The University of Twente

Faculty of Electrical Engineering, Mathematics & Computer Science
Department of Computer Architecture for Embedded Systems

A Lightweight Hardware Architecture For Intermittent Computing

Hiram Rayo Torres Rodriguez

UNIVERSITY
OF TWENTE.

A Lightweight Hardware Architecture For Intermittent Computing

Thesis

Submitted in partial fulfillment of the requirements for the

Master of Science

degree in

Embedded Systems

by

Hiram Rayo Torres Rodriguez
born in Mazatlan, Sinaloa, Mexico.

To be defended publicly on August 28, 2019.

at

The University of Twente

Student number:	1910051	
Submission date:	August 21, 2019	
Thesis Committee:	Prof. Dr. Ir. Marco Bekooij	University of Twente
	Ir. Arvid van den Brink	University of Twente
	Dr. Ir. Przemysław Pawełczak	Delft University of Technology
	Ir. Vito Kortbeek	Delft University of Technology

A Lightweight Hardware Architecture For Intermittent Computing

Author: Hiram Rayo Torres Rodriguez
Email: hrayotorres@gmail.com, h.rayotorresrodriguez@student.tudelft.nl
h.rayotorresrodriguez@student.utwente.nl

Abstract

The pervasiveness nature of IoT devices and recent developments on energy-harvesting technologies have enabled the development of a novel brand of battery-less devices whose operation relies entirely on energy harvested from the environment. Unfortunately, ambient energy leads these devices to experience frequent power failures, making data consistency, and forward progress, one of the biggest challenges to overcome. Most of the research on this domain has focused on software and compiler-based approaches to allow existing off-the-shelf processors to operate intermittently. Nevertheless, most of them present abstractions difficult to use for non-expert programmers, incur in high run-time overhead, or limit programmability features common to low-level programming languages. To address these shortcomings, we propose **Qwark**, a hardware/software solution which utilizes simple yet efficient memory management techniques to transparently solve the problems introduced by intermittency with little extra overhead and no programmer intervention. Qwark continuously tracks memory accesses and uses address translation to isolate harmful accesses, creating a log in an isolated memory segment, effectively preventing the system from reaching inconsistent states due to uncommitted non-volatile memory writes. Furthermore, during run-time, Qwark promotes variable-sized segments of the stack to non-volatile memory to enforce a worst-case checkpoint time, using address translation to link volatile and non-volatile segments, reducing run-time and power consumption. This work also presents the first hardware solution targetting the MSP430 architecture, to provide a fair performance assessment against current software-based solutions. Qwark is evaluated running a set of benchmarks, showing significant improvements over existing state of the art software-based implementations.

Preface

It was one year ago, during one of professor's Przemysław Pawełczak lectures, that I had the opportunity to learn about energy harvesting and battery-less devices. The promises and challenges of battery-less devices and intermittent computing inspired me to choose this topic for my final project, to hopefully contribute to the future of the Internet of Things.

What you are about to read is not only the culmination of my master studies. It is also the consummation of several years of preparation that have led to this moment. Therefore, I would like to give thanks to God and to everyone that has lent me a hand on each step of the way.

I would like to first thank my supervisors for their input and instruction throughout these nine months. Marco, thanks for your guidance and for helping me organize everything on the Twente side. Special thanks to Przemek and Vito, my TU Delft supervisors, for your ever willingness to help, and for all our discussions. Thanks for all your advice and for the good times we had during our meetings. All of them helped me to grow both personally and professionally.

I am also very grateful with the University of Twente and to CONACyT for funding me throughout these two years. Without this support, it would have been impossible for me to have this fantastic experience.

Thanks to my friends for their support, company and advice. The time we spent sharing our frustrations and having some laughs made everything easier. Thanks to all of you guys. Also, special thanks to Simon van der Jagt, Omar Link and Mario Lizana for your support during the thesis project. It was an honor to work with you guys.

I would like to thank Hosana Morales (la pingüi), for her loving support. Thank you for listening and for encouraging me to push forward on every step of the way. Finally, I would like to thank my family for always being there for me in spite of the distance. To my mother, for showing me the value of patience and endurance, and to my father, for teaching me the importance of always keeping my head up, even in the face of trouble. This work is for you.

Hiram Rayo Torres Rodriguez
Delft, The Netherlands
August 21, 2019

Contents

Abstract **i**

Preface **ii**

Contents **iii**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 State of The Art . . . 2
1.3 Research Goal . . . 3
1.4 Contributions . . . 4
1.5 Thesis Outline . . . 4

2 Background and Motivation **5**
2.1 Data Consistency . . . 5
2.2 Forward Progress . . . 7
2.3 Portability of Legacy Applications . . . 7
2.4 Checkpoint Scalability . . . 7
2.5 Why not Hardware ? . . . 8

3 Design **9**
3.1 The Translation Lookaside Buffer . . . 11
3.2 WAR Detection . . . 11
3.3 Stack Tracking . . . 11
3.3.1 Stack grow . . . 13
3.3.2 Stack shrink . . . 13
3.4 Checkpoints . . . 13

4 Implementation **15**
4.1 The OpenMSP430 . . . 15
4.2 Qwark Implementation . . . 16
4.2.1 OpenMSP430 Address Space . . . 17
4.2.2 TLB Implementation . . . 17
4.2.3 WAR Detection . . . 17
4.2.4 System Restore . . . 17

5 Evaluation **18**
5.1 Application Benchmarks 18
5.2 Results 20
5.3 Component Overhead 23
5.4 Comparison against software-based approaches 27

6 Discussion **29**

7 Conclusions and Future Work **30**
7.1 Conclusions 30
7.2 Future Work 30

Bibliography **32**

Chapter 1

Introduction

Over the past few years, the Internet of Things (IoT) [19]: a network of devices that communicate with each other to share information, has been in the spotlight of both industry and the scientific community due to the broad range of applications enabled by this technology. Recent developments in e-health [18], smart grids [2], and autonomous vehicles [41], indicate that a future where everything is connected is closer than ever. Nevertheless, this vision comes with its own set of challenges. Forecasts predict that by 2020 [55], the world will be populated by billions of these devices, all of which will require a battery to operate. However, batteries represent a highly incompatible energy source due to them being bulky, and dangerous for the environment [42], preventing miniaturization [13] and the development of reliable and cost-effective solutions. Luckily, recent developments in ultra-low power IC design [44] and energy harvesting (EH) [35] have enabled the possibility of completely removing batteries from these devices and replacing them with simpler, yet more energy efficient storage buffers. These developments have created a new paradigm: *Energy-driven computing* [20], a novel brand of embedded systems whose operation relies entirely on power obtained from ambient sources such as wind, solar and radio frequency (RF).

1.1 Problem Statement

Energy harvesting has become a solution which promises to solve the power bottleneck of the IoT, enabling batteries to be completely removed from any IoT device [31]. Nevertheless, this paradigm shift comes with new challenges, being *intermittent execution* one of the most prominent [12]. Uninterrupted execution on battery-less devices can only be achieved whenever the energy being harvested exceeds the one being consumed at any given time. If this condition is met, it is said that the device is *energy neutral*, i.e. it operates as if it had a constant energy source [20]. Nevertheless, more often than not, this is not the case, since the power density obtained from ambient sources is usually smaller compared to the one required by most devices to operate [28]. Therefore, the lack of sufficient energy to sustain continuous operation leads these devices to experience frequent power failures, resulting in data and progress loss.

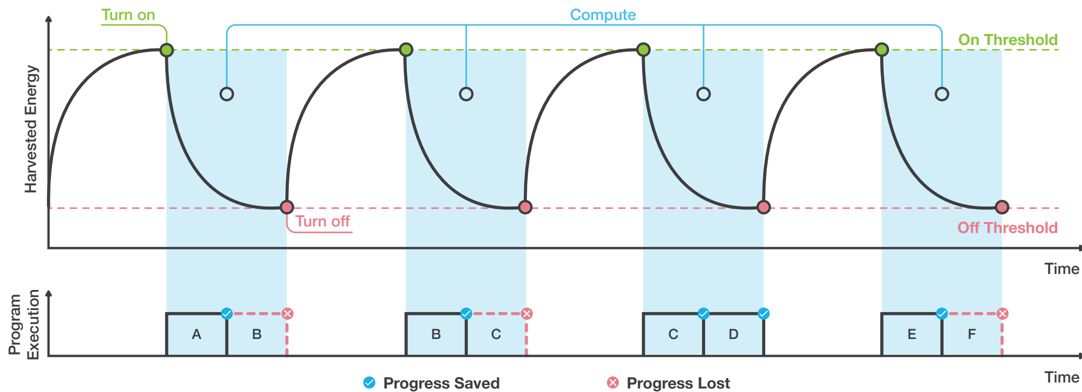

Figure. 1.1: **Harvested energy leads devices to experience frequent power failures.** Battery-less devices can only execute when the amount of energy is found within the operating thresholds. In the figure, checkpoints are introduced between program blocks (denoted with letters A-F), allowing programs to make forward progress in spite of power outages.

After a power failure, proper operation can only be resumed whenever the energy being harvested has reached a certain threshold (which varies between devices). Nevertheless, without any *fault-tolerance* mechanism, the device would resume operation from the start, hoping for enough energy to maintain continuous operation next time or at least, enough energy to complete any significant task before power runs out.

For this reason, the scientific community has relied on *checkpointing* [9] to ensure forward progress, continuously saving the device's volatile state, and restoring it after power failures. This process illustrated in Figure 1.1. Moreover, recent developments in non-volatile RAM (NV-RAM) [53, 17] have enabled *transiently-powered devices* to switch to different volatility schemes, replacing most of the onboard memory with NV-RAM [26]. Therefore, reducing the amount of data to be saved during checkpoints. Nevertheless, the utilization of non-volatile memory can lead to inconsistent states due to uncommitted memory writes, complicating the usage of NV-RAM as the primary resource for data allocation [8].

1.2 State of The Art

Current solutions on the *intermittent computing* domain can be classified by one of the following approaches: *task-based* and *checkpoint-based* systems. In task-based systems [32, 36, 1, 11], programs are manually transformed into a set of restartable tasks, whose interactions are predefined at compile-time. On the contrary, checkpoint-based systems introduce run-time support without relying on code transformation, journaling the processor's volatile state (register file, stack) to restore it upon reboots [9]. Moreover, checkpoint-based solutions can be classified as static or dynamic. Static checkpoint-based systems [30, 38, 52, 40] utilize energy-agnostic policies to trigger checkpoints. Contrarily, dynamic checkpoint-based solutions [22, 15, 16] continuously monitor energy and trigger checkpoints whenever the voltage drops below a certain threshold. Performance-wise, task-based systems impose less runtime and memory overhead, but require the complete re-implementation of programs, making the portability of legacy non-intermittent applications unfeasible. Additionally, they limit standard programmability features, such as pointers or recursion due to them being static [38]. On the contrary, checkpoint-based solutions

introduce higher runtime and memory overhead but facilitate portability and maintainability of applications [40]. To deal with the shortcomings of utter software and compiler-based solutions (task and checkpoint-based), some have proposed the implementation of entirely non-volatile processors [34], introducing non-volatility at the *flip-flop* level. Others have resorted to some degree of structural support to track memory accesses leading to erratic states [43]. Nevertheless, a simple and efficient solution which correctly handles the different aspects of intermittent execution is yet to be found.

