HIGH-PERFORMANCE AND RELIABLE INTERMITTENT COMPUTATION

by

Jongouk Choi

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Changhee Jung, Chair

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Dr. Z. Berkay Celik

Department of Computer Science

Dr. Muhammad Shahbaz

Department of Computer Science

Approved by:

Dr. Kihong Park

ACKNOWLEDGMENTS

First of all, I would like to thank our Lord, Jesus Christ, who has delivered me and granted countless blessing. As always, our Lord is magnificent, eternally glorious, and faithful. Thanks to His guidance, I have been able to complete this dissertation. In particular, I would like to thank to my wife, Sujeoung. There has been times I felt down. But in the disappointing hours, she encourages me, comforts me, and lets me overcome.

I am deeply indebted to my advisor, Prof. Changhee Jung. He introduced me to an interesting research domain, energy harvesting system research, provided me challenges to improve my writing/teaching/presentation skills, and helped me out with brilliant ideas. I have learned a lot lessons from him, not only how to find a research problem, how to see a problem from a different angle, how to solve a problem, but also how to be a good scholar, teacher, and mentor. I owe thanks to the committee members of my dissertation, Prof. Pedro Fonseca, Prof. Z. Berkay Celik, and Prof. Muhammad Shahbaz. They all gave invaluable feedback and spent their time to shape this dissertation. I am also grateful to Prof. Dongyoon Lee, Prof. Changwoo Min, Prof. Amro Awad, Dr. Hyunwoo Joe, and Dr. Yongjoo Kim for their advice and support that helped me to develop this work. I would also like to thank Dr. Doug Joseph, Dr. Casey Battaglino, Dr. Alejandro Rico, and Prof. Fredrik B. Kjolstad for their mentorship during my internship at ARM research.

I also owe thanks to my lab-mates and colleagues during my Ph.D journey, Qingrui Liu, Tong Zhang, James Davis, Xinwei Fu, Hongjune Kim, Larry Kittinger, Jianping Zeng, Shao-Yu Huang, Yuchen Zhou, and Byounguk Min. I thank my church, Purdue Korean Presbyterian Church where I spent wonderful tine in Christ. Lastly, I thank my family for their unconditional love that made this work possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

An energy harvesting system (EHS) provides the intriguing possibility of battery-less computing and enables various applications such as wearable, industrial or environmental sensors, and implantable medical devices. The biggest challenge of EHS is the instability of energy sources (e.g., Wi-Fi, solar, thermal energy, etc.) which causes unpredictable and frequent power outages. To address the challenge, existing works introduce software-based and hardware-based power failure recovery solutions that ensure program correctness across a power outage. However, they cause a significant performance overhead without providing the high quality of service in reality, and suffer from a reliability issue. In this dissertation, we address the limitations of recovery solutions across the system stack, from the compiler-directed approach and run-time systems to hardware mechanisms, and demonstrate the effectiveness of the approaches using real EHS platforms and simulators. We first present software-based recovery solutions by leveraging compiler support. We develop a compiler-directed solution built upon commodity EHS platform that can achieve 3X speedup compared to the software-based state-of-the-art solution. We also introduce a compiler optimization technique that can cooperate with run-time systems and hardware support, achieving 8X speedup compared to the software-based solution. We then present hardware-based recovery solutions by leveraging compiler and hardware support. We develop an architecture/-compiler co-design solution that re-purposes existing hardware components in a core for power failure speculative execution, a new speculation paradigm, and leverages a novel compiler analysis for correct power failure recovery. Our result highlights $2 \sim 3$x performance improvement compared to the hardware-based state-of-the-art solution without requiring hardware modification. Next, we present a new cache design for EHS that can achieve cost-effective, high-performance intermittent computing. According to experimental results, the new cache design outperforms the state-of-the-art cache scheme by 4X and reduces the hardware cost by 90%. Finally, we present an operating system (OS)-driven solution to address a reliability problem on EHS devices while all existing works are vulnerable, causing the wrong recovery across power failure. Our experiments demonstrate that the solution causes less than 1% run-time overhead and successfully addresses the reliability problem without compromising correct power failure recovery.

1. INTRODUCTION

The number of devices connected to the Internet of Things (IoT) continues to grow, enabling new application domains such as smart homes, buildings, structures, and cities. Powering these IoT devices is a key challenge since their batteries are bulky, have a limited lifetime, and are expensive to replace. This has brought a tremendous amount of interest in energy harvesting system (EHS) that extracts ambient energy from surroundings for free, e.g., sunlight, radio waves, vibration, and thermal gradients, and provides the intriguing possibility of battery-less computing. Thanks to its self-powering, maintenance-free, and environmentally-friendly nature, EHS is the logical next step in the evolution of IoT.

Due to the unstable nature of harvested energy and the absence of a battery, EHS leverages a tiny capacitor as an energy buffer. It intermittently computes when it provides sufficient energy, which would otherwise die, thus being called *intermittent computation*. This implies that frequent power interruptions become the norm of program execution, producing incorrect output or failing to provide any service to end-users. Hence, researchers adopt low-power microcontroller (MCU) with byte-addressable nonvolatile memory (NVM) and offer some form of recovery support to checkpoint (backup) necessary data and restore them across power outages [1]–[9]. From the system administrator's perspective, however, it is uniquely challenging to design hardware and develop software for EHS. This is because merely ensuring correct power failure recovery is insufficient to bring EHS to the IoT applications; major challenges remain in run-time overhead, energy efficiency, and reliability.

To achieve high-performance and reliable intermittent computing, we design, develop, and evaluate compiler-directed approaches, architecture/compiler co-design recovery mechanisms, OS-driven reliability solution, and hardware support across the system stack, showing that they help existing EHS platforms to deliver high quality of service to end-users.

1.1 Challenges of Energy Harvesting Systems

The correct recovery challenge is at the heart of EHS. To achieve power failure recovery, EHS often creates a checkpoint on which the volatile registers are saved into the NVM to roll back to the most recently checkpointed states when the power comes back after an outage. Nevertheless,

this simple checkpointing mechanism alone cannot consistently achieve correct recovery due to the inconsistency of data in NVM. NVM data can be corrupted across power failure when there exists write-after-read (WAR) dependence [4], [10]. For example, increasing an NVM-allocated variable `i`, whose initial value is 0, after the checkpointing of registers (i.e., `checkpoint;i++`) should generate 1 as the output value of `i` provided the code executes without power failure. However, if the code is power-interrupted after `i` is updated to 1 in NVM with the increment, the rollback recovery—in the wake of power failure—restarts the code from the checkpoint point and thus ends up re-doing the increment, thereby generating wrong output 2. Thus, such WAR-induced memory inconsistency makes EHS fail to achieve correct power failure recovery.

