empty. This means that the bits that are dedicated to the other regions are actually wasted.

To counter this effect, one approach could be to use a different mapping between the addresses and the corresponding bloom filter, instead of just dividing the address space in consecutive regions.

5.4.3 Hybrid Hierarchical Bloom-Freezer

The last architecture that we propose is a combination of the bitmap approach of Freezer as described in Chapter 3 [PMS20] with a Bloom filter. The idea is to use the bitmap to divide the address space in contiguous regions, or pages, where each bit of the bitmap corresponds to a different region, and it is set to 1 if any of its words is modified in between backups. Then, a single Bloom filter is used and shared between the different regions to mark the modified blocks. Contrary to the previous case, the Bloom filter needs to use all the bits of the address. The blocks are the basic units of memory that are tracked by the system for the backup. A block could consist of few contiguous words (e.g., two or four words) or of a single word, if word-level tracking is desired.

The main idea behind this approach is to use the bitmap to constrain the search space, as only the marked regions need to be checked. This is a simpler alternative to using boundary registers with a plain Bloom filter, where pairs of registers hold the value of the minimum and maximum addresses determining the search regions. Instead, the bitmap is used to mark fixed, direct mapped, and aligned regions. While this does not offer the flexibility of tracking unaligned address ranges, it allows to track more regions with the same number of bits with respect to using those bits for the boundary registers.

Figure 5.2 shows an example of how the region bitmap and the Bloom filter could be used to check if a block is dirty (i.e., has been modified). The figure shows a 20-bit address, corresponding to an address space of $1MB$. In the example, the address space is divided into 16 regions, thus the 4 MSBs of the address are used to index the *Region bitmap*. In Figure 5.2, blocks of 4 words are been tracked, which means that the 16 MSBs of the address are used to index the blocks. Because only blocks are tracked, and not the individual words or bytes, the 4 LSBs are ignored. The block address is then feed to the hash function block, to generate k hash values used to index the Bloom array and read k

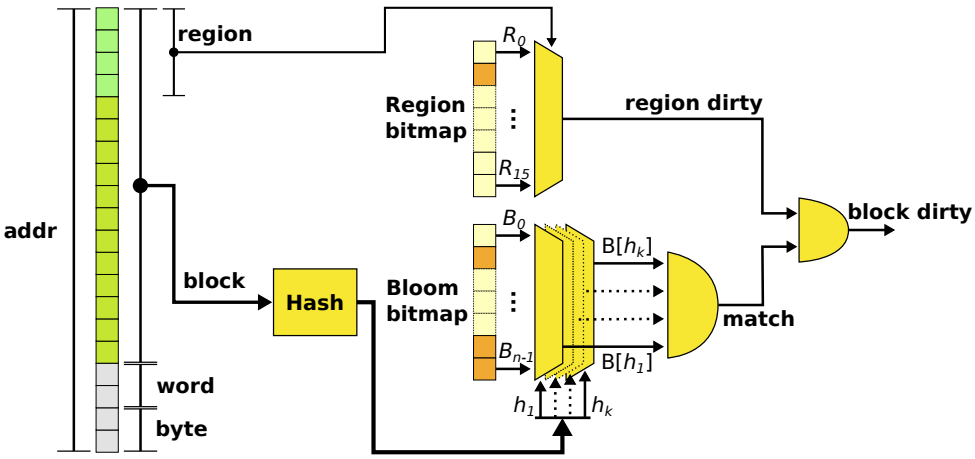


FIGURE 5.2 – Conceptual view of the dirty check mechanism of the Hybrid Bloom-Freezer approach.

bits. Only if all of them are set, the block matches (i.e., there is a high probability that it was inserted before). As shown in the figure, a block is considered dirty only if all the k bits are set and the region is dirty.

With the hybrid Bloom-Freezer approach, at run-time, when a store is executed, the most significant bits of the address are used to access the region bitmap and to set the corresponding bit to one. In parallel, the full block address (i.e., including the region bits) is used as input of the hash function, to compute the indexes for the Bloom bit-vector, and the corresponding bits are also set to one. At backup time, the information on the regions stored in the bitmap is used to limit the search through the Bloom filter to those regions that have their bit set. For each marked region, all the block addresses in the region must be checked against the Bloom filter. This means that, for all the blocks in a marked region, the hashes must be computed and the Bloom bit-vector must be accessed at the corresponding indices. Only if all those bits are set, the block is copied from SRAM to the NVM.

This approach speedups the backup time, with respect to a plain Bloom filter, by reducing the search space and limiting to only the addresses of the dirty regions. However, as it can be deduced, checking every block in a region, can also slow down the backup operation, especially when regions are very large and accesses are sparse. For this reason a careful balance must be searched when deciding how many bits should be allocated for the region bitmap, and how many for the Bloom bit-vector. In particular, increasing the size of the region bitmap, leads to a higher number of regions, because the memory size

is the same. This means that each region will be smaller, reducing the number of block addresses that must be checked for each region. This also helps in situations where the access pattern is sparse, as the overhead of checking a region with few blocks to backup is reduced. On the other hand, increasing the size of the region bitmap, leaves less bits for the Bloom filter, reducing its accuracy and increasing the possibility of false positives, and so increasing the size of the backup.

In the following Section some of these trade-offs are explored, and the results of experiments with the different parameters are presented.

5.5 Results

This section presents the results of a simulation where different filter strategies are compared with different parameters. Additionally, the results of a preliminary hardware implementation are reported.

5.5.1 Simulation Setup

In order to test the performance of the Bloom-filter based approaches and compare them with the bitmap-based approach such as Freezer, parametrizable models of the different strategies have been implemented in C++. The simulation takes a memory access trace as an input, while the different filters to be compared are all hard-coded in an array and embedded in the program at compile time. The simulation proceeds by reading lines from the trace file. Each line contains three fields : a time-stamp expressed in clock cycles, indicating when the current memory access was executed. The next field is the type of memory access, that can be *'L'* for load or *'S'* for store accesses. The last field contains the memory address. This is the same trace file as the one described in Table 3.1 of Chapter 3.

The clock cycle information is used to divide the execution in intervals. Each interval is supposed to be an uninterrupted period of execution, at the end of which there is a power failure. For these simulations, fixed-size intervals have been considered, which means that power failures happen always every N clock cycles. The interval size is a parameter of the simulation. Values of the interval size ranging from 100 thousands to 1 million clock cycles have been tested.

Another parameter of the simulation is the total number of bits available. This number gives the total size of the bitmap for a Freezer approach, while for the hybrid Bloom-

Freezer strategies, the total number of bits must be divided between the Freezer and the Bloom parts. For this reason, several configurations were tested, with different ratios of the total number of bits allocated to the Freezer part : 1/2, 1/4 and 1/8. As an example, a 1/4 ratio configuration with 2048 total bits, would result in a bitmap of 512 bits (i.e. one fourth of the total number of bits) for the Freezer part, and a 1536-bit Bloom filter.

When a store is encountered all the methods in the filter array are updated, i.e., inserting the address value in the bitmap for Freezer-like strategies, or updating the Bloom filter for filter-based strategies. When the last memory access of the current interval is read from the trace file, the backup procedure is triggered for all methods. At this point, the number of dirty bits and the size of the backup is computed for all the tested methods.

As a baseline comparison, a *Set* strategy is also implemented, a strategy that uses a perfect set (i.e. without collisions), to keep track of the actual number of modified addresses. To implement such a strategy using a bitmap to mark modified addresses, the bitmap would need to have one bit for each word in the main memory. This *Set* strategy is used as the reference providing the best solution, but is not of practical interest in a real system.