1.3 Research Goal

In this thesis, the challenges of *intermittent computing* are addressed by designing and implementing a novel *hardware* architecture. Most of the research in this domain relies on software and compiler-based support due to the perception that hardware accelerators can lead to significant energy overheads, making them a burden, instead of a solution [11]. Nevertheless, recent work has proven that hardware-based solutions are indeed worth looking into [43]. Therefore, this work aims to demonstrate how a simple, yet efficient hardware implementation, along with minimal software support, can succeed without introducing significant hardware overhead. Moreover, for this work to be successful, the proposed solution should i) ensure forward progress across power failures, ii) ensure data consistency, iii) enable the portability of legacy non-intermittent applications and iv) introduce minimal overhead.

As a result, the main research questions of this work are the following:

1. ***How can intermittent computing be supported by lightweight mechanisms?***

2. ***What performance improvement can be obtained by a hardware/software solution?***

By answering these questions, it will be possible to design and develop a new solution which serves as a baseline for future developments and encourage further research on hardware-based solutions. Furthermore, it is the intention of this work to motivate the inclusion of such a system in current EH focused embedded processors.

1.4 Contributions

Qwark is an *intermittency management unit* peripheral designed for the MSP430 architecture [24]. Furthermore, Qwark makes use of simple address translation mechanisms, resembling traditional Memory Management Units (MMUs), to ensure correct execution and forward progress across power failures. We leverage the already available address tracking mechanisms of architectural components, such as Memory Protection Units (MPUs), to detect consistency violations on the fly and redirect them to safe memory locations. Therefore, guaranteeing writes are not committed to the actual physical location until a checkpoint is made, ensuring a consistent state is always reached. Qwark's architecture also provides an efficient solution for scalable checkpoints, allowing the utilization of a mix-volatility stack, providing a worst-case checkpoint time without compromising performance [40]. Furthermore, by developing a hardware-based solution, a transparent operation is guaranteed to the programmer, enabling portability of legacy applications without compromising common programmability features, contrarily to most compiler and software-based solutions [11, 38, 1, 37, 30].

This work presents the following contributions:

- The introduction of a lightweight address translation-based architecture to enable memory consistency via redo logging.
- A stack promotion scheme which builds on the work proposed by [40], enabling the usage of a mixed-volatility stack to provide an upper bound on checkpoint time, with little overhead.
- The first example of structural support for intermittent computing targetting the MSP430 architecture [24].
- A fair performance assessment between Qwark and current state of the art software and compiler-based solutions [48].
- An open source design and implementation on the openMSP430 [48], which can serve as a baseline for future research.

1.5 Thesis Outline

This work is structured as follows. Chapter 2 gives the reader an overview of the background and related work in this field to understand the state of the art and the motivation to opt for a hardware-based solution. Chapter 3 presents a detailed description of the proposed design. Afterward, Chapter 4 presents some of the implementation details. Chapter 5 presents an evaluation of the system, benchmark results, and a comparison between Qwark and current software and compiler-based implementations. Chapter 6 presents a discussion of the obtained results. Finally, conclusions and future research directions are presented in Chapter 7.

Chapter 2

Background and Motivation

Recently, the development of devices which operate entirely on harvested energy has become a reality. Battery-less platforms, such as WISP [3], have proven that developing computationally non-intensive applications [51, 14] running on harvested energy is feasible. Nevertheless, the frequent power failures experienced by these devices, makes the implementation of higher-complexity applications challenging, as long running computations require a mean to preserve progress and ensure data consistency. This motivates the introduction of novel mechanisms to deal with the issues introduced by intermittency. Furthermore, with the massive emergence of connected devices, and high-performance computing moving closer towards the edge, solving the challenges of intermittent execution is more important than ever [57].

2.1 Data Consistency

The usage of non-volatile memory (NV-RAM) as the principal resource for memory allocation can easily result in programs reaching inconsistent states due to writes not being successfully captured by checkpoints [8]. Previous attempts have resorted to different methods to deal with this issue. Generally, checkpoint-based systems rely on run-time logging, compile-time analysis or a mix of both to deal with inconsistencies [52, 43, 40]. Chinchilla [38] makes use of *undo logging* to rollback uncommitted changes upon reboots. Ratchet [30] segments programs into chunks of idempotent [54] sections; code fragments guaranteed to provide the same result upon every re-execution, inserting checkpoints between these to prevent writes from breaking idempotency. More importantly, as shown in Figure 2.1, *Write-After-Read* (WAR) violations are identified as the source for inconsistencies.

On the contrary, task-based systems ensure consistency by isolating shared data and treating tasks as idempotent pieces of code which execute atomically [11, 36]. Chain [1] implements disjointed input/output channels to guarantee shared data is always read consistently. Alpaca [37] uses static analysis to privatize shared data on isolated buffers, committing it to main memory upon task completion.

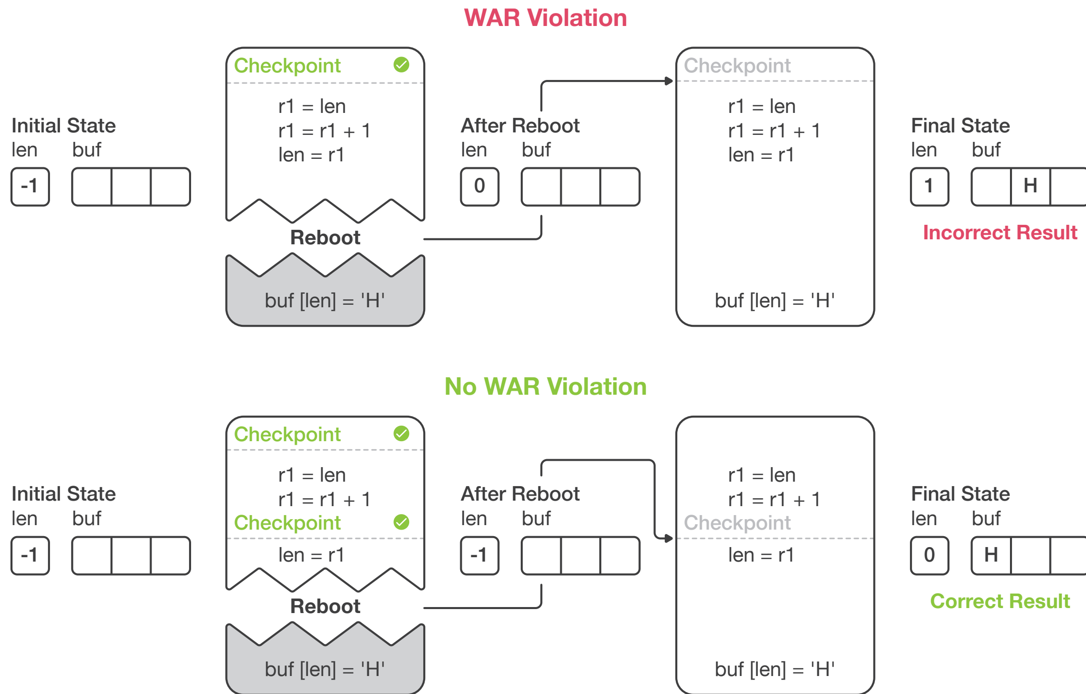

Figure. 2.1: **Write-After-Read (WAR) violations lead to inconsistent states after power failures.** Variables `len`, `r1` and `buf`, are located in non-volatile memory. **Top:** `len` is read, and then written with the value of `r1`. After rebooting, the system restarts execution from the latest checkpoint. `len` preserves its value and is incremented again, ultimately updating `buf` incorrectly. **Bottom:** Inserting a checkpoint just before `len` is written removes the WAR violation, enabling the system to execute correctly in spite of power failures.

2.2 Forward Progress

Checkpoint-based systems store the device's volatile state in non-volatile memory to restore it after power failures. Nevertheless, one key challenge is to determine when to checkpoint [12]. Too many checkpoints might introduce significant overhead, while only a few might result in the application not making any progress. To deal with this issue, dynamic checkpoint-based systems continuously monitor residual energy in the decoupling capacitor to checkpoint right before power runs out. Mementos [9] inserts energy-measurement code after function calls and loop iterations, while Hibernus [15] and QuickRecall [22] reduce overhead by triggering a checkpoint interrupt upon voltage drop-down detection, using an onboard voltage comparator [26].

Nevertheless, the utilization of power-hungry resources, such as an ADC, is unsuitable for operating conditions where energy is an already scarce resource [43]. Because of this, static checkpoint-based systems opt for energy-agnostic heuristics to ensure forward progress at the least energy cost, trading off run-time overhead due to an excess of checkpoints. Chinchilla [38] instruments programs by introducing checkpoints which are dynamically enabled/disabled during runtime. In [52], compile-time analysis and *Control Flow Graphs* (CFG) are used to define and add checkpoints at transactional regions. On the contrary, in task-based systems, forward progress is guaranteed as long as tasks can be completed in a single execution thread. Nevertheless, this complicates task division, as reasoning about energy requirements is challenging and unintuitive [40].

2.3 Portability of Legacy Applications

Developments on transiently powered computing have opened a new range of possibilities. However, porting applications coming from other domains, such as *Wireless Sensor Networks* (WSN) remains challenging with current intermittent programming approaches.

Task-based models require developers to manually identify dependencies to create tasks, specify the interactions between these, and provide special annotations. Nevertheless, task transformation is not trivial, since too many small tasks can introduce significant overhead, while only a few large ones can lead to *non-termination* [38]. Furthermore, task transformation hinders programmability, as the usage of standard programming features such as pointers, and recursion, are not supported by current models. Because of this, the research community has moved on to checkpoint-based systems, as these impose fewer limitations on programmability, facilitating the portability of existing applications.

2.4 Checkpoint Scalability

As progress is made, checkpoint-based systems continuously journal the system's volatile state. However, stack growth might result in too much data being checkpointed (E.g., the complete stack). This situation increases the likelihood of non-termination, as checkpoints might take too long to complete. Ratchet [30] and Chinchilla [38] solve this by working with an utterly non-volatile stack. Nevertheless, this limits execution speed, and increase power and run-time overhead. To address this shortcoming, VISP [40] implements a mix-volatility stack, dividing the complete stack into fixed-sized blocks, keeping only the working stack in volatile memory. However, block swapping mechanisms result in a high run-time overhead, making it significantly slower than previous approaches.

System	Pointer Support	Recursion Support	Scalability	System Overhead	Memory Overhead	Porting Effort
Chain (T) [1]	No ✗	No ✗	Poor ✗	Low ✓	Low ✓	High ✗
Alpaca (T) [37]	No ✗	No ✗	Poor ✗	Low ✓	Low ✓	High ✗
Ratchet (C) [30]	Yes ✓	No ✗	Poor ✗	High ✗	High ✗	High ✗
Chinchilla (C) [38]	Yes ✓	No ✗	Poor ✗	Low ✓	High ✗	None ✓
VISP (C) [40]	Yes ✓	Yes ✓	High ✓	High ✗	High ✗	None ✓
Clank (H) [43]	Yes ✓	Yes ✓	Poor ✗	N/A ✗	N/A ✗	None ✓
Qwark (H) (this work)	Yes ✓	Yes ✓	High ✓	Low ✓	Low ✓	None ✓

Table 2.1: **Comparison of the key characteristics of current intermittent computing solutions.** **T:** Software task-based system. **C:** Software checkpoint-based system. **H:** Hardware-based system. **N/A:** Clank's overhead cannot be directly compared, as its implementation targets the ARM architecture. The table shows Qwark ability to comply with all of the key characteristics of intermittent systems.