With that in mind, researchers and practitioners have designed various power failure recovery solutions. While they ensure correct recovery across frequent power failure, there are three critical challenges still hindering the wide adoption of such solutions in practice: (1) run-time overhead, (2) energy overhead with expensive hardware cost, and (3) reliability issues. This section elaborates on these problems and discusses the relevant state-of-the-art solutions for each category. More discussion on other related works is deferred to Chapter 2.

1.1.1 Run-time Overhead of Recovery Solutions

To ensure correct recovery, researchers develop many software and hardware solutions; however, adopting the solutions in IoT applications is complex since they incur high run-time and energy (power) overheads. The software solutions caused a significant slowdown, and the hardware solutions were cost-ineffective and energy-inefficient. The following two problem statements (PS) summarize these two overhead problems.

PS 1. *Software recovery solutions cause a high run-time overhead.*

Existing software-based recovery schemes partition program into a series of recoverable regions (tasks) by checkpointing/logging their input register/memory data in NVM. If any region is interrupted due to power failure, the recovery schemes, in the wake of the failure, first restore the checkpointed/logged data by loading them from NVM and then resume the program at the beginning of the interrupted region [2], [4], [11], [12]; this is so-called rollback recovery.

Unfortunately, the existing recovery schemes are not systematic since they form their regions sometimes too conservatively or aggressively. If regions are too short (i.e., unnecessarily making frequent checkpoints at each region boundary), the schemes consume more energy for checkpointing but use less energy for computing. This is because checkpoints are essentially NVM stores, which are the most energy-consuming instructions in EHS. While one could take an aggressive approach by forming long regions for fewer checkpoints, expensive re-execution overhead has to be paid by restarting such a long region, possibly many times across power outages. Either way, the forward execution progress is limited, leading to significant performance degradation. Even worse, the existing recovery schemes could suffer from a *stagnation* problem [8], [9]—livelock-like situation where power failure repeatedly occurs before some long region finishes—making no forward progress despite continuous energy consumption (also called a non-terminating bug [13]).

PS 2. *Hardware recovery solutions are energy-inefficient and cost-ineffective.*

Hardware recovery solutions introduce nonvolatile processors (NVP) [3], [14]–[18]; however, it is hard to adopt the NVPs to real EHS devices since they require expensive hardware modification and consume hard-won energy inefficiently. First, unlike traditional embedded devices, NVPs use specialized hardware components for correct recovery. NVPs rely on a voltage monitor based on a just-in-time (JIT) checkpoint mechanism that checkpoints volatile registers—when the voltage monitoring system detects the voltage drop below a defined threshold—by using the buffered energy in the capacitor. In addition to the voltage monitor, the schemes require non-trivial hardware modifications such as nonvolatile flip-flops, that must be laid out next to volatile flip-flops for fast backup/restoration, special hardware checkpoint/controller logic, and additional capacitors for the voltage monitor. Second, existing works reserve a significant amount of energy for checkpointing purposes, which cannot be used for forward progress execution. This is mainly because the voltage monitor may cause stability issues such as excessive leakage or crack of the capacitors leading not only to reduced capacitance [19], [20] but also to voltage detection delay with unexpected cold-start glitch [21]. Consequently, they waste hard-won energy, making no forward progress until such a high voltage is secured to wake up the system for sure.

Furthermore, NVPs leverage the same JIT checkpoint mechanism to enable a volatile cache, that has a high potential to improve performance for EHS devices. Given an energy budget, EHS can make a further forward progress by avoiding NVM accesses on cache hits. However, all dirty cache lines are lost upon a power failure so the NVM state upon power failure could be inconsistent, causing incorrect program behavior when the program resumed. To ensure crash consistency, a traditional SRAM-based write-through cache can be used in EHS devices without modification since it naturally supports crash consistency by persisting data at every store in a synchronous manner while updating the same data in the cache—though the requirement of synchronous writes lead to the long store latency as in the case without a cache.

For crash consistency, NVCache [22], [23] is designed as a full non-volatile cache, instead of a traditional SRAM-based volatile one. However, NVCache is inevitably slower and requires more energy than a traditional SRAM-based cache. Finally, the state-of-the-art introduces NVSRAM-Cache [24]–[27] that couples a traditional write-back SRAM cache with an NVM counterpart. It achieves crash consistency via JIT checkpointing; it monitors a remaining energy in a capacitor (energy buffer) and copies the SRAM cache states to the NVM counterpart right before a power loss. NVSRAMCache can achieve higher performance improvement as it uses write-back policy and absorb write hits (unlike WTCache) and it uses a SRAM-based cache at runtime (unlike NVCache). However, they require to reserve more energy for failure-atomic JIT checkpointing, causing an energy efficiency issue. Moreover, the NVM counterpart is underutilized since it is only required for checkpointing purpose.

1.1.2 Reliability Issue of Recovery Solutions

EHS leverages a capacitor as an energy buffer, achieving the full potential of maintenance-free batteryless IoT. However, we found that a capacitor can be degraded in real energy harvesting settings. Since ambient energy sources are unstable, causing frequent power failure, the capacitor is repeatedly charged and discharged, which is a stressful condition expediting the capacitor degradation [19], [28]–[30]. Furthermore, since energy harvesting systems are used as IoT devices in various environments, their capacitors are often exposed to other stressful conditions, e.g., high humidity/temperature [28], [31]. Under the circumstances, the capacitor is seriously degraded over

time and finally unable to buffer enough energy to run applications; we refer to the problem as a *capacitor error*.

PS 3. *Both software and hardware recovery solutions are vulnerable to the capacitor reliability issue.*

Due to the insufficient amount of buffered energy in the degraded capacitor, software recovery solutions can fail the forward progress execution. Since the solutions generate a series of recoverable program regions, the statically-formed program region size can be too long to be completed within a given energy budget in the degraded capacitor, suffering from the stagnation problem. To address the problem, they can generate shorter regions; however, they require additional checkpoint stores, spending hard-won energy inefficiently and causing a significant run-time overhead (by more than 5x slowdown compared to original recovery solution [12]).