5.5.2 Simulation Results

In Figure 5.3, the normalized average backup size with respect to the ideal *Set* backup strategy is shown as a function of the interval size. All strategies shown in Figure 5.3, except *Set* and *Freezer-B8*, use 4096 bits in total. For XBF (xorshift Bloom-Freezer) strategies, three configurations are shown : *XBF-1/4-B1* with a 1/4 ratio and 1 hash keys for the Bloom filter, the *XBF-1/4-B2* which also has a 1/4 ratio but uses 2 keys, and the *XBF-1/8-B1* configuration where the bits are divided with a 1/8 ratio and that uses a single key to access the Bloom filter.

As it can be seen from Figure 5.3, the average backup size of the hybrid strategies is much closer to that of a perfect *Set* strategy. On the other hand, using Freezer with 4096 bits (*Freezer* in Figure 5.3) results in a significant increase in the backup size, which gets between 2.3× and 1.7× that of the perfect *Set* strategy.

Moreover, the backup size of hybrid XBF strategies are between 2.2× and 1.8× smaller, depending on the interval size, with respect to that of Freezer strategy, with the same number of available bits (4096).

To show the higher efficiency of hybrid strategies over Freezer, Figure 5.3 also shows a Freezer configuration that tracks blocks of 8 words (*Freezer-B8*). However, this version

would require 32K bits (as the memory space is 1MB) instead of 4096 bits for the other strategies, making this approach much more expensive. Even with this 4× bigger bitmap, the hybrid strategies still outperform the simple Freezer approach.

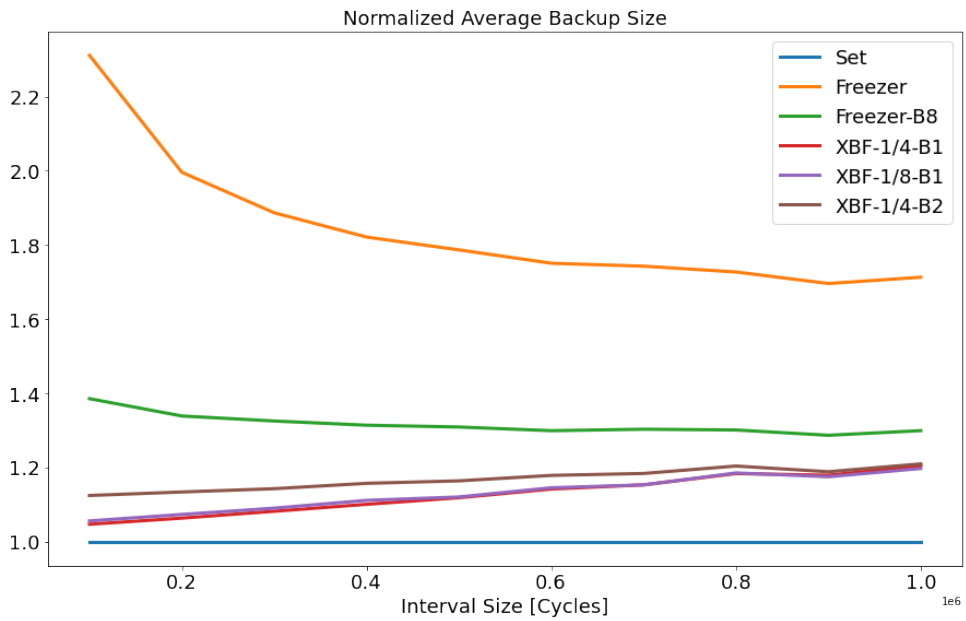


FIGURE 5.3 – Average backup size as a function of the interval size, normalized with respect to the ideal *Set* backup strategy.

5.5.3 Hardware Model

To provide an overview of the potential hardware requirements of implementing our hybrid approach (*Bloom-Freezer* module), an unoptimized high-level synthesis model has been implemented and synthesized. Additionally, a similar model of a Freezer approach (*Freezer* module) has also been implemented and synthesized to provide a comparison. Both models have been synthesized using Cadence Stratus HLS, with the default 45nm technology node available with the tool.

Figures 5.5 and 5.4 show the tree map view of the hardware resources of the Freezer and Bloom-Freezer modules after the high-level synthesis process, respectively. In this view, the size of the components is proportional to the expected area. This view only gives a high-level approximation of the area, as the final result in terms of area and resource utilization can be changed (normally improved) by the logic synthesis step, and finally

during the physical implementation phase. However, the figures are useful as they give an overview of the area breakdown for each component and allow us to compare the two implementations. As it can be seen, for both modules, the bulk of the area is occupied by the bit-array used for the bitmap in case of the *Freezer* module, and for both the Freezer-bitmap and the Bloom filter in the *Bloom-Freezer* module. The rest of the area is divided between few registers, mainly the counters for looping through the dirty words, and the register to output the SRAM addresses, and few other resources and multiplexers. The control logic is also indicated in the figure but it only occupies a small portion of the area.

As it can be seen from Figures 5.5 and 5.4, the area of the two modules is mainly dependent on the total number of bits. However, due to the separation of the bits into two arrays for the *Bloom-Freezer* module, there is a small overhead. Additionally, the Bloom-Freezer module also requires few more counters due to the slightly more complicated looping algorithm. The Bloom-Freezer module area also depends on the number of read and write ports added to the Bloom bit-array. In fact, depending on how-many bits are read and written in the Bloom filter for each insertion/query, an additional read and write port is necessary.

An alternative to increasing the number of ports could be to serialize the accesses, introducing an overhead in the latency of each insertion and query operation, and ultimately impacting the throughput. A different approach could be to divide the bit-array into multiples banks to allow multiple parallel reads and writes, this however would require collision resolution logic for the cases when a query or a write needs to access two locations in the same bank.

As it can be expected, the area of both the *Freezer* and *Bloom-Freezer* modules scales proportionally to the overall bit number of the module. In Figures 5.6 and 5.7, the combinational and sequential area breakdown, and the scaling with respect of the total number of bits is shown, for the Freezer and Bloom-Freezer modules, respectively. Figure 5.7 also shows different configurations with the same total number of bits, but with a different ratio between the direct mapped array (the freezer part) and the bloom filter array. In Figure 5.7, the name of the configuration *L_BloomFreezer_X_Y_18_LOWP* refers to the logic synthesis configuration, with X bits for the bitmap, Y bits for the Bloom filter and 18 bits for the address. *LOWP* refers to the low-power synthesis options. The different bit array organizations tested were :

— 1/2 split between the direct mapped part and the Bloom filter

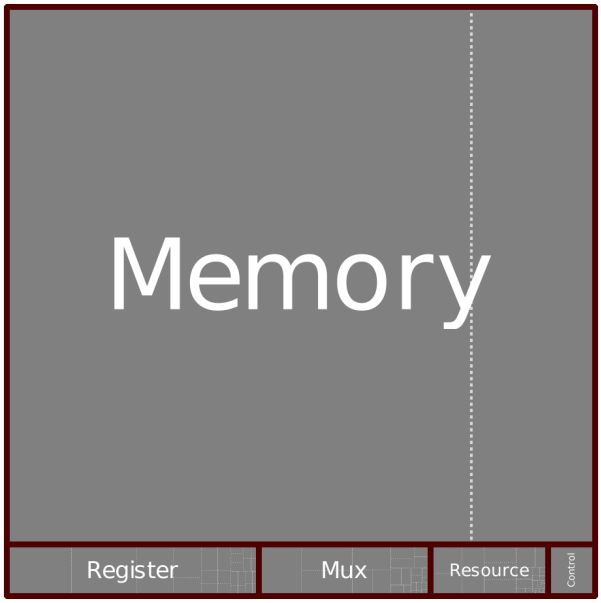


FIGURE 5.4 – Tree-map view of the *Bloom-Freezer* module.