2.5 Why not Hardware ?

Compared to the plethora of research [1, 37, 38, 40, 15, 22, 9] on software and compiler-based solutions, little work has been done on the hardware side. In [32], an architecture for a *Non-Volatile Processor* (NVP) is proposed, introducing non-volatility at flip-flop level to allow for seamless forward progress and consistency. Nevertheless, this kind of architecture significantly increases the amount of power consumption and impacts processing speed. To reach a middle-ground, Clank [43] includes custom hardware to detect WAR violations during run-time, and store violating data and addresses on a cache-like structure. Future matches to this structure are treated as cache-hits, and once enough violations have occurred, data is flushed to main memory. Nevertheless, Clank fails to provide a solution for scalable checkpoints. Furthermore, contrarily to previous approaches [1, 38, 30, 37], which target the MSP430 architecture [24], Clank is designed for ARM [7], which complicates a fair performance assessment against current compiler and software-based solutions. This is mainly due to the architectural and compiler differences which impact the number and source of violations.

In Table 2.1, the key characteristics of state of the art solutions are compared with Qwark. Task-based models limit programmability and portability, while current checkpoint-based solutions increase run-time and memory overhead [40]. Nevertheless, as shown by [43], structural support can relieve pressure on the software side, reducing the overhead experienced by current software and compiler checkpoint-based systems. Additionally, lightweight architectural components could provide a solution for scalable checkpoints without introducing significant overhead [40]. Furthermore, recent work on sub-threshold and near-threshold design [6] has shown that extremely low-power, and stable solutions can be achieved, making the idea of highly efficient hardware accelerators an attractive choice to provide the best results in terms of performance and power.

Chapter 3

Design

This work proposes a static checkpoint-based system intended to maximize performance at the least energy cost. Furthermore, it follows recent research trends to enable the portability of legacy non-intermittent applications without compromising programmability features [38, 40]. Nevertheless, this work diverges from previous approaches [38, 37, 30, 52, 22, 9] by relying on lightweight structural support to relieve pressure on the software side. Furthermore, this work targets the MSP430 architecture [24] due to it being commonly used in EH-based applications [51, 14] and intermittent computing research [29, 45], to allow a fair comparison with existing intermittent programming solutions and motivate future research in this domain.

Qwark is a memory-mapped peripheral designed for the MSP430 architecture. By continuously tracking memory accesses, Qwark detects WAR violations, and redirects them to isolated locations, creating a redo log [37] in a dedicated segment in non-volatile memory. By doing this, hazardous accesses which could result in errant states, are prevented from directly modifying data. Furthermore, Qwark provides a solution for scalable checkpoints, enabling the utilization of a mix volatility stack to provide an upper bound on checkpoint time with little overhead. By continuously analyzing the stack behaviour, Qwark promotes variable-size segments to non-volatile memory, effectively dividing the stack into two segments; volatile and non-volatile, linked via address translation. During checkpoints, volatile state and supporting data is journaled on non-volatile memory, and the redo log is copied back to memory, allowing for consistent states to be reached. Furthermore, as shown in Figure 3.1, Qwark's functionality is achieved by three modules: The Translation Lookaside Buffer (TLB), WAR detection, and Stack Tracking. The following sections expand on these and their interactions.

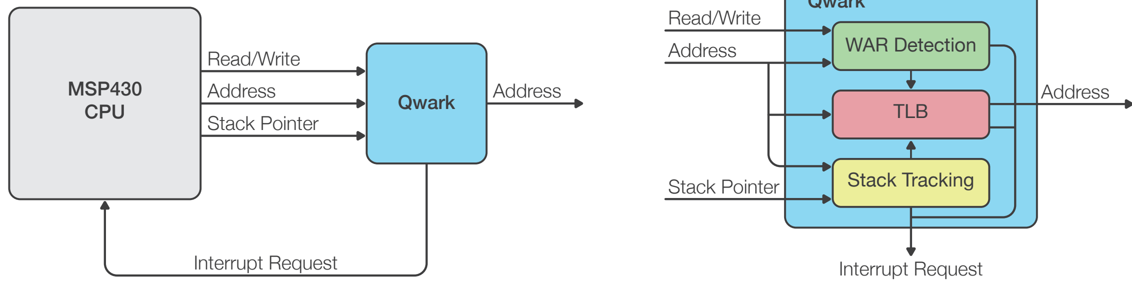

Figure. 3.1: **Qwark system overview.** A hardware/software solution which gives the user the illusion of programming a continuously-powered system by transparently dealing with the challenges of intermittent execution. **Left :** The Input/Output signals illustrate the interaction of Qwark with the MSP430 CPU from a system perspective. **Right:** Inner composition of Qwark, depicting the modules whose interaction achieve the system functionality.

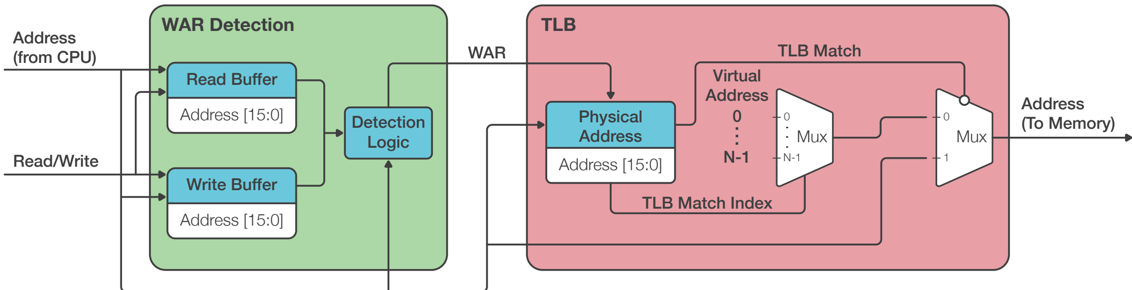

Figure. 3.2: **The TLB and WAR modules enable hazardous accesses to be isolated, creating a log in a dedicated segment in non-volatile memory.** The WAR detection block stores and compares each access to identify write-after-read violations. The WAR signal triggers a request for the TLB to buffer the hazardous memory access. The TLB Match, and TLB Match Index signals are used to translate the address to a fixed location in non-volatile memory. The virtual address annotation symbolize the number of reserved memory locations which comprise the redo log. Checkpointing logic is not depicted in this figure to facilitate understanding.

3.1 The Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) oversees the translation of memory accesses that have or can lead to inconsistent states. When a violation is detected, the hazardous access is stored in the TLB, and a separate address located in the redo log; a reserved segment in non-volatile memory, is assigned to it, forming a `(physical, virtual)` address value pair. Afterwards, the TLB match and its entry index are used to redirect the access to the corresponding location in the log, leaving data on the physical address untouched. This mechanism is illustrated in Figure 3.2. Moreover, subsequent accesses to the same location are translated by merely verifying the content of the TLB, ensuring data is fetched from the right memory location. Contrarily to [43], Qwark presents a more simplistic approach, reducing the amount of hardware required for its operation by eliminating the need for storing data in addition to addresses.

3.2 WAR Detection

Borrowing from previous work [30, 43], the WAR detection block provides a mean to identify harmful memory accesses, which could result in inconsistencies. Inside the block, read/write addresses are stored in separate buffers and compared with subsequent writes to identify WAR violations [43]. This mechanism is used to notify the TLB whether an access should be isolated. Additionally, it is essential to emphasize the importance of incorporating such a block since not all write accesses can result in inconsistent states being reached. E.g., *Read-after-Write* (RAW) or *Write-Read-Write* (WRW) accesses do not break idempotency [30, 43]. Therefore, such a block becomes necessary to maximize performance by precisely identifying which accesses should be isolated to stretch computation as much as possible [30, 43].

3.3 Stack Tracking

Building upon the idea proposed by [40], this work introduces the notion of *Stack Promotion*. Qwark continuously monitors the *Stack Pointer* (SP) to identify if the amount of volatile stack usage has exceeded a built-in threshold. If so, the stack tracking block triggers a checkpoint interrupt to define the *segment boundary*; a limit which serves as a frontier to divide the stack into two volatile and non-volatile sections. Moreover, the segment boundary is always defined to allow the working stack to remain in volatile memory. Afterwards, all the stack up to the segment boundary is promoted (copied) to non-volatile memory. This process is shown in Figure 3.3 a). Accesses made to addresses lower (SP diminishes while the stack grows) than the segment boundary are redirected to the corresponding address in non-volatile memory, which is calculated by subtracting an offset to the address. On the contrary, accesses made to addresses higher than the segment boundary are allowed to work with volatile memory. Moreover, after promotion, consistency mechanisms are utilized to ensure correctness on the non-volatile stack segment.

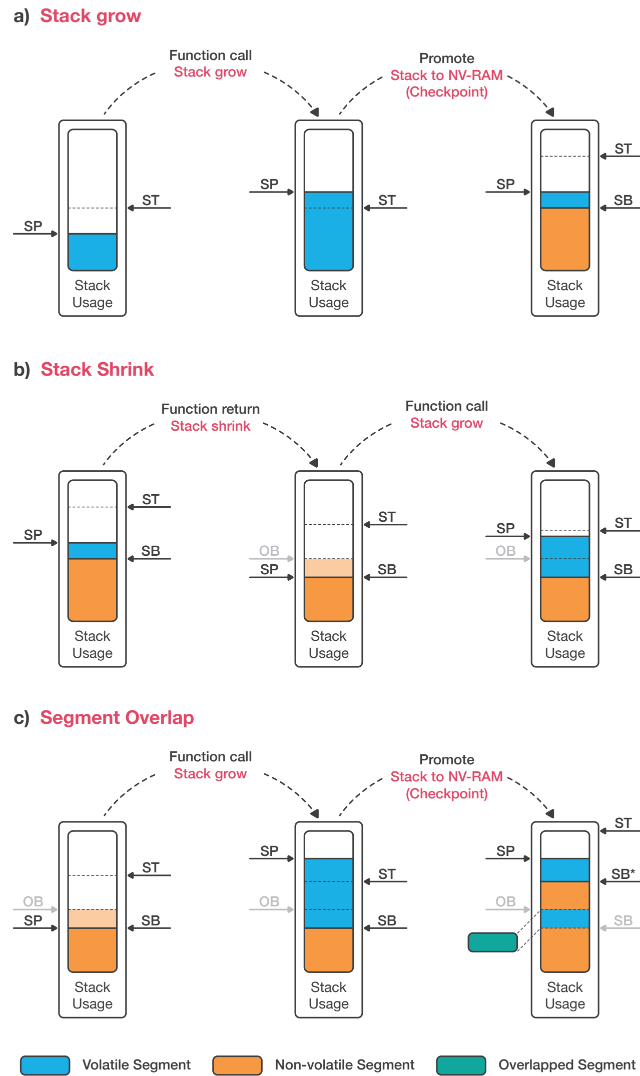

Figure. 3.3: **Stack promotion allows Qwark to segment the stack into variable-sized volatile and non-volatile segments. a)** when the stack pointer (SP) grows beyond the segment threshold (ST), a segment boundary (SB) is defined, and the stack is promoted to non-volatile memory using the SB as upper bound. **b)** stack shrink pass the active segment boundary triggers an update to the SB, redefining the SB to the current SP to allow further stack grow to use volatile memory. OB denotes the old segment boundary. **c)** subsequent checkpoints can lead to an overlap between the OB, the current SB and the new segment boundary (SB*). Blindly promoting the stack in this condition can lead to inconsistencies in case the overlapped segment is partially updated.