On the other hand, when the capacitor does not buffer sufficient energy due to the reliability issue, hardware solutions fail the JIT checkpointing, corrupting/losing volatile data (without providing any further service) across power failure. To ensure safe JIT checkpointing, existing works reserve 10% checkpoint voltage margin in case of unexpected errors [3], [6], [7], [14]. However, it is impractical to leverage the large safe margin for safe JIT checkpointing. Since the increased voltage/energy can only be used for checkpointing purposes, not for computation/progress, it also causes significant performance degradation. Moreover, such a large safe margin can further increase the start-up voltage level, leading to a lot longer reboot/recharge time to secure enough energy across power failure. More importantly, although they reserve such a large safe margin, the prior works are still not free from the reliability issue. Since the capacitor can eventually be degraded by more than 50% within a year in real energy harvesting situation, the large safe margin can finally be unavailable for the reliability problem.

1.2 Thesis Statement

This thesis aims to solve the research challenges mentioned above and design a high-performance and reliable system stack that can support IoT applications on EHS devices. The scope of the system stack spans from compiler analysis to mechanisms in the device's hardware and includes operating system support for diagnosis and maintenance. The system brings value to the application by

providing high-quality service without effort from programmers. Once incorporated into existing EHS platforms, the system stack we introduce in this thesis serves as evidence for the following thesis statements (TS):

TS 1: *To address the performance problem in software recovery solutions, we leverage compiler-directed approaches with hardware support available in commodity hardware (originally designed for other purposes).*

Software solutions statically partition program into a series of recoverable regions, the boundary of which serves as a rollback recovery point in case the following region is interrupted by power failure [12], [32], [33]. The implication is that they make a checkpoint that entails multiple energy-consuming NVM writes at every (fixed) pre-defined task boundary, which would be unnecessary under stable energy-harvesting conditions. The crux of the problem is that due to the compile-time fixed tasks, they always need to checkpoint data at every boundary, even when there is no power failure, causing a significant slowdown (50∼400%). Also, the solutions must roll back to the beginning of the interrupted task across power failure, and re-execute the same instructions causing the re-execution penalty [10], [34], [35]. In the worst case, they can suffer from the stagnation problem by repeatedly re-executing the same interrupted region across power failure. Instead of using the region boundaries, we see an opportunity to make a recovery point flexibly by leveraging a timer-based checkpoint scheme with a copy-on-write (CoW) memory protection mechanism. In Chapter 3, we demonstrate how the solution can be used to ensure correct power failure recovery without requiring program region formation. Furthermore, we also study compiler analysis to remove unnecessary checkpoint stores and eliminate the re-execution penalty caused by rollback recovery. In Chapter 4, we show how to leverage compiler optimization techniques for checkpoint reduction and rollback-free intermittent execution.

TS 2: *To address energy-efficiency and cost-effectiveness problems in hardware solutions, we leverage software support while improving run-time performance.*

To ensure correct power failure recovery, researchers propose hardware solutions (NVPs) with a capacitor-backed JIT checkpointing that checkpoints volatile registers to non-volatile counterparts placed right next to a register file by spending buffered energy in a capacitor. To achieve correct recovery without requiring unconventional architectural support, we developed a new spec-

ulation paradigm called power failure speculation with the help of compiler support. Instead of using the non-volatile counterpart with extra energy, we find a way to ensure correct power failure recovery by leveraging a compiler-directed approach. For example, we study an architecture/-compiler co-design scheme that works for commodity in-order processors used in EHS devices. The speculation assumes that power failure will not occur, and thus a processor holds all committed stores—as if they were speculative—in case of mispeculation. If power failure occurs during speculative execution, all speculative stores disappear. Then, when the program control reaches a compiler-directed recovery point, a processor assumes that the speculation turns out to be successful and persists all speculative stores. More details are explained in Chapter 5.

On the other hand, we see a chance to enable volatile caches on top of NVPs by leveraging a small size buffer (e.g., write-back buffer) without requiring such a non-volatile counterpart and a significant amount of extra energy. We develop a new cache architecture for EHS devices that can reduce hardware costs yet accelerate run-time performance. More details are explained in Chapter 6.

TS 3: *To address reliability problem in power failure recovery solutions, we leverage operating system modules that can detect the problem and prevent from it in a seamless manner.*

Both software and hardware solutions assume that a capacitor (i.e., energy buffer) in EHS is reliable; however, it turns out to be a fallacy. Indeed, capacitors can be degraded, losing their original capacitance by more than 20% regardless of their size or material—a capacitor is considered as dead at 20% degradation [28]–[31], [36]. When the capacitor is degraded, both solutions can suffer from capacitor degradation such as data loss/corruption or stagnation; we call it as a capacitor error.

To deal with the capacitor error, we study common capacitor degradation patterns in real EHS devices and apply them to build a degradation detection mechanism. However, the detection mechanism is not enough because neither software nor hardware recovery solutions can ensure forward progress execution, i.e., no service to end-users with the degraded capacitor. Fortunately, we find out that we can leverage the capacitor's resilience nature to prevent no forward progress execution issues, which is that the capacitor can restore its original capacitance by itself with the help of its resilient nature. By leveraging nature, we can not only detect the capacitor degradation but also re-

cover the degraded capacitor and keep providing high-quality service as a practical capacitor error resilience solution. In Chapter 7, we present an OS-driven capacitor error resilience solution built upon this analysis.

1.3 Contributions

In this thesis, we focus on solving three problems: (PS1) the run-time overhead issue in software solutions, (PS2) the energy overhead and the hardware cost issue in hardware recovery solutions, and (PS3) the reliability issue in EHS devices. We envision that the contributions made in this thesis can be applied to real EHS devices to improve their performance and efficiency. We summarize our efforts in solving the problems as contributions in the following sections.