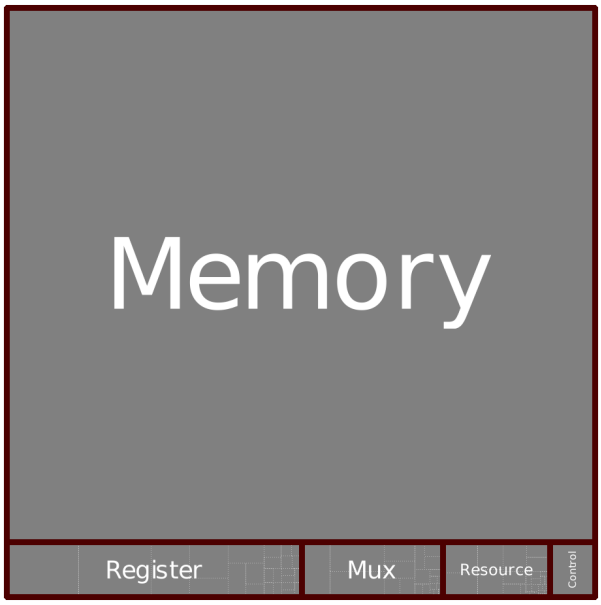


FIGURE 5.5 – Tree-map view of the *Freezer* module.

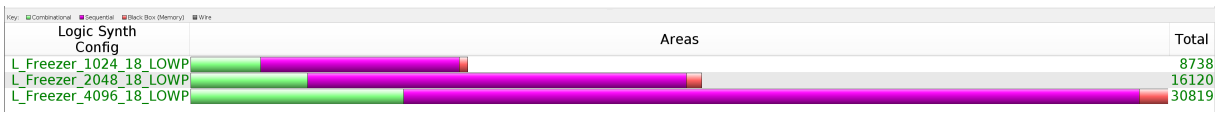


FIGURE 5.6 – *Freezer* module area breakdown after logic synthesis.

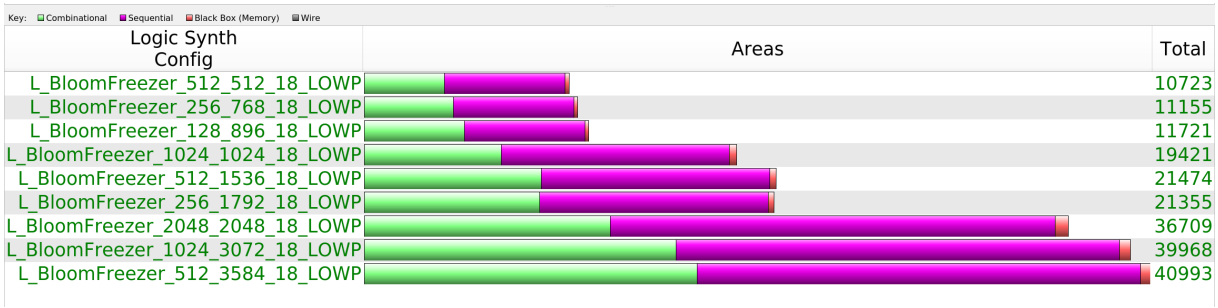


FIGURE 5.7 – *Bloom-Freezer* module area breakdown after logic synthesis.

— 1/4 of the bits for the direct mapped array, and 3/4 for the Bloom filter
— 1/8 of the bits for the direct mapped array, and 7/8 for the Bloom filter

As it can be seen in Figure 5.7, there is a slight overhead when the number of bits is not divided in half between the direct mapped and the Bloom arrays. The data reported in Figure 5.7 and Figure 5.6 are taken after the logic synthesis step, executed with the Genus synthesis tool from Cadence. The area and component reports after the logic synthesis step show an improvement with respect to the approximated results reported after the high-level synthesis step only. For the logic synthesis, the *low_power* options were enabled.

The summary result of the logic synthesis for the *Bloom-Freezer* modules, with different direct map and Bloom bit-array size (*FBits* and *BBits*, respectively) are reported in Table 5.2. Table 5.1 instead reports the logic synthesis data for three configurations of the *Freezer* module. As it can be seen from the two tables, there is an overhead with the *Bloom-Freezer* modules with respect to a *Freezer* module with the same number of

TABLE 5.1 – Logic synthesis results summary for the *Freezer* module.

Bits	Combinational Area	Sequential Area	# FFs	Total Area	Total Power
1024	2,203	6,262	1,143	8,738	0.06
2048	3,689	11,937	2,181	16,120	0.08
4096	6,715	23,181	4,235	30,819	0.12

Table 5.2 – Logic synthesis results summary for the *Bloom-Freezer* module.

FBits	BBits	Combinational Area	Sequential Area	# FFs	Total Area	Total Power
512	512	4,908	6,300	1,150	11,462	0.08
256	768	5,687	6,300	1,150	12,196	0.08
128	896	6,677	6,301	1,151	13,180	0.07
1024	1024	8,531	11,898	2,173	20,819	0.09
512	1536	11,076	11,904	2,174	23,311	0.11
256	1792	11,632	11,904	2,174	23,848	0.11
2048	2048	15,564	23,179	4,235	39,451	0.14
1024	3072	20,481	23,106	4,222	44,184	0.39
512	3584	22,416	23,111	4,223	46,073	0.48

bits. As an example, the 4096-bit configuration of Freezer is $\sim$ 22% smaller, after logic synthesis, w.r.t. the Bloom-Freezer module with 2048 *FBits* and 2048 *BBits*, even if the number of flip-flops is the same for both modules. The area difference is mainly due to the additional circuitry required to address two separate register banks. In fact, as shown in Tables 5.2 and 5.1, the sequential are of both modules is almost the same, while the combinational area of the *Bloom-Freezer* module is 2.3$\times$ larger.

This overhead could be reduced by implementing the *Bloom-Freezer* module with a unified memory block or register bank, shared between the Bloom filter and the bitmap. This way, the multiplexing and decoding circuitry would be equivalent to that of the *Freezer* module, but it would not be possible to access in parallel both the bitmap and Bloom filter arrays.

Overall, we consider the area overhead introduced by *Bloom-Freezer* a small price to pay for the reduced backup size, when dealing with large main memories. Moreover, this area increase could be reduced even more, as neither the *Bloom-Freezer* nor the *Freezer* modules presented in this Chapter, are optimized and complete designs. As an example, both modules lack the implementation of the restore state machine, which would be the same for both modules, and would reduce the difference in percentage of area between the two designs.

5.6 Conclusion

Intermittently-powered systems need to preserve the state of the system, even when facing unpredictable power failures. In the previous chapters, we presented techniques for

on-demand differential backup using a bitmap to track modified sections of memory. The bitmap approach works best when the size of each memory section is relatively small (like 8-word sections). However, this is difficult to scale to larger memory sizes, as the size of the bitmap would also increase significantly.

To overcome this limitation, this chapter has explored the use of Bloom filters, in combination with bitmaps, to track larger memories, with a fine granularity, but without requiring a large number of bits. The approach that leads to the best results is the *Bloom-Freezer* approach (*BF*), that makes use of a bitmap to track large memory sections (i.e., pages), combined with a single Bloom filter used to track the single memory words. This approach limits the collisions and the search space of the Bloom filter to only the memory sections marked in the bitmap. The technique was able to achieve, in our simulations, significantly better results than the *Freezer* approach described in Chapter 3, with the same number of bits. With the *BF* strategy resulting in a backup size that is between 1.8× and 2.2× as small as that of the *Freezer* approach, depending on the length of interval between two consecutive backups.