3.3.1 Stack grow

During program execution, Qwark analyzes the stack behaviour to identify the amount of stack grow/shrink between checkpoints. By doing this, Qwark performs differential checkpoints using the maximum SP value and the segment boundary, as an upper and lower limit for promotion [49], speeding up the promotion process. Nevertheless, it is vital for the promotion algorithm to take the previously checkpointed segment boundary into account, as blindly promoting the stack can lead to inconsistent states being reached in case of a power failure happening mid-promotion. This is shown in Figure 3.3 c), where subsequent checkpoints result in an overlap between promoted segments. To overcome this problem, Qwark identifies and double buffers the overlapped portion of the stack (if any) to prevent partial updates. The non-overlapped portion of the stack is then promoted directly to non-volatile memory, leveraging the fact that this process is naturally idempotent [54] since, in case of a failure, this *half-promoted* stack is discarded automatically by merely using the previous segment boundary.

3.3.2 Stack shrink

Contrarily to [40], when the stack shrinks beyond the segment boundary, Qwark does not enforce a copy-back of the promoted non-volatile stack to volatile memory, since this would result in considerable run-time overhead. Instead, Qwark allows the system to continue working uniquely with non-volatile memory. Nevertheless, to prevent the application from getting stuck utilizing only non-volatile memory, a boundary update is enforced whenever the SP shrinks beyond the segment boundary, adjusting it to the SP value to allow for new stack frames beyond the new limit to be allocated in volatile memory. This is shown in Figure 3.3 b).

3.4 Checkpoints

To ensure forward progress, Qwark introduces three sources to trigger checkpoint interrupts:

1. **Buffer-Full Interrupt:** Triggered when the TLB or the WAR block reach a full condition. Since both buffers are finite resources, it is necessary to trigger a checkpoint to prevent any violation from being missed.

2. **Stack Grow Interrupt:** Triggered when stack grow surpasses the segment threshold.

3. **Timer-driven Interrupt:** Triggered by timer expiration. It is possible for code sections to not result in neither buffer-full conditions nor stack grow. To overcome this problem, a timer is used to allow for periodical checkpoints to be taken as an additional measure to prevent non-termination.

An important thing to consider is the coherence between the checkpoint triggers. To ensure correctness, all three sources must perform the same checkpoint operation. E.g., if a checkpoint is triggered due to stack promotion, the redo log should also be copied back to main memory. Similarly, if the interrupt is triggered by a buffer-full condition, the stack should be promoted as well. Otherwise, the register file would be saved, but local variables located in the volatile stack segment would be lost. Moreover, when a checkpoint is triggered by a buffer-full condition or a timer-driven interrupt, the SP is used as the segment boundary.

Furthermore the checkpoint ISR is divided in two phases. First, the TLB `physical` addresses are copied to a dedicated segment in non-volatile memory, along with an index which indicates the number of violations held in the TLB. Then, the segment boundary is committed and the stack is promoted. Afterwards, the first phase is completed by atomically updating a flag located in non-volatile memory, signaling a *half-way* checkpoint.

The second phase copies back the data stored in the `virtual` locations to their original `physical` addresses. Moreover, the overlapped portion (if any) of the promoted stack is copied back to complete the promotion process. Finally, the flag is updated to indicate the checkpoint completion. The two-phase division allows checkpoints to be restarted in case of a power failure, as long as the first phase has been finalized since all the data required to complete the process is already stored in non-volatile memory by that point [37]. Upon reboot the restore routine uses the flag to perform a second phase retry, in case a half way checkpoint exists in memory.

Moreover, it is possible for the system to run out of power during checkpoints. This could result in partially committed checkpoints which would leave the system in an errant state. For this reason, all checkpoint data (register file, TLB addresses, segment boundary, etc) is double buffered to ensure there is always a consistent checkpoint stored in memory, using the atomic flag update to signal both the buffer switch and the two-phase division, similar to previous approaches [37, 52].

Chapter 4

Implementation

For fairness, we implement Qwark's design on the MSP430 architecture [24], using the open-MSP430 [48] as the baseline for development. Furthermore, due to the absence of onboard NV-RAM, we emulate non-volatility, simulating intermittency as induced soft-resets, and manually wiping up the volatile state after reboots.

4.1 The OpenMSP430

The OpenMSP430 is a synthesizable 16-bit core written in Verilog, completely compatible with the MSP430 microcontroller family [26], and fully capable of executing code generated by any MSP430 toolchain with near-cycle accuracy.

To probe the validity of the comparison between the OpenMSP430 and the real MSP430, three benchmarks commonly used in this research domain (Activity Recognition (AR), Bitcount (BC) and Cuckoo filter) [43, 38, 40] were taken from the MiBench repository [10] and executed on the OpenMSP430, MSP430G2553 and MSP430FR5969 platforms. All benchmarks were performed at 1 MHz on continuous power, using the *msp430-gcc-7.3.1.24* compiler from *Mitto Systems* [27] under three different optimization levels (O0-O2), using the same compiler flags, and utilizing only the MSP430 ISA to prevent the usage of the extended MSP430X ISA (only for the MSP430FR5969) [24]. Additionally, the generated object code was analyzed and compared using *msp430-elf-objdump* to ensure the three platforms executed the same code. The only differences came from some of the Embedded Application Binary Interface (EABI) [25] routines introduced due to the difference among linker scripts. Nevertheless, these differences were minimal, compared to the rest of the program. Moreover, execution time was measured by toggling a digital output at the beginning and the end of each benchmark using a saleae logic analyzer [50]. Additionally, the cycle counter tool on Code Composer Studio was used as an extra source to compare the accuracy of the measurement for the MSP430G2553 and the MSP430FR5969, showing similar results.

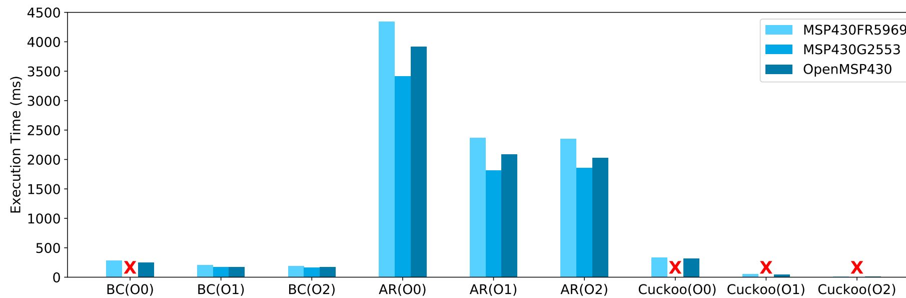

Figure. 4.1: **Execution time comparison between MSP430 platforms.** Red cross (✗) denotes the application did not fit in memory. For fairness, benchmarks were executed on continuous power at 1 MHz. Results show a near-constant deviation between each pair of benchmarks, under different optimization settings **(O0-O2)** allowing Qwark to be compared with intermittent frameworks implemented on the actual MSP430 processor.

Following Figure 4.1, it can be observed that while not being equal, the results obtained from the OpenMSP430 are within the range of both the MSP430FR5969 and the MSP430G2553, showing a near-constant deviation between benchmarks. Moreover, a higher similarity can be identified between the MSP430G2553 and the OpenMSP430. This similarity makes sense since the OpenMSP430 was designed to resemble the original MSP430 architecture [48], contrarily to the MSP430FR5969, that although forced to use only the MSP430 ISA, utilizes a different core, and presents more differences on linked routines, compared to both the OpenMSP430 and the MSP430G2553. More importantly, based on these results, it can be concluded that the OpenMSP430 performance can be compared to the real MSP430 processor. Therefore, making the results obtained by Qwark valid and expected in an actual silicon implementation, allowing the comparison with compiler and software-based systems.

4.2 Qwark Implementation

Qwark's implementation is divided into two functional blocks, the peripheral handling logic for accessing control registers and interrupt handling, and the *Qwark core*, which implements most of the functionality described in Chapter 3. Furthermore, it is essential to mention that most of the peripheral's logic is developed on top of the peripheral infrastructure already provided by the OpenMSP430 [48]. Similar to other peripherals (e.g., UART, Timer), Qwark can be enabled or disabled upon request by merely writing the enable bit on its control register. When enabled, Qwark executes all of its functionality automatically, without any user intervention. Otherwise, when disabled, all features are shut down, and Qwark is bypassed, leaving every address access untouched.

4.2.1 OpenMSP430 Address Space

Qwark should effectively track every access to non-volatile memory to ensure consistency. Therefore, Qwark's operation is entirely dependant on the amount of non-volatile memory, and its distribution over the address space of the system in which Qwark is integrated. Because of this, Qwark's implementation provides parameters to configure the valid address range that it should protect, allowing it to adapt to different volatility distributions.

4.2.2 TLB Implementation

The TLB functionality is divided between Qwark's functional blocks (peripheral and core). All the logic for the TLB handling is located inside Qwark's core. Nevertheless, when a violation is detected by the WAR block, the `physical` address of the (`physical,virtual`) pairs are directly written into memory mapped registers inside the peripheral block to allow these addresses to be directly read by the checkpoint ISR. Furthermore, the TLB is implemented as a register-based data structure with a width and depth of 16×8, respectively. It is also important to emphasize that the depth of the TLB can be easily increased or decreased to analyze the hardware logic and performance trade-off. Nevertheless, due to the timing constraints imposed for this work, this depth was chosen for analysis.

4.2.3 WAR Detection

As mentioned in Chapter 3, the WAR block implementation follows the base idea of address tracking presented by [43]. Nevertheless, for this work, the WAR detection was entirely re-implemented based on the characteristics of the MSP430 architecture. Moreover, the read/write buffers are implemented as register-based data structures to allow for look-ups to occur in parallel, following a composition symmetric to the TLB, with a width and depth of 16×8, respectively.

4.2.4 System Restore

The checkpoint ISR and the system restore routines are provided to the user in device driver files to facilitate their usage. Nevertheless, instead of modifying the built-in compiler startup routines, the restore function is merely linked to the startup sequence, to verify the existence of a valid checkpoint in memory. If no checkpoint is found (first boot), the compiler's default startup routine is executed. Otherwise, the restore routine takes over to restore the system's state using the latest valid checkpoint. First, the atomic flag is read to verify whether a half-way (first-phase completed only) checkpoint exists in memory. If a half-way checkpoint is found, the second phase is retried, the checkpoint is completed, and the atomic flag is modified to signal the checkpoint validity. Finally, the register file is restored.

Chapter 5

Evaluation

In this chapter, we present and evaluate three versions of Qwark to analyze performance and trade-offs. First, we evaluate Qwark utilizing an *unbounded* volatile stack, checkpointing the complete stack (no differential checkpoints) at each checkpoint, without stack promotion implemented. Second, we configure Qwark to work with a fully non-volatile stack without Stack Promotion, to analyze the run-time overhead and motivate the need for implementing a mix-volatility stack. Third, we evaluate Qwark's complete implementation; stack promotion implemented with a threshold boundary of 64 bytes to analyze the performance improvement obtained by including Stack Promotion. Finally, we present Qwark's component overhead, power consumption estimation, and compare its performance with current compiler and software-based solutions.

5.1 Application Benchmarks

To evaluate performance, five benchmarks commonly used in this research domain were taken from MiBench [10]: Activity Recognition (AR), Bitcount (BC), Cuckoo filter, Advanced Encryption Standard (AES), and Cyclic Redundancy Check (CRC). AR implements a machine learning based activity recognition algorithm utilizing a data set with accelerometer data. BC implements bit counting on a string with seven different routines. Cuckoo implements cuckoo filtering on a set of pseudo-random numbers. AES performs Electronic Codebook (ECB) and Cipher Block Chaining (CBC) encryption and decryption on a 64-byte plain-text. CRC implements a fast and slow algorithm using the CRC_CCITT polynomial. For fairness, all benchmarks were done on continuous power, at 1 MHz under different optimization settings (O0-O2) using the *msp430-gcc-7.3.1.24* compiler from Mitto Systems [27]. Moreover, execution time was measured utilizing a saleae logic analyzer [50], and compared against the original *non-intermittent* C implementation running on the OpenMSP430 [48].