- We propose a compiler-directed rollback recovery solution, which strongly guarantees forward execution progress across power failure by leveraging a boundary-free checkpointing mechanism. It requires neither user intervention nor program partitioning for region (task) formation while its boundary-free checkpointing can maximize the forward progress; our implementation is released in the Elastin library [8]. Experimental results show that Elastin is able to complete all benchmark applications, whereas the state-of-the-art work cannot due to stagnation.
- We develop a compiler-directed program analysis tool that statically identifies the no-forward-progress (stagnation) problem in intermittent programs and decomposes a program into stagnation-free regions to ensure it makes forward progress when executed intermittently. Within the analysis tool, we develop a compiler optimization technique that can cooperate with a run-time system and hardware support. The technique makes each program region free from power failure interrupt, achieving stagnation-free and power-failure-interrupt-free intermittent execution. We implemented the analysis within the LLVM framework and released it as RockClimb [37].
- We build an architecture/compiler co-design for hiding long latency of NVM writes and improving performance during intermittent program execution. This solution enables a new speculation paradigm called power failure speculation. By leveraging the speculation, the solution overlaps NVM writes with the next instructions' execution. Such instruction level

parallelism (ILP) gives an illusion of out-of-order execution on top of the in-order processor. We develop the co-design named CoSpec [9] and demonstrate the improved performance and correct recovery that the new speculation brings.

- We develop specialized cache architecture with a new write policy for power-hungry EHS. Without requiring non-volatile counterparts or a huge amount of extra energy, the proposed cache architecture can energy-efficiently and cost-effectively persist volatile data in NVM across power failure. In particular, we build the cache architecture upon traditional SRAM cache design, taking advantages of both write-back's efficiency and write-through's persistence. It adaptively behaves as a write-through or a write-back cache by changing its characteristic back and forth with a help of run-time system. We introduce the new cache design named as Write-light Cache and demonstrate the performance improvement and cost reduction.
- We discover a capacitor error where all EHS devices can corrupt/lose their data or fail to provide any service when their energy buffer (i.e., capacitor) is degraded. To address the error, we introduce an OS-driven capacitor error resilience solution called CapOS, that can preserve volatile data against capacitor errors and recover the degraded capacitor by leveraging its self-recovery nature. Our experiments demonstrate that CapOS causes less than 1% performance overhead on average compared to a (unprotected) roll-forward recovery when there is no capacitor error. In the presence of capacitor errors, CapOS incurs only 15% performance overhead on average.

1.4 Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses related work for architectural design of EHS, software- and hardware-based power failure recovery solutions, and capacitor reliability problem. Each of the following chapters elaborates on each challenge of intermittent computing briefly outlined in the introduction and presents our contribution that addresses it in detail. These chapters are ordered by the abstraction level, from software to hardware. Chapter 3 presents a software recovery solution, introducing a compiler-directed recovery scheme that can ensure stagnation-free intermittent execution without requiring fixed region boundaries. Chapter 4

presents a compiler-directed high-performance intermittent computation scheme with power failure immunity, that enables rollback-free and power-failure-interrupt-free execution by leveraging static analysis with run-time system and hardware support. Chapter 5 presents a compiler/architecture co-design recovery solution, that introduces a new speculative execution paradigm called power failure speculation, enabling the ILP execution for performance improvement. Chapter 6 introduces a new cache architecture for high-performance intermittent computing in EHS devices. Chapter 7 presents an OS-drive capacitor error resilience solution that detects the capacitor error and preserve all data against the error. We conclude in Chapter 8.

2. BACKGROUND AND RELATED WORK

In this chapter, we discuss the hardware design of EHS devices and their capabilities and limitation on the execution of program. Then, we focus on the power failure recovery problem and discuss related works. We first study software-based recovery solutions, which includes compiler support and a new programming model, and discuss their limitation. We then investigate existing hardware solutions for power failure recovery with a voltage monitor based JIT checkpointing mechanism. Finally, we discuss a reliability problem in EHS devices.

2.1 Energy Harvesting System Platform

Given the power-hungry nature of EHS devices, existing platforms have opted for extremely low-power in-order processor architecture, leveraging a low-power MCU such as TI-MSP430 [38]. Despite the low performance of in-order processor architecture, it is more suitable than power-consuming and complex out-of-order processor architecture [39], [40]—though some prior works propose to use out-of-order processors or even hybrid cores equipped with both in-order and out-of-order pipelines by assuming strong energy harvesting source that can deliver stable power for the out-of-order execution. With the low-power processor architecture, the platforms use a tiny capacitor as energy storage to intermittently compute only when sufficient energy is buffered in the capacitor. If the buffered energy is depleted, EHS dies due to the lack of enough energy to power the device, i.e., it is power-interrupted. In light of this execution behavior, the term *intermittent computation* [41] is often used to characterize the execution of EHS. With the intermittent computation in mind, the existing platforms use (1) byte-addressable NVM, e.g., FRAM [42], as the main memory for data to survive frequent power failure and (2) some form of crash consistency to checkpoint necessary data at run time and restore them in the wake of the power failure. We target the low-power in-order processor architecture made up of NVM without cache as prior works [4], [5], [8], [9], [14], [16], [43]. In the absence of cache, only data in a processor core, i.e., registers, are transient and will be lost when power is cut off. Thus, registers need to be checkpointed—i.e., saved in NVM—for their safe restoration in the wake of power failure.

2.2 Software-based Approaches for Power Failure Recovery

2.2.1 Rollback Recovery

Due to unreliable ambient energy sources, energy harvesting systems suffer frequent power failure that must be recovered in a crash consistent manner [4], [7], [18], [44]–[64]. To achieve crash consistency, software-based prior works partition program into a series of recoverable regions/tasks (and back up necessary data therein to NVM) so that their re-execution results in the same and correct output across power failure; hereafter, we use the term region(s) as the same meaning as task(s).

To a large extent, there are two crash consistency approaches that both require handling memory antidependence [65] also known as Write-After-Read (WAR) dependence—since it overwrites an input to be read for re-execution. First, automatic idempotent region formation scheme such as Ratchet [4], places a region boundary to cut the antidependence(s), and checkpoints live [65] registers at compile time; the checkpoints are essentially NVM store instructions. Second, programmers can alternatively partition program into a series of regions on their own by using a new programming model. The model can help users to preserve the memory locations being overwritten by antidependent stores by logging the original value to NVM, and checkpoint registers [2], [11], [12]. In the wake of power failure, these two types of prior works restart from the beginning of the interrupted region after restoring the checkpointed registers (and the logs, if necessary) from NVM.

2.2.2 Challenge in Software-based Recovery Solutions

Unfortunately, the software-based recovery solutions can have a critical performance problem. Suppose a region whose execution time is greater than the power failure period, i.e., the time between the failures. If they periodically occur with the same frequency, the program ends up rolling back to the beginning of the region indefinitely. That is because the failures keep occurring before the end of the region is reached, in which case the program just wastes harvested energy in vain making no forward execution progress. Researchers call this livelock-like phenomenon *stagnation* [8], [9], [11]–[13], [66], [67].