This chapter also presents an unoptimized hardware implementation of both the *BF* and the *Freezer* strategies, realized using the same synthesis tools and with the same technology library. The result of the hardware synthesis shows that the additional complexity introduced by the *BF* algorithm does result in an increased area (e.g., *BF* is 22% larger then *Freezer* with 4096 bits), however the area is overall still dominated by the register banks, and thus by the number of bits.

In summary, this chapter explores a path for the optimization of on-demand differential backup, for intermittently-powered systems. The proposed strategy could enable the use of more capable systems with larger main memory, in transiently-powered scenarios. While this is an early exploration, the use of approximate set membership data structures seems to be a promising route to achieve fine granularity memory writes tracking, without a large overhead. Possible future explorations might include the use of a more complex hierarchy for the dirty memory tracking, and the use of more complex data structures such as Cuckoo filters or more advanced variations of a Bloom filter [Fan+14 ; GL20].

Conclusions

The emergence of environmental energy harvesting as a source to power wireless sensor nodes, Internet-of-Things (IoT) and other embedded devices, together with the emergence of new Non-Volatile Memory (NVM) technologies, have driven the interest towards Non-Volatile Computers (NVC) and Non-Volatile Processors (NVP) for intermittent computing scenarios. NVPs are attractive for transiently-powered applications because of their ability to quickly save the state of the system in a non-volatile fashion. This allows to preserve forward progress, and carry on the computation despite the occurrence of unexpected power failures.

In the literature both hardware and software solutions have been proposed, presenting NVPs based on different NVM technologies, as well as proposing techniques to allow transiently powered computing on NVM equipped devices. Software-based solutions are usually easier to adopt and implement, they can consist of static compile-time techniques, and/or of run-time systems that handle the backup and restore procedures. However, software-level solutions are usually slower and less energy efficient. Hardware-based solutions use NVM integration, with components such as hybrid non-volatile flip-flops (nvFF), to implement fully non-volatile processors, with the objective to minimize backup and restore times. However, this approach usually comes with large overhead in terms of chip area, they are difficult to implement due to the use of exotic technologies and intrusive micro-architectural modifications, and offer no protection against consistency errors. Moreover, NVM-only solutions have the limitations of current NVM technologies, which have lower endurance, worse performance in terms of execution speed, and worse active energy consumption with respect to SRAM. As a first contributions, this work presents a review of the main emerging NVM technologies, where the strength and weaknesses of each are compared using data collected from various works and surveys from the literature.

The different check-pointing strategy presented in the literature, can be categorized as either *on-demand* or *static*. Static check-points are usually placed at fixed location in the code, with some static compile-time techniques. With on-demand check-pointing, the system saves the state in reaction to an imminent power failure event, which is usually triggered by the voltage across the buffer capacitor falling below a threshold. On-demand

check-pointing is less wasteful than static check-pointing as it only saves when necessary. Moreover, it avoids the significant cost of roll-backs as the application normally restarts from where it was interrupted. However, the existing on-demand solutions do not optimize the size of the backup, resulting in wasteful full-memory backups, when only a portion of the memory could be saved instead.

To address these limitations, in this work, we propose Freezer, a backup controller scheme that implements an on-demand differential incremental backup. To achieve that, Freezer tracks the memory writes at run-time, marking the modified memory regions in an internal bitmap. Then, at backup time, the modified memory regions are copied by Freezer to the corresponding location in the NVM. Freezer also takes care of restoring the full content of the SRAM by copying it back from the NVM, when the restore process is executed. Contrary to other more intrusive techniques, Freezer does not introduce any major penalty at run time, as it only spies the memory writes, without ever accessing the memory or interfering with the rest of the system. This allows for minimal overhead, with the power consumed by Freezer at run-time that can be as low as 0.82% of the total system power when running the application. This power efficiency means that almost all the available energy is used for progressing the computation. Moreover, Freezer also incurs in a relatively small area overhead, $\approx 0.4\%$, when compared with the total area of a wireless sensor node SoC. Finally, Freezer achieves a 87.7% average reduction in the backup size, and results in a two orders of magnitude reduction in the total backup-time when compared with software-based, state-of-the-art solutions. When compared with architecture based only on NVM, Freezer shows a potential reduction in the memory access energy of 83.0% and 76.3%, depending on the NVM technology.

One problem that intermittently-powered system have to face when using NVMs for state retention, is to keep the state consistent. Consistency errors can arise after a backup, if the NVM is updated, and then the power is lost before the next backup can be executed, leaving the NVM, and thus the system, in an inconsistent state. This error can happen during the backup process itself, if the backup execution is interrupted, leaving a corrupt backup in the NVM. In the literature, *double-buffering* of the backup is often proposed to solve potential corruption issues. However, double buffering is normally applied to full-memory backups, which are not an optimal solution. To address this issue, without loosing the advantage of incremental backups, in this work, we propose two robust incremental backup algorithms. The proposed algorithms similarly use a bitmap to track

modified memory regions, but they guarantee, at any time, the existence of a consistent state that can be restored in case of errors. Additionally, the loss of progress due to an error can be limited to the last correct backup before the error occurred. Alternatively, one of the proposed algorithm allows to trade off the loss of progress (*i.e.*, restoring to an even earlier backup), with a reduction in the number of NVM writes performed by the algorithm. Both our proposed algorithms achieve an average reduction in the total energy and in the total run-time of more than 23%, when compared to a *double-buffering* approach.

Finally, we looked at extending to larger memory capacity the approach of a backup controller, without sacrificing the granularity of the incremental backup, and without requiring huge bitmaps. To address this problem, this work presents a method to use Bloom filters, in conjunction with a bitmap, to implement a fine granularity dirty memory tracking. Our simulations show that the proposed hybrid Bloom-Freezer approach is able to track modifications, in a $1MB$ address space, achieving, on average, a backup sizes which is half the backup size of a comparable Freezer approach. Moreover, the hybrid Bloom-Freezer approach also achieves a $1.4\times$ smaller backup size when compared to a $8\times$ larger Freezer.

Perspectives

The solutions proposed in this dissertation revolve around the idea of having a dedicated hardware backup controller that keeps tracks of changes and executes the backup and restore operations when needed. This approach departs from the more common hardware solutions, which usually focus on modifications at the circuit and device levels, to inject non-volatile elements in the design. We believe that having a dedicated controller brings several advantages, as it opens the possibility to integrate this type of solution in any System-on-Chip (SoC), regardless of the type of the processor or the particular technology of the NVM used for backup.

An interesting extension that could be introduced in a dedicated backup controller, is to add integrity checks, such as Cyclic-Redundancy-Checks sums (CRCs), to the backup process. In an architecture such as Freezer, this could be easily added, either at the whole backup level or per memory block. As an example, Freezer could compute a CRC or a

rolling hash of all the memory words while they are being moved from SRAM to the NVM, and then it could store this CRC as the last word of the backup process. Alternatively, a CRC or a parity code could be added at the block level (*e.g.*, one parity word for every 8 words block) during the backup, and then verified during the restore, making the backup fault tolerant.

In a similar way, encryption could be introduced during the backup process, to protect the data at rest, for those applications requiring confidentiality.

A dedicated hardware block could also expose some functionalities to the software, via memory mapped registers. This would allow to develop new software solutions based around the basic functionalities of the backup controller, such as the memory tracking and the hardware managed data transfer. One example could be to allow the software, maybe in the form of an operating system (OS), to clear some of the memory regions that it knows are no longer needed. This would further reduce the cost and size of the backups.