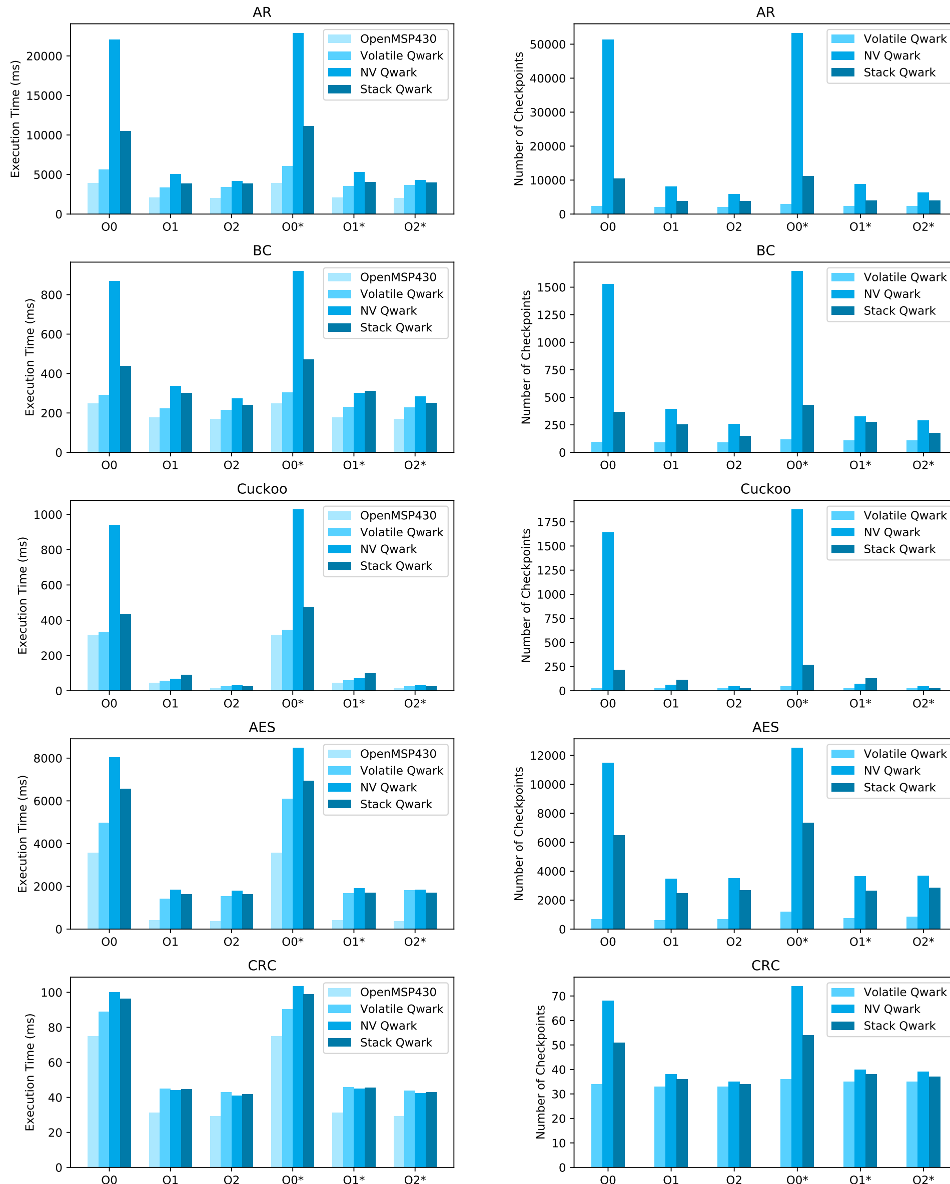

Figure. 5.1: **Qwark application benchmarks. OpenMSP430:** Original non-intermittent results. **Volatile Qwark:** Qwark implementation without stack promotion, tracking only *.data* and *.bss* sections, and saving the complete stack at each checkpoint. **NV Qwark:** Fully non-volatile implementation without stack promotion. **Stack Qwark:** Qwark with stack promotion implemented, with a segment threshold of 64 bytes. **O0*-O2*** Same implementation but including the timer-driven interrupt to checkpoint every 10ms. For fairness, all benchmarks were done on continuous power at 1 MHz. Results obtained by Stack Qwark show an overall performance improvement over NV-Qwark, proving the benefits of introducing stack promotion, and the usage of a mix-volatility stack.

5.2 Results

Benchmark results are shown in Figure 5.1. On the one hand, plots show that Qwark's utterly non-volatile solution presents an overall run-time overhead of 2x compared to the original *non-intermittent* results. This behaviour is to be expected due to the significant increase in checkpoints, based on the necessity of having to track the entire address space to ensure correctness. On the other hand, results obtained by introducing stack promotion show improvements across all benchmarks, presenting an average run-time reduction of 107% for AR, BC and Cuckoo, and 22% and 4% for AES and CRC respectively, under the O0 optimization. To explain the difference in run-time improvement, we plot the stack pointer and segment boundary, and analyze the results. Figure 5.2 illustrates how the segment boundary tracks the stack pointer, segmenting the stack. Furthermore, it can be observed how run-time decrease relates to the mix-volatility utilization, showing better results where the stack utilization presents a better volatility balance (e.g., AR, BC and Cuckoo), contrarily to AES and CRC where stack utilization leans towards full non-volatility. Thus, impacting run-time overhead as the amount of checkpoints increase. Moreover, such imbalance is introduced by a dominance of buffer-full driven checkpoints, which lean stack utilization towards non-volatility by updating the segment boundary to the current stack pointer.

Furthermore, by analyzing the results obtained by Cuckoo and CRC under O1, we observe a worse performance compared to the fully non-volatile implementation. This situation might seem unintuitive if NV Qwark's run-time is assumed to represent an upper bound on execution time. Nevertheless, by running and analyzing system simulations, results showed that different checkpoint placement can lead to more checkpoints being taken afterward, based on how read/write dependencies are segmented between checkpoints. This implies that triggering checkpoints at *buffer-full* conditions might not always lead to the best results.

Additionally, we evaluate Qwark's complete implementation with different segment boundaries to analyze the impact of different boundary values on run-time overhead. The results shown in Figure 5.3 indicate that a smaller boundary value can significantly increase run-time overhead due to the increase in checkpoints. Moreover, this is directly related to the amount of stack utilization, showing a higher impact on benchmarks which utilize more stack. Nevertheless, the results obtained by AES, and CRC indicate that this is not true for benchmarks where buffer-full driven checkpoints dominate, resulting in approximately the same performance for different segment values.

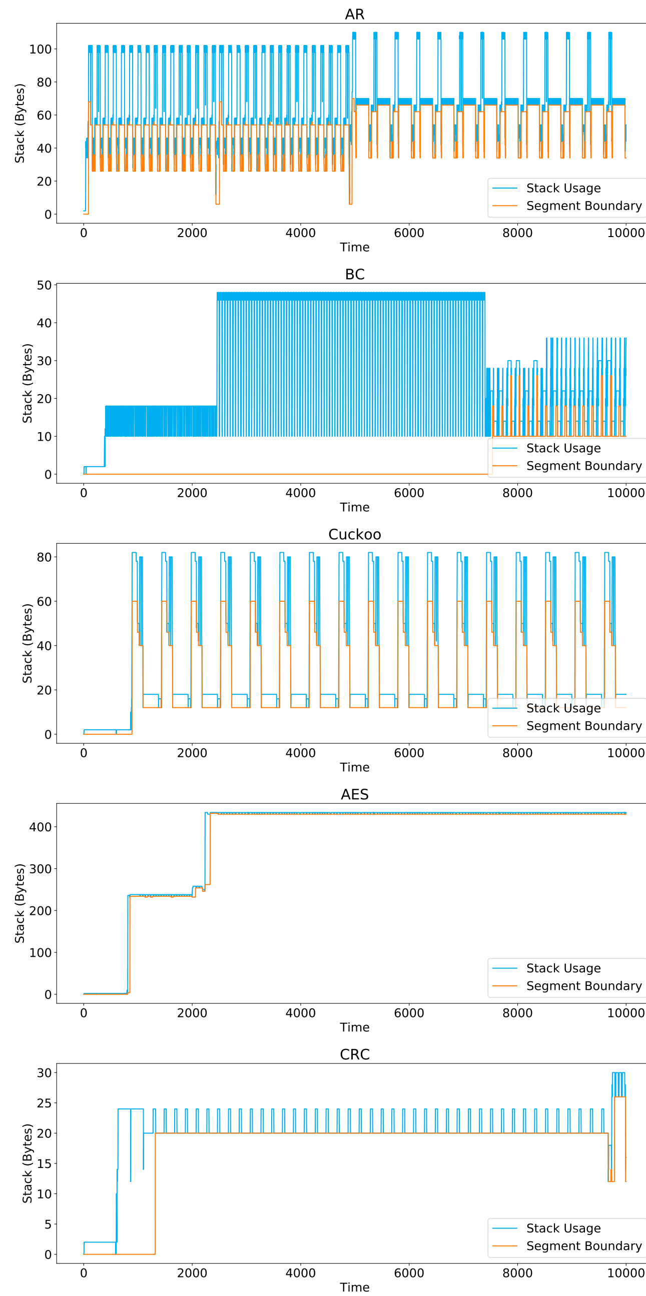

Figure. 5.2: **Benchmarks mix-volatility stack utilization.** Results are obtained by simulating and measuring each benchmark running on Qwark's complete implementation using Icarus Verilog [56], under O0 optimization. The **Stack Usage** trace depicts the stack pointer behaviour, representing the amount of stack memory being utilized by the program at each time. The **Segment Boundary** trace shows the division of mix-volatility over time. Accesses made to addresses below the boundary are redirected to the promoted non-volatile stack, while accesses made to addresses above the boundary are made to the volatile stack. This mechanism reduces the amount of non-volatile memory utilization, and consequently runtime and power overhead.