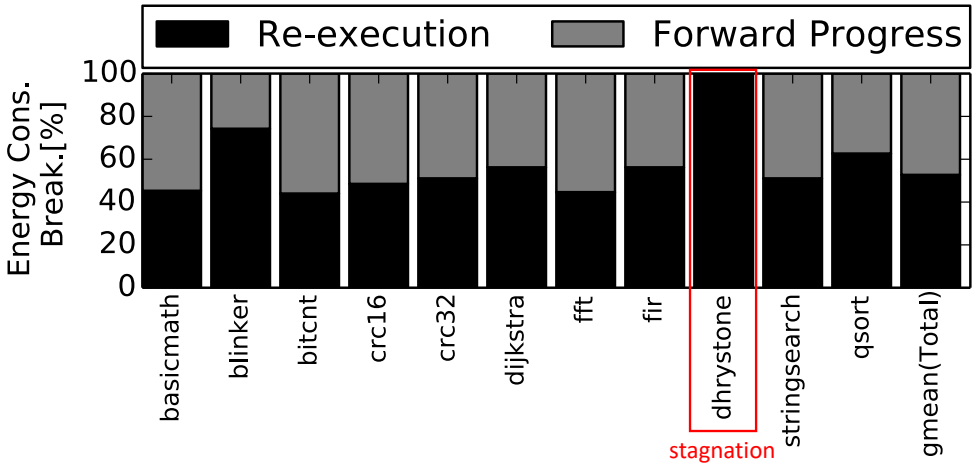


Figure 2.1. Energy breakdown of Ratchet for a real energy harvesting condition. For *dhrystone*, 100% re-execution means stagnation. A geometric mean (gmean) is calculated only for those non-stagnated.

It turns out that prior works can suffer the stagnation problem [2], [4], [68]–[72]. To investigate the phenomenon, we conducted experiments by using one of the prior works, the idempotence-based power failure recovery scheme called Ratchet [4]. We ran 11 benchmark applications on a real energy harvesting board (the evaluation setting is described in Section 4.6.1) and analyzed the cost of re-execution across power failure by breaking down the total energy consumption of each application into two parts: re-execution and forward progress as shown in Figure 5.13. Note that we disabled Ratchet's timer based checkpointing, since it could result in wrong recovery for those regions that have Write-After-Read-After-Write (WARAW) dependence [34], [65].

We discover that Ratchet can be trapped in some long regions leading to stagnation, thus never finishing the program as in the case of *dhrystone* in Figure 5.13. Even for non-stagnating applications, Ratchet ends up wasting 47-75% of hard-won energy by repeatedly checkpointing/restarting the same interrupted region across power failure—before getting out of the region. Overall, Ratchet spends 52% of the total energy consumption for re-executions, leading to significant performance degradation. The performance problem leads to the advent of hardware based approaches.

2.3 Hardware-based Approaches for Power Failure Recovery

2.3.1 Roll-forward Recovery

A root cause of the wrong recovery (memory inconsistency) is that in the wake of power failure, program control rolls back across WAR dependence, reading values updated by stores left behind the failure. The insight is that it is possible to eliminate the inconsistency if EHS moves forward for recovery instead of the roll-back. Hardware-based crash consistency schemes leverage the insight taking the roll-forward only recovery. When power comes back, the roll-forward recovery mechanism resumes the interrupted program exactly at the same failure point on which the power interruption occurred. Thus, the hardware roll-forward recovery schemes never cross WAR dependence, thereby achieving crash consistency [3], [6], [7], [14].

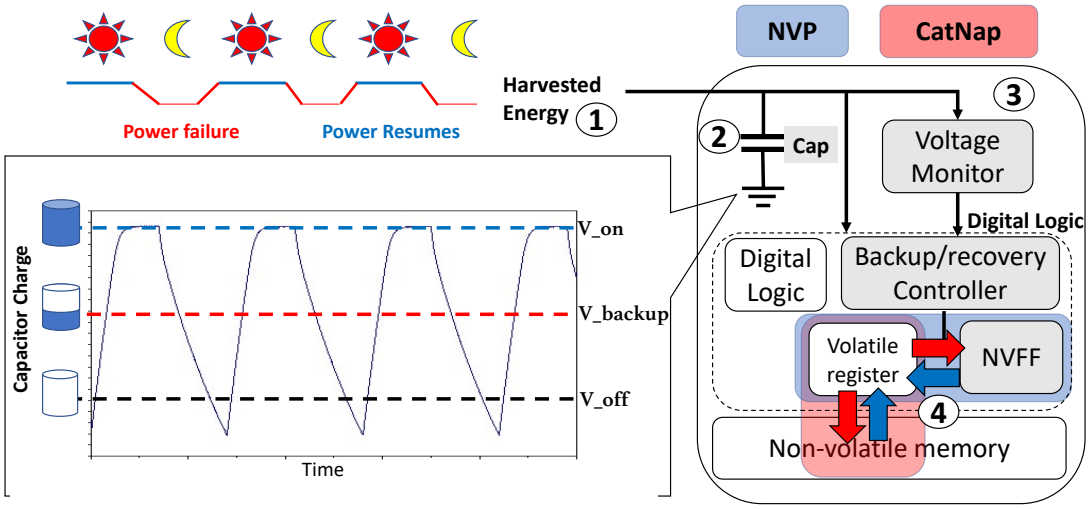


Figure 2.2. Just-In-Time (JIT) checkpointing mechanism

Figure 2.2 describes the high-level design of the hardware schemes, i.e., NVP [14] and QuickRecall [3] that are two most common EHS solutions. To achieve energy-efficient checkpoint/recovery, NVP takes advantage of a hybrid register file circuitry comprising standard flip-flops and non-volatile flip-flops (NVFF). Since the volatile and non-volatile flip-flips are laid out right next to each other in the circuit, their data movement is swift, enabling fast register checkpoint/recovery. However, such a hybrid register file requires significant microarchitecture modification. To lower the hardware design cost, QuickRecall dedicates a part of non-volatile memory (NVM) as

checkpoint storage of registers instead of using the NVFF. Both hardware schemes exploit the roll-forward recovery to achieve crash consistency based on another circuit mechanism called just-in-time (JIT) checkpointing.