Bibliographie

[Ovs68] Stanford R. Ovshinsky, « Reversible Electrical Switching Phenomena in Disordered Structures », in : *Physical Review Letters* 21.*20* (nov. 1968), Publisher : American Physical Society, p. 1450-1453, doi : 10.1103/PhysRevLett.21.1450, url : https://link.aps.org/doi/10.1103/PhysRevLett.21.1450 (visité le 19/01/2021).

[Blo70] Burton H. Bloom, « Space/time trade-offs in hash coding with allowable errors », in : *Communications of the ACM* 13.*7* (juill. 1970), p. 422-426, issn : 0001-0782, doi : 10.1145/362686.362692, url : https://doi.org/10.1145/362686.362692 (visité le 26/02/2021).

[Chu71] L. Chua, « Memristor-The missing circuit element », in : *IEEE Transactions on Circuit Theory* 18.*5* (sept. 1971), Conference Name : IEEE Transactions on Circuit Theory, p. 507-519, issn : 2374-9555, doi : 10.1109/TCT.1971.1083337.

[Moo+95] J. S. Moodera et al., « Large Magnetoresistance at Room Temperature in Ferromagnetic Thin Film Tunnel Junctions », in : *Physical Review Letters* 74.*16* (avr. 1995), Publisher : American Physical Society, p. 3273-3276, doi : 10.1103/PhysRevLett.74.3273, url : https://link.aps.org/doi/10.1103/PhysRevLett.74.3273 (visité le 26/01/2021).

[Dur+03] M. Durlam et al., « A 0.18 μm 4Mb toggling MRAM », en, in : *IEEE International Electron Devices Meeting 2003*, Washington, DC, USA : IEEE, 2003, p. 34.6.1-34.6.3, isbn : 978-0-7803-7872-8, doi : 10.1109/IEDM.2003.1269448, url : http://ieeexplore.ieee.org/document/1269448/ (visité le 08/07/2022).

[Mar03] George Marsaglia, « Xorshift RNGs », en, in : *Journal of Statistical Software* 8 (juill. 2003), p. 1-6, issn : 1548-7660, doi : 10.18637/jss.v008.i14, url : https://doi.org/10.18637/jss.v008.i14 (visité le 06/03/2022).

[Eng+05] B. N. Engel et al., « A 4-Mb toggle MRAM based on a novel bit and switching method », in : *IEEE Transactions on Magnetics* 41.*1* (jan. 2005), Conference Name : IEEE Transactions on Magnetics, p. 132-136, issn : 1941-0069, doi : 10.1109/TMAG.2004.840847.

[Sor+07] Jacob Sorber et al., « Eon : A Language and Runtime System for Perpetual Systems », in : *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, New York, NY, USA : ACM, 2007, p. 161-174, isbn : 978-1-59593-763-6, doi : 10.1145/1322263.1322279, url : http://doi.acm.org/10.1145/1322263.1322279 (visité le 06/07/2018).

[Str+08] Dmitri B. Strukov et al., « The missing memristor found », en, in : *Nature* 453.*7191* (mai 2008), Number : 7191 Publisher : Nature Publishing Group, p. 80-83, issn : 1476-4687, doi : 10.1038/nature06932, url : https://www.nature.com/articles/nature06932 (visité le 01/02/2021).

[Der+09] DerChang Kau et al., « A stackable cross point Phase Change Memory », in : *2009 IEEE International Electron Devices Meeting (IEDM)*, ISSN : 2156-017X, déc. 2009, p. 1-4, doi : 10.1109/IEDM.2009.5424263.

[Lee+09] Benjamin C. Lee et al., « Architecting Phase Change Memory As a Scalable Dram Alternative », in : *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, New York, NY, USA : ACM, 2009, p. 2-13, isbn : 978-1-60558-526-0, doi : 10.1145/1555754.1555758, url : http://doi.acm.org/10.1145/1555754.1555758 (visité le 27/02/2018).

[AS10] H. Akinaga et H. Shima, « Resistive Random Access Memory (ReRAM) Based on Metal Oxides », in : *Proceedings of the IEEE* 98.*12* (déc. 2010), Conference Name : Proceedings of the IEEE, p. 2237-2251, issn : 1558-2256, doi : 10.1109/JPROC.2010.2070830.

[Bur+10] Geoffrey W. Burr et al., « Phase change memory technology », en, in : *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics : Materials, Processing, Measurement, and Phenomena* 28.*2* (mars 2010), Publisher : American Vacuum SocietyAVS, p. 223, issn : 2166-2746, doi : 10.1116/1.3301579, url : https://avs.scitation.org/doi/abs/10.1116/1.3301579 (visité le 19/01/2021).

[GP10] S. Gerardin et A. Paccagnella, « Present and Future Non-Volatile Memories for Space », in : *IEEE Transactions on Nuclear Science* 57.*6* (déc. 2010), Conference Name : IEEE Transactions on Nuclear Science, p. 3016-3039, issn : 1558-1578, doi : 10.1109/TNS.2010.2084101.

[Vul+10] Rudd J.M. Vullers et al., « Energy Harvesting for Autonomous Wireless Sensor Networks », in : *IEEE Solid-State Circuits Magazine* 2.*2* (2010), Conference Name : IEEE Solid-State Circuits Magazine, p. 29-38, issn : 1943-0590, doi : 10.1109/MSSC.2010.936667.

[GM11] Michael T. Goodrich et Michael Mitzenmacher, « Invertible bloom lookup tables », in : *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, sept. 2011, p. 792-799, doi : 10.1109/Allerton.2011.6120248.

[RSF11] Benjamin Ransford, Jacob Sorber et Kevin Fu, « Mementos : System Support for Long-running Computation on RFID-scale Devices », in : *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, New York, NY, USA : ACM, 2011, p. 159-170, isbn : 978-1-4503-0266-1, doi : 10.1145/1950365.1950386, url : http://doi.acm.org/10.1145/1950365.1950386 (visité le 04/06/2018).

[Val+11] Ilia Valov et al., « Electrochemical metallization memories—fundamentals, applications, prospects », en, in : *Nanotechnology* 22.*25* (mai 2011), Publisher : IOP Publishing, p. 254003, issn : 0957-4484, doi : 10.1088/0957-4484/22/25/254003, url : https://doi.org/10.1088/0957-4484/22/25/254003 (visité le 03/02/2021).

[Xue+11] Chun Jason Xue et al., « Emerging Non-volatile Memories : Opportunities and Challenges », in : *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, New York, NY, USA : ACM, 2011, p. 325-334, isbn : 978-1-4503-0715-4, doi : 10.1145/2039370.2039420, url : http://doi.acm.org/10.1145/2039370.2039420 (visité le 06/07/2018).

[Yu+11] W. k Yu et al., « A non-volatile microcontroller with integrated floating-gate transistors », in : *2011 IEEE/IFIP 41st International Conference on*

Dependable Systems and Networks Workshops (DSN-W), juin 2011, p. 75-80, DOI : 10.1109/DSNW.2011.5958839.

[Zwe+11] M. ZWERG et al., « An 82 μA/MHz microcontroller with embedded FeRAM for energy-harvesting applications », in : *2011 IEEE International Solid-State Circuits Conference*, fév. 2011, p. 334-336, DOI : 10.1109/ISSCC.2011.5746342.

[Don+12] X. DONG et al., « NVSim : A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7 (juill. 2012), Conference Name : IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, p. 994-1007, ISSN : 1937-4151, DOI : 10.1109/TCAD.2012.2185930.