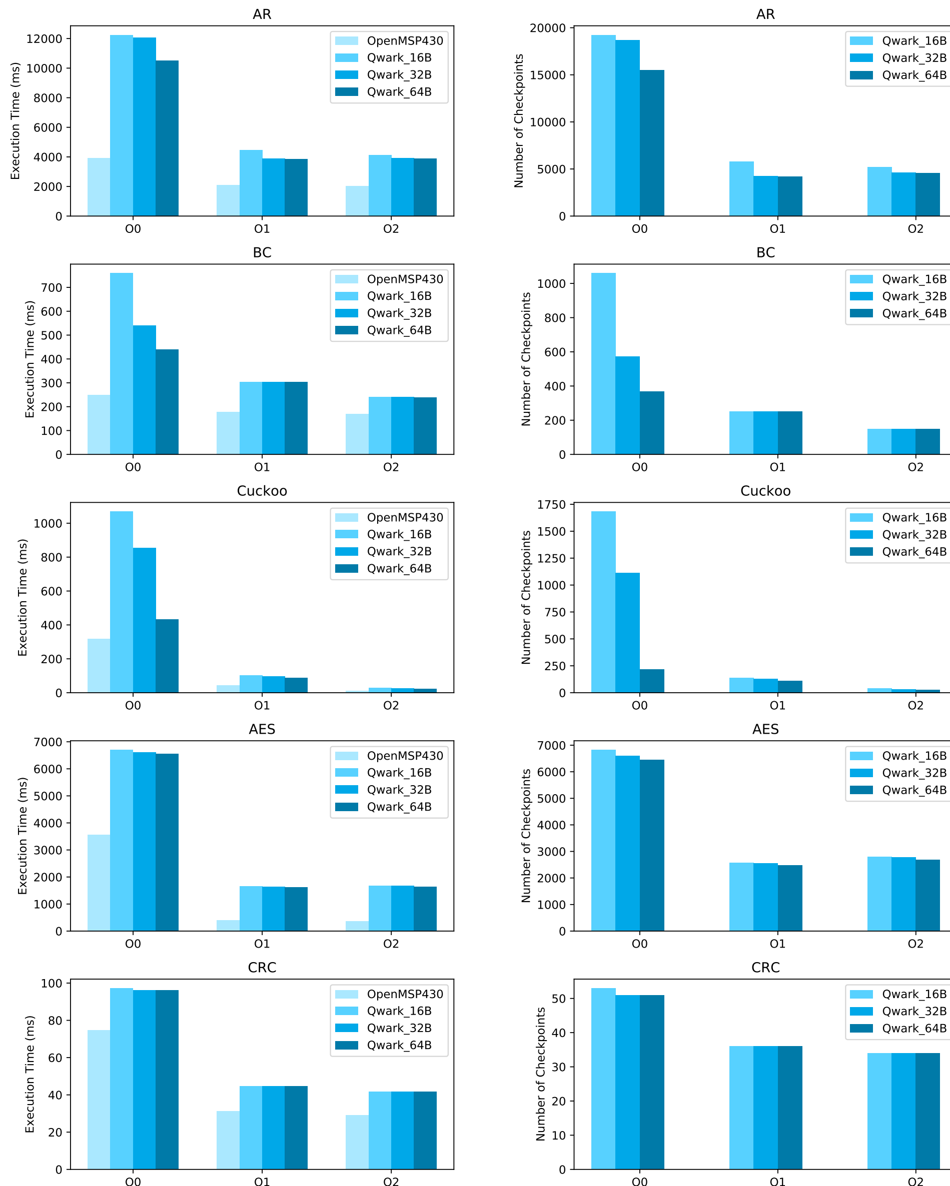

Figure. 5.3: **Qwark application benchmarks with different segment boundaries. OpenMSP430:** Original non-intermittent results. For fairness, all benchmarks were done on continuous power at 1 MHz for different optimization settings (O0-O2) without timer-driven checkpoints and segment boundaries of 16B, 32B, and 64B respectively. Results show that a smaller boundary usually leads to higher run-time overhead, except for AES and CRC where plots indicate similar performance due to checkpoints being driven mostly by buffer-full conditions.

System	ALUTs	Logic Registers	Static Current (A) ICCINT (1.20V)	Dynamic Current (A) ICCINT (1.20V)
Volatile Qwark	837	590	0.000	0.018
NV Qwark	837	590	0.000	0.018
Stack Qwark	1172	729	0.000	0.023

Table 5.1: **Hardware overhead of the three implementations** Logic utilization is obtained by running analysis and synthesis on Quartus Prime Version 18.1.0 under balanced optimization. Qwark incurs in a relatively low area increase, with Qwark's core Adaptive Lookup Tables (ALUTs) being the major source of overhead due to the amount of logic required to correctly detect WAR violations.

Design	ALUTs	Logic Registers	Memory Bits
Qwark	1172	729	0
AES128	2378	924	0
QSPI	1061	889	0
802.15.4 PHY	1461	386	20
RTC	659	458	0
I2C	302	153	0
CAN	1920	1175	320

Table 5.2: **Peripheral resource utilization comparison.** All logic utilization is obtained by running analysis and synthesis on Quartus Prime Version 18.1.0 under balanced optimization. Peripheral designs were borrowed from OpenCores [47]. Qwark presents similar resource utilization compared to the chosen designs.

	Volatile Qwark		NV-Qwark		Stack Qwark	
	.text	*.data*	*.text*	*.data*	*.text*	*.data*
Checkpoint ISR	472	0	424	0	716	0
Restore Routine	380	0	358	0	456	0
Redo Log	0	118	0	118	0	120
Stack	0	*	0	0	0	**

Table 5.3: **Memory overhead (Bytes) of the three implementations. *:** Application dependant (Unbounded). ****:** Twice the segment-size in the worst-case. Qwark presents a fixed code utilization of approximately 1KB across all of its implementations.

5.3 Component Overhead

To measure component overhead, we implement Qwark on an Altera Cyclone IV-E FPGA (60nm), using the DE0 nano [5] development board. Furthermore, we configure Quartus optimization mode to balanced and verify that each implementation meets timing, setting the clock at 1 MHz. For power consumption, we utilize the built-in Quartus Early Power Estimator (EPE) tool. Table 5.1 shows the results obtained by the EPE tool. Power consumption is reported as a function of the static and dynamic current, with dynamic current being the only source of consumption incurred by logic switching. Nevertheless, it is essential to mention that Qwark's switching comes mainly from WAR tracking, which depends entirely on the application and the volatility of the address space. This situation complicates a precise calculation of power consumption. Furthermore, current consumption would significantly vary on an ASIC implementation, the final target of this research [23]. Therefore, these numbers are merely informative. Moreover, Table 5.1 presents the hardware logic required by each of the three implementations, showing the same amount of Adaptive Look-up tables (ALUTs) and logic

registers being utilized for Qwark's volatile and non-volatile implementation, requiring only configuration changes to adapt it to each condition, introducing little extra logic to support the stack tracking functionality described in chapter 3.

Furthermore, to provide a reference for Qwark's resource utilization, six publicly available designs were taken from OpenCores [47] and compared against Qwark results. In selecting the benchmarks, two factors were considered. First, the designs had to be FPGA proven to guarantee their proper functionality. Second, they had to be peripherals commonly found in existing microcontroller units (MCUs), to ensure a fair comparison, as the intention is to motivate the inclusion of Qwark in EH-focused MCUs. The results in Table 5.2 show that Qwark resource utilization (ALUTs and Logic Registers) is relatively similar to the presented benchmarks, showing a slightly bigger resource utilization than simpler peripherals such as I2C or RTC, but smaller utilization than higher complexity peripherals like AES128.

Table 5.3 presents the memory overhead imposed by the integration of Qwark's support routines, showing a maximum code utilization (.text) of approximately 1KB for Stack Qwark. For *.data*, we consider the amount of memory required to double buffer the register file, along with supporting data. Moreover, we calculate the overhead imposed by stack double buffering to be bounded by twice the maximum segment size, assuming a complete overlap in the worst-case.

Figure 5.4 presents a calculation of the worst-case checkpoint and restore time. These numbers are obtained by assessing the amount of cycles required for the checkpoint and restore routines to reach completion, assuming the longest paths in the code are taken, and a full TLB. Also, we consider restore routines to perform a second-phase retry. Moreover, for stack Qwark, we assume a 64-byte segment size and a complete segment overlap. Furthermore, these calculations do not depend on the optimization level, since these routines are coded entirely in assembly, and prevented from being optimized. Notwithstanding, following the actual checkpoint time distribution of the benchmarks shown in Figure 5.5, we consider these to be a pessimistic calculation, and unlikely to occur in real conditions.

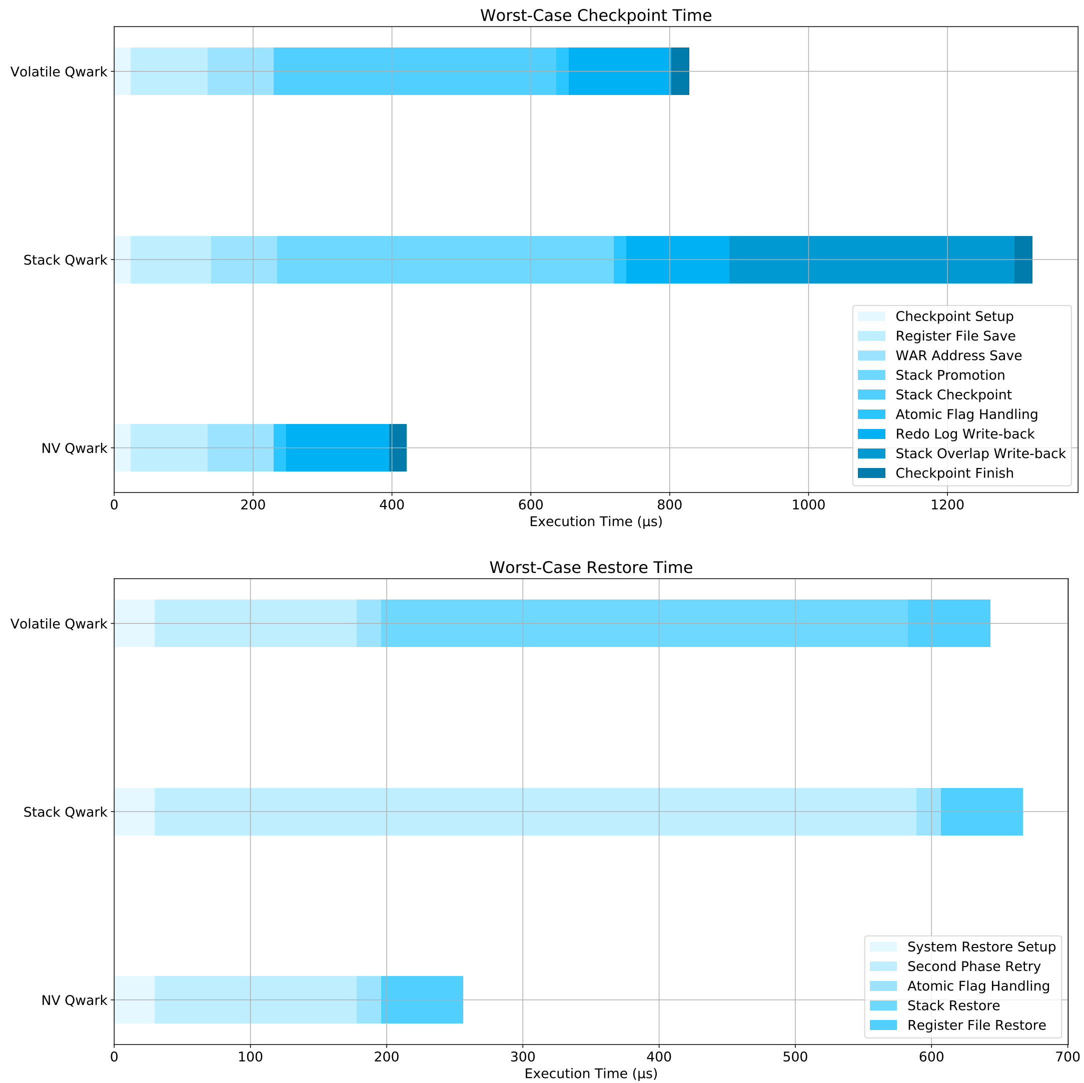

Figure. 5.4: **Worst-case checkpoint and restore times.** Worst-case execution is calculated by assuming longest paths being taken by each routine. For volatile Qwark, a fixed 64 byte stack is considered. For Stack Qwark, a segment boundary of 64 bytes is considered. Execution time of both routines is not optimization dependant.