The JIT checkpointing saves volatile registers to their checkpoint storage—i.e., NVFF as with NVP or NVM as with QuickRecall—when EHS is about to have a power outage. To recognize the impending power failure, the EHS leverages a capacitor and a voltage monitor. If the voltage level in the capacitor is lower than a pre-defined voltage threshold, i.e., V_{backup} for power-failure-free checkpointing of all registers, the voltage monitor assumes that power is about to be cut off. Thus, the monitor sends a signal to the controller logic that lets the processor copy all registers to their checkpoint storage, i.e., NVFF or NVM.

Note that EHS can also identify when a sufficient amount of energy is secured in the capacitor to start the processor. As shown in Fig. 2.2, NVP and QuickRecall first buffer harvested energy into the capacitor. The voltage monitoring system can then check whether the buffered energy is enough to operate the system by comparing the capacitor's current-voltage level to another pre-defined threshold V_{on}. Suppose the voltage level has become greater than the threshold since power failure. In that case, the voltage monitor sends a wake-up signal to the controller to restore checkpointed registers (from NVFF or NVM) and, in turn, resume the (interrupted) program. In summary, the JIT checkpointing enables EHS to observe both consistent NVM and register file states—since the recovery point is the same as the power-off point—and ensures the absence of roll-back, thereby always making forward progress across power failure.

2.3.2 Challenge in Hardware-based Recovery Solutions

Although the voltage monitor based JIT checkpointing mechanism can address the memory inconsistency, it is not free from cost-effectiveness and energy efficiency issues. First, the JIT checkpointing requires expensive hardware support for roll-forward recovery such as NVFF, special checkpoint/controller logic, voltage monitor, and additional capacitors for voltage monitor. Second, the JIT checkpointing wastes hard-won energy to run hardware support. Unfortunately, the voltage monitor can be unstable, due to excessive leakage or crack of the capacitors, causing voltage detection delay. If the monitor cannot detect the power failure at a right time, the JIT check-

pointing mechanism can fail to persist all required data, thereby causing wrong recovery problem. To mitigate the issues, existing works aggressively increase the voltage threshold of the system wake-up/backup. Consequently, they waste hard-won energy with making no forward progress until such a high voltage is secured to wake up the system for sure.

2.4 Reliability Problem

Although all the prior works assume that a capacitor is reliable, it turns out to be a fallacy. Indeed, capacitors can be degraded losing their original capacitance by more than 20% regardless of their size or material—a capacitor is considered as dead at 20% degradation [28]–[31], [36]. The capacitor error can be caused in following conditions. First, the humidity change can damage the capacitor. If its film is absorbed moisture, the capacitor can lose its original capacitance [28]. Second, the high operating temperature can make a negative impact on its quality of electrode metalization [28]. Third, the over voltage can degrade the capacitor film's capacity and increase leakage current flow[31] and operating temperature. Fourth, the continuous charge/discharge can also degrade the original capacitance due to electro-chemical corrosion by high operating temperature or formation of additional dielectric layer[30]. In particular, the fourth condition is normal in energy harvesting systems due to frequent power failure, i.e., the systems are naturally exposed to such stressful condition. Finally, when they are degraded, capacitor errors occur.

We found that there are two types of capacitor errors, data loss/corruption and stagnation [8]—making no forward progress in spite of continuous energy consumption (also called a *non-terminating* bug [12], [13], [33]). The data loss problem occurs when the JIT checkpoint fails to persist volatile data. On the other hand, the stagnation problem happens when the JIT checkpoint fails to make a correct recovery point (e.g., PC or SP lost); the systems can resume the interrupted program from the same or wrong recovery point across power outages.

2.4.1 Capacitor Error Experiments

To analyze the capacitor error in energy harvesting systems, we conducted experiments with a roll-forward recovery solution [6], having a 1mF supercapacitor as energy buffer on a real evaluation board (MSP430FR5994); we tested four different 1mF supercapacitors (e.g., cap.1∼4).

For realistic experiments, we developed the power generator board with MSP430FR5969 to incur power failures; the power generator supplies voltages to our target board through GPIO pins. Then, we mimicked energy harvesting situation by injecting square wave voltages [73] into the target board. For square wave voltage input, we used a power trace whose frequency was 0.3Hz at 5V; We empirically found that the frequency affects the capacitor as the most in our experimental setting by considering a resistor-capacitor (RC) time constant [74], where R is the MCU connected to the capacitor (C). At this frequency, the capacitor is charged and then discharged quickly. We periodically measured the capacitance of each capacitor.

Table 2.1. Power failure recovery solutions for energy harvesting systems. Any type of recovery solution is susceptible to the capacitor error.

Scheme	Recovery	Cap. Error	Cap. Rest.
Chinchilla [12]	Rollback	Stagnation	No
Sytare [75]	Roll-forward	Data Loss/Stagnation	No
QuickRecall [3]	Roll-forward	Data Loss/Stagnation	No
Samoyed [6]	Rollback/forward	Data Loss/Stagnation	No

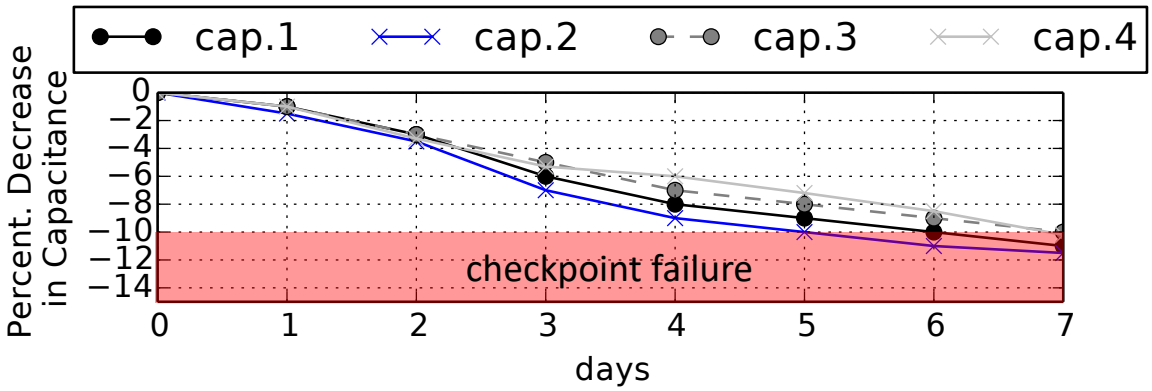


Figure 2.3. Capacitor degradation in real energy harvesting systems. Within seven days, a capacitors can be severely degraded causing capacitor error.