[MSS12] A. MAKAROV, V. SVERDLOV et S. SELBERHERR, « Emerging memory technologies : Trends, challenges, and modeling methods », en, in : *Microelectronics Reliability* 52.4 (avr. 2012), p. 628-634, ISSN : 00262714, DOI : 10.1016/j.microrel.2011.10.020, URL : https://linkinghub.elsevier.com/retrieve/pii/S0026271411004859 (visité le 16/06/2020).

[Sla+12] J. M. SLAUGHTER et al., « High density ST-MRAM technology (Invited) », in : *2012 International Electron Devices Meeting*, ISSN : 2156-017X, déc. 2012, p. 29.3.1-29.3.4, DOI : 10.1109/IEDM.2012.6479128.

[Wan+12] Y. WANG et al., « A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops », in : *2012 Proceedings of the ESSCIRC (ESSCIRC)*, sept. 2012, p. 149-152, DOI : 10.1109/ESSCIRC.2012.6341281.

[And+13] T. ANDRE et al., « ST-MRAM fundamentals, challenges, and applications », in : *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*, ISSN : 2152-3630, sept. 2013, p. 1-8, DOI : 10.1109/CICC.2013.6658449.

[Bar+13] S. C. BARTLING et al., « An 8MHz 75 μA/MHz zero-leakage non-volatile logic-based Cortex-M0 MCU SoC exhibiting 100% digital state retention at VDD=0V with lt ;400ns wakeup and sleep transitions », in : *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, fév. 2013, p. 432-433, DOI : 10.1109/ISSCC.2013.6487802.

[Fuj13] Yoshihisa Fujisaki, « Review of Emerging New Solid-State Non-Volatile Memories », en, in : *Japanese Journal of Applied Physics* 52.*4R* (avr. 2013), p. 040001, issn : 1347-4065, doi : 10.7567/JJAP.52.040001, url : https://iopscience.iop.org/article/10.7567/JJAP.52.040001/meta (visité le 06/03/2019).

[Liu+13] T. Liu et al., « A 130.7mm2 2-layer 32Gb ReRAM memory device in 24nm technology », in : *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSN : 2376-8606, fév. 2013, p. 210-211, doi : 10.1109/ISSCC.2013.6487703.

[Bri+14] A. van den Brink et al., « Spin-Hall-assisted magnetic random access memory », en, in : *Applied Physics Letters* 104.*1* (jan. 2014), p. 012403, issn : 0003-6951, 1077-3118, doi : 10.1063/1.4858465, url : http://aip.scitation.org/doi/10.1063/1.4858465 (visité le 29/01/2021).

[Fac+14] R. Fackenthal et al., « 19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology », in : *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, ISSN : 2376-8606, fév. 2014, p. 338-339, doi : 10.1109/ISSCC.2014.6757460.

[Fan+14] Bin Fan et al., « Cuckoo Filter : Practically Better Than Bloom », in : *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, CoNEXT '14, New York, NY, USA : Association for Computing Machinery, déc. 2014, p. 75-88, isbn : 978-1-4503-3279-8, doi : 10.1145/2674005.2674994, url : https://doi.org/10.1145/2674005.2674994 (visité le 25/01/2021).

[HAW14] Seungbum Hong, Orlando Auciello et Dirk Wouters, éd., *Emerging Non-Volatile Memories*, en, Springer US, 2014, isbn : 978-1-4899-7536-2, doi : 10.1007/978-1-4899-7537-9, url : https://www.springer.com/gp/book/9781489975362 (visité le 21/01/2021).

[Kha+14] S. Khanna et al., « An FRAM-Based Nonvolatile Logic MCU SoC Exhibiting 100% Digital State Retention at $VDD = 0V$ Achieving Zero Leakage With $<$ 400-ns Wakeup Time for ULP Applications », in : *IEEE Journal of Solid-State Circuits* 49.*1* (jan. 2014), p. 95-106, issn : 0018-9200, doi : 10.1109/JSSC.2013.2284367.

[LHE14] J. K. Jerry Lee, Amr Haggag et William Eklow, « Protecting against emerging vmin failures in advanced technology nodes », in : *2014 International Test Conference*, ISSN : 2378-2250, oct. 2014, p. 1-7, doi : 10.1109/TEST.2014.7035278.

[RL14] Benjamin Ransford et Brandon Lucia, « Nonvolatile memory is a broken time machine », in : *Proceedings of the workshop on Memory Systems Performance and Correctness*, MSPC '14, Edinburgh, United Kingdom : Association for Computing Machinery, juin 2014, p. 1-3, isbn : 978-1-4503-2917-0, doi : 10.1145/2618128.2618136, url : https://doi.org/10.1145/2618128.2618136 (visité le 06/03/2020).

[Sak+14] N. Sakimura et al., « 10.5 A 90nm 20MHz fully nonvolatile microcontroller for standby-power-critical applications », in : *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, fév. 2014, p. 184-185, doi : 10.1109/ISSCC.2014.6757392.

[Bal+15] D. Balsamo et al., « Hibernus : Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems », in : *IEEE Embedded Systems Letters* 7.*1* (mars 2015), p. 15-18, issn : 1943-0663, doi : 10.1109/LES.2014.2371494.

[Jay+15] Hrishikesh Jayakumar et al., « QuickRecall : A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers », in : *J. Emerg. Technol. Comput. Syst.* 12.*1* (août 2015), 8 :1-8 :19, issn : 1550-4832, doi : 10.1145/2700249, url : http://doi.acm.org/10.1145/2700249 (visité le 15/11/2018).

[JBT15] Bojan Jovanović, Raphael M. Brum et Lionel Torres, « Comparative Analysis of MTJ/CMOS Hybrid Cells Based on TAS and In-Plane STT Magnetic Tunnel Junctions », in : *IEEE Transactions on Magnetics* 51.*2* (fév. 2015), Conference Name : IEEE Transactions on Magnetics, p. 1-11, issn : 1941-0069, doi : 10.1109/TMAG.2014.2347009.

[Lee+15] A. Lee et al., « RRAM-based 7T1R nonvolatile SRAM with 2x reduction in store energy and 94x reduction in restore energy for frequent-off instant-on applications », in : *2015 Symposium on VLSI Circuits (VLSI Circuits)*, juin 2015, p. C76-C77, doi : 10.1109/VLSIC.2015.7231368.

[Ma+15] Kaisheng Ma et al., « Architecture exploration for ambient energy harvesting nonvolatile processors », in : *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, ISSN : 2378-203X, fév. 2015, p. 526-537, DOI : 10.1109/HPCA.2015.7056060.

[Xie+15] Mimi Xie et al., « Fixing the broken time machine : consistency-aware checkpointing for energy harvesting powered non-volatile processor », in : *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, San Francisco, California : Association for Computing Machinery, juin 2015, p. 1-6, ISBN : 978-1-4503-3520-1, DOI : 10.1145/2744769.2744842, URL : https://doi.org/10.1145/2744769.2744842 (visité le 15/07/2020).

[Zha+15] M. Zhao et al., « Software assisted non-volatile register reduction for energy harvesting based cyber-physical system », in : *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, mars 2015, p. 567-572.

[Bal+16] D. Balsamo et al., « Hibernus #x002B ; #x002B ; : A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.*12* (2016), p. 1968-1980, ISSN : 0278-0070, DOI : 10.1109/TCAD.2016.2547919.