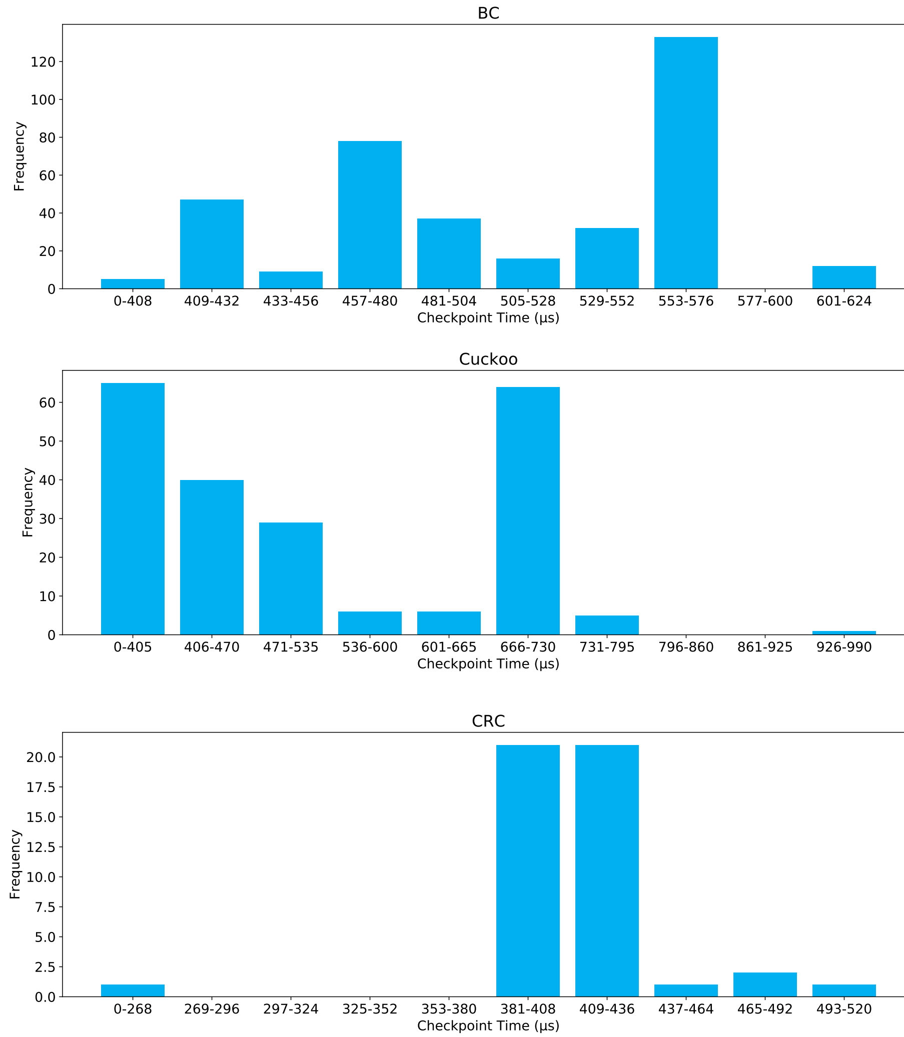

Figure. 5.5: **Qwark checkpoint time distribution.** Data is obtained by simulating each benchmark running on Qwark's complete implementation using Icarus Verilog [58], under O2 optimization. Results show the distribution of checkpointing time being lower than the worst-case checkpointing time presented in Figure 5.4.

	Original		**Ink [36]**		**VISP [40]**		**Qwark**	
	.text	*.data*	*.text*	*.data*	*.text*	*.data*	*.text*	*.data*
AR	3776	124	3442	4459	14135	13384	4948	372
BC	2206	256	2922	4433	14721	13510	3378	504
Cuckoo	3274	200	2648	4693	16467	14282	4446	448

Table 5.4: **Memory overhead runtime comparison. Qwark**: Complete implementation without considering memory corresponding to the promoted stack. **Ink:** memory overhead corresponds to the task-based implementation of the original benchmark. Relying on structural support allows for Qwark's memory footprint to be significantly smaller compared to the presented software-based runtimes.

5.4 Comparison against software-based approaches

To analyze Qwark's performance against software-based solutions, we compare three task-based systems: Ink [36], Alpaca [37], and Mayfly [32]. Moreover, we compare Qwark's implementation to VISP [40], the first checkpoint-based system to introduce the concept of a mix-volatility stack. We borrow three commonly used benchmarks: AR, BC, and cuckoo, and analyze the execution time of each running on continuous power at 1 MHz. Additionally, we consider the memory overhead of each scheme. More importantly, it is crucial to emphasize that data corresponding to software-based frameworks was obtained by measuring execution time on a MSP430FR5969 platform. Therefore, comparisons should not be made in a one to one basis, due to the openMSP430[48] being similar to the MSP430 [26], not equal. Instead, it is our intention to indicate potential improvement, instead of precise numbers. Figure 5.6 shows the data for comparison. Considering the aforementioned limitations, results indicate Qwark can reduce execution time compared to any of the chosen run-times. Moreover, due to mainly relying on hardware to achieve its functionality, Qwark introduces little memory overhead, contrarily to software-based solutions [40, 36, 38], which significantly increase memory footprint. This overhead is shown in Table 5.4.

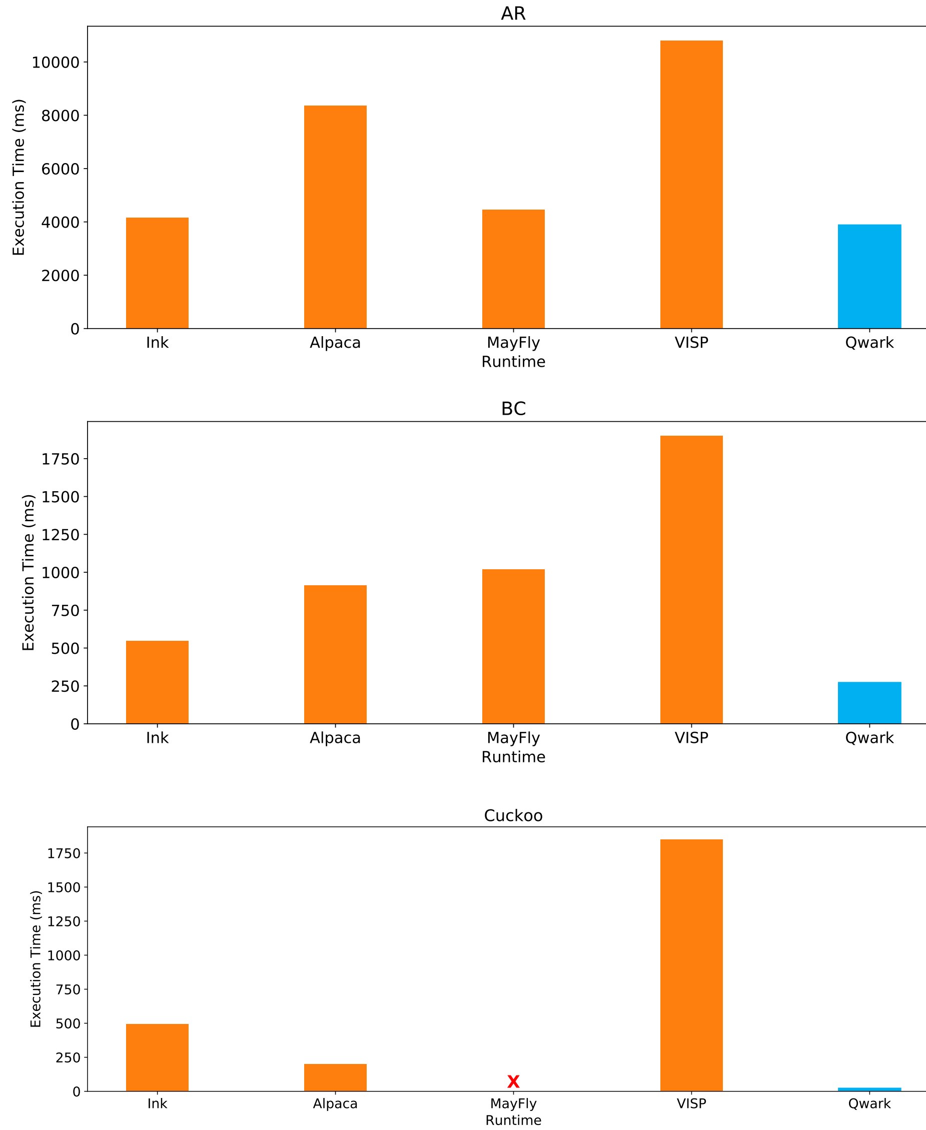

Figure. 5.6: **Qwark comparison against software-based systems.** Red cross (✗) denotes cuckoo cannot be implemented in Mayfly due to loops not being allowed in its programming model. All benchmarks are executed on continuous power, under O2 optimization and compared with Qwark's complete implementation. Results show a similar or better performance compared to task-based systems and significantly better results than VISP, a checkpoint-based system, showing Qwark's ability to outperform current systems.

Chapter 6

Discussion

By doing this research, it has been possible to design a hardware architecture that transparently deals with the challenges introduced by intermittency. Additionally, it has been possible to show the benefit of relying on structural support, indicating the amount of improvement to be expected without introducing significant overhead. Nevertheless, Qwark's implementation is still far from perfect as there is still plenty of room for optimization and re-engineering in the hands of expert designers. Instead, this thesis aims to serve as a foundation and motivation for future research and development on this domain. Because of this, it is crucial for this work to point out the limitations and opportunities identified during the development of this work.

In spite of being a stable and reliable platform, the interrupt latency observed from the OpenMSP430 [48] imposed significant limitations on Qwark's functionality. Several situations were observed during development, in which upon triggering an interrupt request, the execution unit took several instructions before servicing the request. This situation implies that upon reaching a buffer-full condition, untracked instructions could still break idempotency. To relieve back-pressure, and solve this problem, Qwark's implementation introduced additional locations on the Read/Write buffers to prevent further accesses from directly modifying non-volatile data. Nevertheless, in spite of successfully solving the problem for the utilized benchmarks, more analysis would be necessary to guarantee proper operation under every possible scenario. Moreover, interrupt latency also impacts the stack promotion implementation. For this reason, additional mechanisms and calculations became necessary to ensure correct execution, at the cost of introducing more hardware logic. Future work should aim to minimize hardware logic, reduce the implementation complexity, and maximize performance. One alternative could be re-engineering the openMSP430 execution unit to service interrupt requests as fast as possible, or rely on other architectures, such as ARM's Nested Vector Interrupt Controller (NVIC) [7]. Additionally, other open-source MSP430 [26] compliant solutions, such as the NEO-MSP430 [46] could be investigated to identify the ideal platform for future developments.

Furthermore, relying on redo logging created the need for coherently updating the segment boundary at each checkpoint, ultimately reducing the amount of improvement that could be achieved by stack promotion. Future work should explore the possibility of redesigning Qwark to allow for checkpoints to be triggered only by promotion requests. One alternative would be to rely on undo logging. Nevertheless, address translation would not be suitable for such an architecture. Therefore, to identify and develop the mechanisms to achieve this separation is future research work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work presented Qwark, a peripheral which utilizes lightweight architectural components to deal with the problems introduced by intermittency, ensuring correctness and forward progress across frequent power failures. Furthermore, stack promotion proved to be a reliable mean to guarantee a worst-case checkpoint time, effectively reducing run-time overhead, and non-volatile utilization, without significantly increasing the amount of hardware logic nor power consumption. Moreover, Qwark's results outperformed current state-of-the-art software-based solutions, effectively demonstrating the improvements which can be expected from introducing structural support to current embedded processors (e.g., MSP430).