From the experiments, we found that *the capacitors gradually lose its original capacitance* by more than 10% as shown in Fig. 2.3 within seven days, which could cause a JIT checkpoint failure corrupting data in NVM, i.e., capacitor error. For further analysis, we also tested our real energy harvesting board (TIDA-00588), which is equipped with the on-board solar cells with 47mF capacitor as energy buffer, in real harvesting environment. We found that the capacitance could

be decreased by up to 50% within a year. Consequently, with the degraded capacitor, all the prior works suffer the capacitor error problem achieving neither correct recovery (data loss) nor forward progress execution as shown in Table 2.1. Therefore, we believe there is a compelling need to address the capacitor error problem for practical energy harvesting systems.

2.4.2 Capacitor Recovery

Although capacitors are degraded in stressful conditions, a supercapacitor can particularly recover its capacitance when it is (electrically) isolated thanks to its self-recovery nature [29], [76]. A prior work demonstrates the capacitor recovery phenomenon [29] with a recovery model defined as: $C_{recovery}(t,T,V_{end}) = a*exp(-\frac{t}{\tau_1})+b*exp(-\frac{t}{-\tau_2})$, where a and b characterize the capacitor state, and τ_1 and τ_2 are the time constants governing the recovery rate of the capacitor.

To explore the capacitor self-recovery phenomenon in energy harvesting systems, we intentionally stressed supercapacitors to be degraded. Then, when the capacitor was degraded by 10%, we electrically isolated them and measured their capacitance variation over time. Finally, we found that all of the degraded capacitors could be healed within 2 hours.

Type of Capacitors Some prior works for energy harvesting systems leverage another type of capacitor such as ceramic or electrolytic capacitor instead of using a supercapacitor [4], [12], [14]. To explore the capacitor degradation and self-recovery nature broadly, we also tested ceramic (1nF, 10nF, 47nF, and 100nF) and electrolytic (1uF, 10uF, 47uF, and 100uF) in the same environment. We found that they were all degraded causing the capacitor error problem (e.g., checkpoint failure) within at least a month as demonstrated in prior works [28], [30], [31], [36]. However, they were unable to recover their degraded capacitance when they were in idle; their capacitance recovery is highly dependent on chemical reaction in their material [77]. The reliability work in this thesis (Chapter 7) specifically target the supercapacitor as energy buffer like prior works [7], [33], [72], [78]–[83] while assuming a small capacitor for other works.

3. ELASTIN: ACHIEVING STAGNATION-FREE INTERMITTENT COMPUTATION WITH BOUNDARY-FREE WITH BOUNDARY-FREE ADAPTIVE EXECUTION

Ensuring correct power failure recovery is important for intermittent computation in EHS devices, but the high runtime overhead prevents the wide use of software recovery solutions. In this chapter, we present Elastin, a compiler-directed software recovery solution that achieves a boundary-free intermittent computing system without causing the high overhead (PS1). Furthermore, Elastin achieves stagnation-free computation for energy-harvesting devices that ensures forward progress in the presence of frequent power outages; such a boundary-free nature allows Elastin to realize full potential of checkpoint adaptation. Thanks to the adaptive execution, Elastin outperforms the state-of-the-art by 3.5X on average (up to orders of magnitude speedup) and guarantees forward progress.

3.1 Introduction

Adoption of energy harvesting technologies in Internet of Things (IoT) has led to the advent of batteryless low-power embedded systems [84]–[87]. However, due to the unreliable power source, energy-harvesting systems suffer from unpredictable and frequent power failure. To address the power failure problem, prior software recovery schemes partition program into a series of recoverable regions (tasks) so that their re-executions always result in the same and correct output [4], [12], [13], [88]–[90]. Such a recoverability is achieved by either compiler-directed idempotent region formation [4], [10], [49]–[51], [91], [92] or user-based manual task partitioning [12], [13], [88]–[90]. In the wake of power failure, the prior schemes restart from the beginning of the interrupted region (task) after restoring the checkpoints saved at the region/task (task) boundary for correct recovery.

However, the region (task) based schemes [13], [88]–[90], [93], [94] face several critical issues. First, the schemes end up wasting the hard-won energy due to the lack of flexibility in the checkpoint interval; they make a checkpoint, that entails multiple energy-consuming NVM writes, at every pre-defined region (task) boundary, which would be unnecessary under stable energy-

harvesting condition. The crux of the problem is that due to the compile-time fixed regions (tasks), checkpoint interval cannot be adapted to the underlying energy harvesting quality and the power outage behavior.

Unfortunately, the inability to adapt checkpoint interval can cause a more serious issue, i.e., making the system stagnate while consuming the hard-won energy; that is why the prior schemes [43], [89], [90], [95]–[97] cannot ensure forward progress. If power outages repeatedly occur within a certain region (task) before it ends, the schemes continually attempt to re-execute the same interrupted region (task). This work refers to such a livelock-like situation as **stagnation**. Due to the small capacitance of an energy buffer, stagnation often occurs during the execution of long regions (tasks). Without solving the stagnation problem, all other efforts to make energy-harvesting systems reality would eventually fail, calling for a practical solution.

To address above issues, we present Elastin, a stagnation-free intermittent computing system for energy-harvesting devices that ensures forward progress in the presence of frequent power outages. Without requiring statically-defined region boundaries, Elastin leverages both timer-based checkpointing of volatile registers, that checkpoints registers when a time expired, and copy-on-write mappings of nonvolatile memory pages for correct recovery purpose. During each checkpoint interval, Elastin tracks memory writes on a per-page basis and backs up the original page—i.e., the copy-on-write granularity, not a virtual memory page—using software-controlled memory protection without MMU or TLB; Elastin can be regarded as library OS that only offers memory protection and timer interrupt. When a new interval starts at each timer expiration, Elastin clears the write permission of all the pages written in the previous interval and checkpoints all registers including a program counter as a recovery point. Elastin reconfigures the checkpoint interval and the page size based not only on the underlying energy harvesting quality but also on the observed forward progress. Consequently, Elastin achieves stagnation-free intermittent computation, ensuring forward progress across power outages.

The contributions of Elastin are as following:

- Elastin strongly guarantees forward execution progress. Experimental results show that Elastin is able to complete all benchmark applications, whereas the state-of-the-art work cannot due to stagnation.