[Che16] An Chen, « A review of emerging non-volatile memory (NVM) technologies and applications », in : *Solid-State Electronics*, Extended papers selected from ESSDERC 2015 125 (nov. 2016), p. 25-38, ISSN : 0038-1101, DOI : 10.1016/j.sse.2016.07.006, URL : http://www.sciencedirect.com/science/article/pii/S0038110116300867 (visité le 05/03/2019).

[Chi+16] T. K. Chien et al., « Low-Power MCU With Embedded ReRAM Buffers as Sensor Hub for IoT Applications », in : *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.*2* (juin 2016), p. 247-257, ISSN : 2156-3357, DOI : 10.1109/JETCAS.2016.2547778.

[CL16] Alexei Colin et Brandon Lucia, « Chain : Tasks and Channels for Reliable Intermittent Programs », in : *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, New York, NY, USA : ACM, 2016, p. 514-530, ISBN : 978-1-4503-4444-9, DOI : 10.1145/2983990.2983995, URL : http://doi.acm.org/10.1145/2983990.2983995 (visité le 06/07/2018).

[LJ16] Q. Liu et 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)*, août 2016, p. 1-6, doi : 10.1109/NVMSA.2016.7547183.

[Liu+16] Y. Liu et al., « 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, p. 84-86, doi : 10.1109/ISSCC.2016.7417918.

[Ma+16] Kaisheng Ma et al., « Nonvolatile Processor Architectures : Efficient, Reliable Progress with Unstable Power », in : *IEEE Micro* 36.*3* (mai 2016), p. 72-83, issn : 1937-4143, doi : 10.1109/MM.2016.35.

[Sen+16a] S. Senni et al., « Exploring MRAM Technologies for Energy Efficient Systems-On-Chip », in : *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.*3* (sept. 2016), p. 279-292, issn : 2156-3357, doi : 10.1109/JETCAS.2016.2547680.

[Sen+16b] Sophiane Senni et al., « Non-Volatile Processor Based on MRAM for Ultra-Low-Power IoT Devices », in : *ACM Journal on Emerging Technologies in Computing Systems* 13.*2* (déc. 2016), 17 :1-17 :23, issn : 1550-4832, doi : 10.1145/3001936, url : https://doi.org/10.1145/3001936 (visité le 02/12/2020).

[YC16] S. Yu et P. Y. Chen, « Emerging Memory Technologies : Recent Trends and Prospects », in : *IEEE Solid-State Circuits Magazine* 8.*2* (2016), p. 43-56, issn : 1943-0582, doi : 10.1109/MSSC.2016.2546199.

[Yu16] Shimeng Yu, « Resistive Random Access Memory (RRAM) », en, in : *Synthesis Lectures on Emerging Engineering Technologies* 2.*5* (mars 2016), p. 1-79, issn : 2381-1412, 2381-1439, doi : 10.2200/S00681ED1V01Y201510EET006, url : http://www.morganclaypool.com/doi/10.2200/S00681ED1V01Y201510EET006 (visité le 01/02/2021).

[And+17] Koji Ando et al., « Non-volatile Memories », in : *Normally-Off Computing*, sous la dir. de Takashi Nakada et Hiroshi Nakamura, Tokyo : Springer Japan, 2017, p. 27-55, isbn : 978-4-431-56505-5, doi : 10.1007/978-4-431-56505-5_3, url : https://doi.org/10.1007/978-4-431-56505-5_3.

[Bou+17] Jalil Boukhobza et al., « Emerging NVM : A Survey on Architectural Integration and Research Challenges », in : *ACM Trans. Des. Autom. Electron. Syst.* 23.*2* (nov. 2017), 14 :1-14 :32, ISSN : 1084-4309, DOI : 10.1145/3131848, URL : http://doi.acm.org/10.1145/3131848 (visité le 06/07/2018).

[Che+17] Hari Cherupalli et al., « Determining Application-specific Peak Power and Energy Requirements for Ultra-low Power Processors », in : *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, New York, NY, USA : ACM, 2017, p. 3-16, ISBN : 978-1-4503-4465-4, DOI : 10.1145/3037697.3037711, URL : http://doi.acm.org/10.1145/3037697.3037711 (visité le 05/03/2018).

[GGK17] Zahra Ghodsi, Siddharth Garg et Ramesh Karri, « Optimal Checkpointing for Secure Intermittently-powered IoT Devices », in : *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, event-place : Irvine, California, Piscataway, NJ, USA : IEEE Press, 2017, p. 376-383, URL : http://dl.acm.org/citation.cfm?id=3199700.3199750 (visité le 12/12/2019).

[Hag+17] P. A. Hager et al., « A scan-chain based state retention methodology for IoT processors operating on intermittent energy », in : *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, mars 2017, p. 1171-1176, DOI : 10.23919/DATE.2017.7927166.

[Jay+17] Hrishikesh Jayakumar et al., « Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices », in : *ACM Trans. Embed. Comput. Syst.* 16.*3* (avr. 2017), 65 :1-65 :23, ISSN : 1539-9087, DOI : 10.1145/2983628, URL : http://doi.acm.org/10.1145/2983628.

[LC17] Hai Li et Yiran Chen, *Nonvolatile Memory Design : Magnetic, Resistive, and Phase Change*, en, Google-Books-ID : rAlEDwAAQBAJ, CRC Press, déc. 2017, ISBN : 978-1-351-83419-3.

[Su+17a] F. Su et al., « 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 Circuits*, juin 2017, p. C260-C261, DOI : 10.23919/VLSIC.2017.8008585.

[Su+17b] F. Su et al., « A Ferroelectric Nonvolatile Processor with 46 μs System-Level Wake-up Time and 14 μs Sleep Time for Energy Harvesting Applications », in : *IEEE Transactions on Circuits and Systems I : Regular Papers* 64.*3* (mars 2017), p. 596-607, ISSN : 1549-8328, DOI : 10.1109/TCSI.2016.2616905.

[Wan+17] Z. Wang et al., « A 130nm FeRAM-based parallel recovery nonvolatile SOC for normally-OFF operations with 3.9 ×; faster running speed and 11 ×; higher energy efficiency using fast power-on detection and nonvolatile radio controller », in : *2017 Symposium on VLSI Circuits*, juin 2017, p. C336-C337, DOI : 10.23919/VLSIC.2017.8008531.

[Zha+17] Mengying Zhao et al., « Stack-Size Sensitive On-Chip Memory Backup for Self-Powered Nonvolatile Processors », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.*11* (nov. 2017), p. 1804-1816, ISSN : 1937-4151, DOI : 10.1109/TCAD.2017.2666606.

[CYL18a] Patrick Cronin, Chengmo Yang et Yongpan Liu, « A collaborative defense against wear out attacks in non-volatile processors », in : *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA : Association for Computing Machinery, juin 2018, p. 1-6, ISBN : 978-1-4503-5700-5, DOI : 10.1145/3195970.3196825, URL : https://doi.org/10.1145/3195970.3196825 (visité le 06/08/2020).

[CYL18b] Patrick Cronin, Chengmo Yang et Yongpan Liu, « Reliability and Security in Non-volatile Processors, Two Sides of the Same Coin », in : *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, ISSN : 2159-3477, juill. 2018, p. 112-117, DOI : 10.1109/ISVLSI.2018.00030.

[Kis+18] S. Z. Kiss et al., « Bloom Filter with a False Positive Free Zone », in : *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, avr. 2018, p. 1412-1420, DOI : 10.1109/INFOCOM.2018.8486415.

[Agg+19] S. Aggarwal et al., « Demonstration of a Reliable 1 Gb Standalone Spin-Transfer Torque MRAM For Industrial Applications », in : *2019 IEEE International Electron Devices Meeting (IEDM)*, ISSN : 2156-017X, déc. 2019, p. 2.1.1-2.1.4, DOI : 10.1109/IEDM19573.2019.8993516.