Intermittent computing is an exciting and challenging research domain which promises to enable novel applications to run completely battery-less. Nevertheless, the scientific community is still far from developing a fully integrated solution. Because of this, this work aims to bring this reality one step closer, hopefully serving as a baseline for future research and development in this domain, looking forward to a future where ubiquitous intermittent computing is a reality.

7.2 Future Work

In spite of the promising results obtained by this research, Qwark is still far from being a full-fledged solution for battery-less applications. Most research on intermittent computing (including this) has focused primarily on preserving processor state and consistency, disregarding entirely peripheral state retention. Sytare [21] is the first known work to address peripheral support in transiently-powered systems, presenting a software framework that allows the usage of conventional internal peripherals (e.g., ADC, UART). Contrarily, RESTOP [4] proposes a software framework to enable external peripheral's state to be restored upon power failures. Samoyed [39] offers a solution for peripheral state persistence based in Just-In-Time (JIT) checkpoints. Nevertheless, a viable solution to fully support peripheral state retention is yet to be found. Moreover, at the time of writing, there is no hardware-based solution for peripheral support, making this an exciting future research direction.

Furthermore, sensor data taken between checkpoints might not be meaningful after prolonged power outages. This issue introduces the notion of *timeliness*. To deal with this, Mayfly [32] proposes a language and runtime, making use of a remanence timekeeper [33] to combine execution fragments and form coherent sensing schedules. Nevertheless, Mayfly relies on task-based transformation, making it an unsuitable solution for current state-of-the-art approaches. To address this shortcoming, VISP [40] introduces timeliness in checkpoint-based systems, making use of annotations to denote expiration constraints, utilizing a remanence timekeeper to keep track of time in spite of frequent power failures. Nevertheless, VISP's performance falls behind current software-based checkpoint approaches [38]. Perhaps, an integration of VISP's compiler support for timeliness could provide Qwark a mean to deal with this shortcoming, balancing software and hardware to get closer to a full-fledged solution.

Bibliography

[1] Alexei Colin and Brandon Lucia. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. OOPSLA*, Amsterdam, The Netherlands, November 2–4, 2016. ACM.

[2] Adi Candra Swastika, Resa Pramudita, and Rifqy Hakimi. IoT-Based Smart Grid System Design for Smart Home. In *Proc. ICWT*, Palembang, Indonesia, July 27-28, 2017. IEEE.

[3] Alanson Sample, Dan Yeager, Pauline Powledge, Alexander Mamishev, and Joshua Smith. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Trans. Instrum. Meas.*, 57(11), November 2008.

[4] Geoff V. Merrett Alberto Rodriguez Arreola, Domenico Balsamo and Alex S. Weddell. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. In *Sensors*, January 2018. MDPI.

[5] Altera. DE0-Nano User Manual. `https://www.ti.com/lit/ug/tidu737/tidu737.pdf`, 2013. Last accessed: July, 2019.

[6] James Anthony Kitchener. *Subthreshold and Near-Threshold Techniques for Ultra Low-Power CMOS Design*. PhD thesis, The University of Adelaide, Adelaide, Australia, 2015.

[7] ARM. Cortex-M4 - Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf`, 2010. Last accessed: August, 2019.

[8] Benjamin Ransford and Brandon Lucia. Non-Volatile Memory is a Broken Time Machine. In *Proc. MSPC*, Edinburgh, Scotland, June 13, 2014. ACM.

[9] Benjamin Ransford, Jacob Sorber and Kevin Fu. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Proc. ASPLOS*, Newport Beach, CA, USA, March 5–11, 2011. ACM.

[10] Jeremy Bennet. The MiBench testsuite, extended for use in general embedded environments. `https://github.com/embecosm/mibench`, 2012. Last Accessed: July, 2019.

[11] Brandon Lucia and Benjamin Ransford. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. PLDI*, Portland, OR, USA, June 13–17, 2015. ACM.

[12] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL*, Asilomar, CA, USA, May 7–10, 2017. LIPIcs.

[13] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart Dust: communicating with a cubic-millimeter computer. *Computer*, 34(1), January 2001.

[14] Daniel Yeager, Jeremy Holleman, Richa Prasad, Joshua Smith and Brian P. Otis. NeuralWISP: A Wirelessly Powered Neural Interface With 1-m Range. *IEEE Trans. Biomed. Circuits Syst*, 3(6), December 2009.

[15] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Syst. Lett*, 7(1), November 2015.

[16] Domenico Balsamo, Alex Weddell, Anup Das, Alberto Rodriguez Arreola, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 35(12), March 2016.

[17] Everspin Technologies. Comparing Technologies: MRAM vs. FRAM. *Everspin White Paper*, March 2013.

[18] Farshad Firouzi, Bahar Farahani, Mohamed Ibrahim, and Krishnendu Chakrabarty. From EDA to IoT eHealth: Promise, Challenges, and Solutions. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 37(12), December 2018.

[19] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More. Lecture Notes in Computer Science, vol 6462*. Springer, Berlin, Heidelberg, 2010.

[20] Geoff V. Merrett and Bashir M. Al-Hashimi. Energy-Driven Computing: Rethinking the Design of Energy Harvesting Systems. In *Proc. DATE*, Lausanne, Switzerland, March 27-31, 2017. IEEE.

[21] Gutier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset and Guillaume Salagnac. Peripheral State Persistence for Transiently-Powered Systems. In *Proc. GIoTS*, Geneva, Switzerland, June 6-9, 2017. IEEE.

[22] H. Jayakumar, A. Raha and V. Raghunathan. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations Across Power Cycles in Transiently Powered Computers. In *Proc. VLSID*, Mumbai, India, January 5-9, 2014. IEEE.

[23] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 26(2), February 2007.

[24] Texas Instruments. MSP430 Family Architecture Guide and Module Library. `http://www.compendiumarcana.com/forumpics/MSP430%20Family%20Architecture%20Guide%20and%20Module%20Library.pdf`, 2005. Last Accessed: July, 2019.

[25] Texas Instruments. Msp430 embedded application binary interface - application report. `http://www.ti.com/lit/an/slaa534/slaa534.pdf`, 2013. Last Accessed: August, 2019.

[26] Texas Instruments. Msp430fr57xx family user's guide. `http://www.ti.com/lit/ug/slau272d/slau272d.pdf`, 2018. Last Accessed: August, 2019.

[27] Texas Instruments and Mitto Systems. Msp430 gcc. http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/6_0_1_0/index_FDS.html, 2018. Last Accessed: August, 2019.

[28] James M. Gilbert and Farooq Balouchi. Comparison of Energy Harvesting Systems for Wireless Sensor Networks. In *International Journal of Automation and Computing, vol 5*. Institute of Automation, Chinese Academy of Sciences, Beijing, China, 2008.

[29] Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R. Smith. Wisent: Robust Downstream Communication and Storage for Computational RFIDs. In *Proc. INFOCOM*, San Francisco, CA, USA, April 10-15, 2016. IEEE.

[30] Joel Van Der Woude and Matthew Hicks. Intermittent Computation without Hardware Support or Programmer Intervention. In *Proc. OSDI*, Savannah, GA, USA, November 2-4, 2016. USENIX.

[31] Josiah Hester and Jacob Sorber. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proc. SenSys*, Delft, The Netherlands, November 6-8, 2017. ACM.

[32] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proc. SenSys*, Delft, The Netherlands, November 6-8, 2017. ACM.

[33] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. Persistent Clocks for Batteryless Sensing Devices. *ACM TECS*, 15(4), August 2016.

[34] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture Exploration for Ambient Energy Harvesting Nonvolatile Processors. In *Proc. HPCA*, Burlingame, CA, USA, February 7-11, 2015. IEEE.

[35] Kamarul Zaman Panatik, Kamilia Kamardin, Sya Azmeela Shariff, Siti Sophiayati Yuhaniz, Noor Azurati Ahmad, Othman Mohd Yusop, and SaifulAdli Ismail . Energy Harvesting in Wireless Sensor Networks: A Survey. In *Proc. ISTT*, Kuala Lumpur, Malaysia, November 28-30, 2016. IEEE.

[36] Kasim Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proc. SenSys*, Shenzhen, China, November 4-7, 2018. ACM.

[37] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA*, Vancouver, BC, Canada, October 22-27, 2017. ACM.

[38] Kiwan Maeng and Brandon Lucia. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proc. OSDI*, Carlsbad, CA, USA, October 8-10, 2018. USENIX.

[39] Kiwan Maeng and Brandon Lucia. Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints. In *Proc. PLDI*, Phoenix, AZ, USA, June 22-26, 2019. ACM.

[40] Vito Kortbeek. Dependable dynamic checkpoints for batteryless devices. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2019.

[41] Linghe Kong, Muhammad Khurram Khan, Fan Wu, Guigai Chen, and Peng Zeng. Millimeter-Wave Wireless Communications for IoT-Cloud Supported Autonomous Vehicles: Overview, Design, and Challenges. *IEEE Commun. Mag.*, 55(1), January 2017.

[42] M. R. Palacin and A. de Guibert. Why do batteries fail? *Science*, 351(6273), February 2016.

[43] Matthew Hicks. Clank: Architectural Support for Intermittent Computation. In *Proc. ISCA*, Toronto, ON, Canada, June 24-28, 2017. ACM.

[44] Mike Salas. Sub-Threshold Design - A Revolutionary Approach to Eliminating Power. *Ambiq Micro White Paper*, December 2014.

[45] Naveed Anwar Bhatti and Luca Mottola. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proc. IPSN*, Pittsburgh, PA, USA, April 18-21, 2017. IEEE.

[46] S. Nolting. The NEO430 Processor. `https://github.com/stnolting/neo430`, 2019. Last Accessed: August, 2019.

[47] Oliscience. Opencores. `https://opencores.org/`, 2019. Last Accessed: August, 2019.

[48] Olivier Girard. The OpenMS430. `https://opencores.org/projects/openmsp430`, 2018. Last Accessed: August, 2019.

[49] Saad Ahmed, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, Naveed An- war Bhatti, and Luca Mottola. Towards smaller checkpoints for better in- termittent computing. In *Proc. IPSN*, Porto, Portugal, November 11-13, 2018. ACM.

[50] Saleae. Saleae User's Guide. `http://downloads.saleae.com/Saleae+Users+Guide.pdf`, 2018. Last Accessed: July, 2019.

[51] Saman Naderiparizi, Aaron N. Parks, Zerina Kapetanovic, Benjamin Ransford and Joshua R. Smith. WISPCam: A Battery-Free RFID Camera. In *Proc. International Conference on RFID.*

[52] Sara S. Baghsorkhi and Christos Margiolas. Automating Efficient Variable-Grained Resiliency for Low-Power IoT Systems. In *Proc. CGO*, Vienna, Austria, February 24-28, 2018. ACM.

[53] Fujitsu Semiconductor. FRAM Guide Book. `https://www.fujitsu.com/downloads/MICRO/fme/fram/FRAMGuideBooksept05.pdf`, 2005. Last Accessed: February, 2019.

[54] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Exploiting Reference Idempotency to Reduce Speculative Storage Overflow. *ACM TOPLAS*, 28(5), September 2006.

[55] Shivangi Vashi, Jyotsnamayee Ram, Janit Modi, Saurav Verma, and Chetana Prakash. Internet of Things (IoT): A vision, architectural elements, and security issues. In *Proc. I-SMAC*, Palladam, India, February 10-11, 2017. IEEE.

[56] Stephen Williams. Icarus Verilog. `http://iverilog.icarus.com/home`, 2013. Last Accessed: July, 2019.

[57] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A Survey on the Edge Computing for the Internet of Things. *IEEE Access*, 6(1), November 2017.