[Ber+19] Gautier Berthou et al., « Sytare : A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems », in : *IEEE Transactions on Computers* 68.*9* (sept. 2019), Conference Name : IEEE Transactions on Computers, p. 1390-1403, ISSN : 1557-9956, DOI : 10.1109/TC.2018.2889080.

[Bol+19] D. Bol et al., « 19.6 A 40-to-80MHz Sub-4μW/MHz ULV Cortex-M0 MCU SoC in 28nm FDSOI With Dual-Loop Adaptive Back-Bias Generator for 20μs Wake-Up From Deep Fully Retentive Sleep Mode », in : *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, fév. 2019, p. 322-324, DOI : 10.1109/ISSCC.2019.8662293.

[Cho+19] Jongouk Choi et al., « Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution », in : *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, ISSN : 2642-7346, avr. 2019, p. 331-344, DOI : 10.1109/RTAS.2019.00035.

[Liu+19] Yongpan Liu et al., « A 130-nm Ferroelectric Nonvolatile System-on-Chip With Direct Peripheral Restore Architecture for Transient Computing System », in : *IEEE Journal of Solid-State Circuits* 54.*3* (mars 2019), p. 885-895, ISSN : 1558-173X, DOI : 10.1109/JSSC.2018.2884349.

[Riz+19] Rodrigue Rizk et al., « Demystifying Emerging Nonvolatile Memory Technologies : Understanding Advantages, Challenges, Trends, and Novel Applications », in : *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, ISSN : 2158-1525, mai 2019, p. 1-5, DOI : 10.1109/ISCAS.2019.8702390.

[Rok+19] S. Rokicki et al., « What You Simulate Is What You Synthesize : Designing a Processor Core from C++ Specifications », in : *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, ISSN : 1558-2434, nov. 2019, p. 1-8, DOI : 10.1109/ICCAD45719.2019.8942177.

[Ahm+20] Saad Ahmed et al., « Fast and Energy-Efficient State Checkpointing for Intermittent Computing », in : *ACM Transactions on Embedded Computing Systems* 19.*6* (sept. 2020), 45 :1-45 :27, ISSN : 1539-9087, DOI : 10.1145/3391903, URL : https://doi.org/10.1145/3391903 (visité le 02/04/2021).

[Ber+20] Gautier Berthou et al., « MPU-based incremental checkpointing for transiently-powered systems », in : *2020 23rd Euromicro Conference on Digital System Design (DSD)*, août 2020, p. 89-96, DOI : 10.1109/DSD51259.2020.00025.

[FX20] Julien Frougier et Ruilong Xie, « Multi-level ferroelectric memory cell », US20200295017A1, sept. 2020, URL : https://patents.google.com/patent/US20200295017A1/en (visité le 11/02/2021).

[GL20] Thomas Mueller Graf et Daniel Lemire, « Xor Filters : Faster and Smaller Than Bloom and Cuckoo Filters », in : *ACM Journal of Experimental Algorithmics* 25 (mars 2020), 1.5 :1-1.5 :16, ISSN : 1084-6654, DOI : 10.1145/3376122, URL : https://doi.org/10.1145/3376122 (visité le 25/01/2021).

[PMS20] Davide Pala, Ivan Miro-Panades et Olivier Sentieys, « Freezer : A Specialized NVM Backup Controller for Intermittently-Powered Systems », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), Conference Name : IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, p. 1-1, ISSN : 1937-4151, DOI : 10.1109/TCAD.2020.3025063.

[Ins21] Texas Instruments, *MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers Datasheet*, 2021, URL : https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf?ts=1617007636115.

[Evea] Everspin, *EM016LXQ | Everspin*, URL : https://www.everspin.com/family/em016lxq?npath=3843 (visité le 09/07/2022).

[Eveb] Everspin, *MR5A16A | Everspin*, URL : https://www.everspin.com/family/mr5a16a?npath=258 (visité le 10/02/2021).

[Evec] Everspin, *Toggle MRAM Technology | Everspin*, URL : https://www.everspin.com/toggle-mram-technology (visité le 10/02/2021).

[TDK] TDK, *C2012X5R0G476M125AB : Detailed Information | Capacitors - Multilayer Ceramic Chip Capacitors*, en, URL : https://product.tdk.com/en/search/capacitor/ceramic/mlcc/info?part_no=C2012X5R0G476M125AB (visité le 09/07/2022).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Microarchitectures pour la sauvegarde incrémentale, robuste et efficace dans les systèmes à alimentation intermittente

Mot clés : Microarchitecture, systèmes à alimentation intermittente, récolte d'énergie, basse consommation, ordinateurs non volatiles

Résumé : Les appareils embarqués alimentés par la récupération d'énergie environnementale doivent maintenir le calcul tout en subissant des pannes de courant inattendues. Pour préserver la progression à travers les interruptions de courant, des mémoires non volatiles (NVM) sont utilisées pour enregistrer rapidement l'état. Cette thèse présente d'abord une vue d'ensemble et une comparaison des différentes technologies NVM, basées sur différentes enquêtes de la littérature. La deuxième contribution que nous proposons est un contrôleur de sauvegarde dédié, appelé Freezer, qui implémente un schéma de sauvegarde incrémentielle à la demande. Cela peut réduire la taille de la sauvegarde de $87,7\%$ à celle d'une stratégie de sauvegarde à mémoire complète de l'état de l'art. Notre troisième contribution aborde le problème de la corruption de l'état, due aux interruptions pendant le processus de sauvegarde. Deux algorithmes sont présentés, qui améliorent le processus de sauvegarde incrémentielle de Freezer, le rendant robuste aux erreurs, en garantissant toujours l'existence d'un état correct, qui peut être restauré en cas d'erreurs de sauvegarde. Ces deux algorithmes peuvent consommer 23% d'énergie en moins que la technique de "double-buffering" utilisée dans l'état de l'art. La quatrième contribution porte sur l'évolutivité de notre approche proposée. En combinant Freezer avec des filtres Bloom, nous introduisons un schéma de sauvegarde qui peut couvrir des espaces d'adressage beaucoup plus grands, tout en obtenant une taille de sauvegarde qui est la moitié de la taille de l'approche Freezer habituelle.

Title: Microarchitectures for Robust and Efficient Incremental Backup in Intermittently-Powered Systems

Keywords: Microarchitecture, Intermittently Powered Systems, Energy Harvesting, Low-Power, Non-Volatile Computers

Abstract: Embedded devices powered with environmental energy harvesting, have to sustain computation while experiencing unexpected power failures. To preserve the progress across the power interruptions, Non-Volatile Memories (NVMs) are used to quickly save the state. This dissertation first presents an overview and comparison of different NVM technologies, based on different surveys from the literature. The second contribution we propose is a dedicated backup controller, called Freezer, that implements an on-demand incremental backup scheme. This can make the size of the backup 87.7% smaller then a full-memory backup strategy from the state of the art (SoA). Our third contribution addresses

the problem of corruption of the state, due to interruptions during the backup process. Two algorithms are presented, that improve on the Freezer incremental backup process, making it robust to errors, by always guaranteeing the existence of a correct state, that can be restored in case of backup errors. These two algorithms can consume 23% less energy than the usual double-buffering technique used in the SoA. The fourth contribution, addresses the scalability of our proposed approach. Combining Freezer with Bloom filters, we introduce a backup scheme that can cover much larger address spaces, while achieving a backup size which is half the size of the regular Freezer approach.