- Elastin's boundary-free checkpointing requires neither user intervention nor program partitioning for region (task) formation while its 2-dimensional adaptation of timer interval and page size can maximize the forward progress.
- Elastin achieves 3.5X average speedup over the state-of-the-art region based scheme.

3.2 Background and Challenges

This section briefly discusses the challenges in software recovery solutions.

3.2.1 Curse of Stagnation

Suppose a program region/task whose execution time is greater than the power failure period, i.e., the time between the failures. If they periodically occur with the same frequency, the program ends up rolling back to the beginning of the same region again and again. That is because the failures keep occurring before the end of the region is reached, in which case the program just wastes harvested energy in vain making no forward progress.

3.2.2 Lack of Checkpoint Adaptation

Adaptive execution can achieve energy efficient intermittent computation by taking into account the underlying energy harvesting condition. For example, if the amount of harvested energy is sufficient, the energy-harvesting system does not have to frequently checkpoint to back up necessary program status due to low likelihood of power failure. On the other hand, if the harvesting energy source is weak or unstable, the system would need to checkpoint more frequently than usual. Unfortunately, all prior software schemes partition program to regions or re-structuring it as tasks to form recoverable regions/tasks without considering the level of harvested energy. Since the schemes checkpoint program status at each region/task boundary fixed at compile time, they cannot adapt to the varying quality of harvested energy at run time. Even if power failure rarely occurs, the schemes can waste hard-won energy by performing an unnecessary checkpoint at every single boundary during the execution of consecutive regions (tasks). Even worse, the schemes

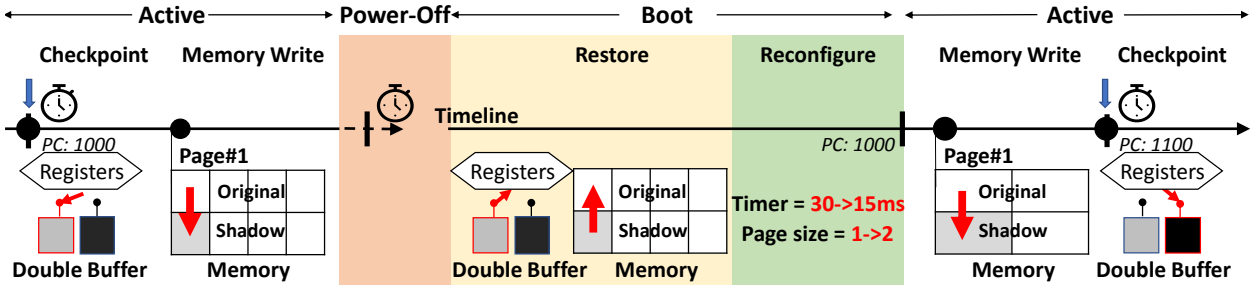


Figure 3.1. Overall workflow: checkpoint interval can be adjusted when it ends at timer expiration and in the wake of power outage at boot time

can suffer from stagnation during the execution of a long region (task) when power outages occur frequently.

3.3 Design

To realize full potential of adaptive execution according to energy-harvesting condition, we design Elastin's backup and recovery mechanisms in a boundary-free way without inserting region or task boundaries to program. For this purpose, Elastin leverages both timer-based checkpointing of volatile registers (Section 3.3.1) and copy-on-write mappings of nonvolatile memory (NVM) pages (Section 3.3.2) to restore them in the wake of power failure. Figure 3.1 describes the overall workflow of Elastin.

3.3.1 Watchdog Timer Based Checkpointing of Volatile Registers

Elastin leverages a watchdog timer, that can be adjusted at both its expiration and boot time (Section 3.3.3), to form flexible checkpoint interval. At each timer expiration where the current checkpoint interval finishes and the new one is about to start, Elastin checkpoints all registers including the program counter (PC) to a reserved area in NVM. In case of power outage during the register checkpoint, Elastin leverages *double buffering* to leave at least one of the two buffers intact [4], [98]; see Figure 3.1.

Note that Elastin saves the registers for the new interval in case it is interrupted due to power failure. As described in Figure 3.1, when power comes back, Elastin uses the PC as a recovery point to restart the interrupted interval after restoring all the other registers; they serve as inputs

to the interval to be restarted. As will be shown in Section 3.3.2, in addition to a volatile register file, Elastin needs to make a copy of NVM pages, which is invalidated at both timer expiration and boot time, for correct recovery. Thus, the recovery process includes the page restoration as well.

3.3.2 Page Protection Based Backup of Nonvolatile Memory

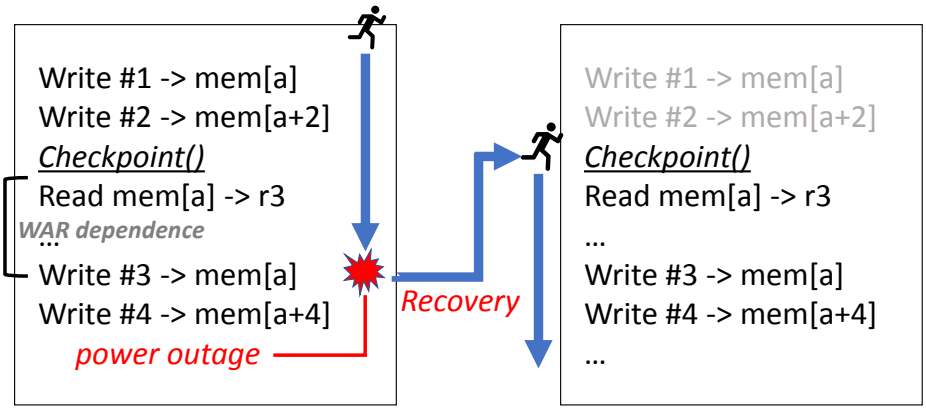


Figure 3.2. Memory inconsistent recovery due to anti-dependence

The timer-based checkpointing alone can lead to a memory inconsistency problem. Consider an example shown in Figure 3.2. Here, an energy-harvesting system checkpoints between write#2 and the following read instruction and encounters a power failure right after write#3. In this case, the write#3 and the Read instruction access to the same memory, mem[a]. Thus, these two instructions are anti-dependent, i.e., they form a WAR (write-after-read) dependence. In the wake of the power failure, the system starts from the most recently checkpointed point, thus subse-

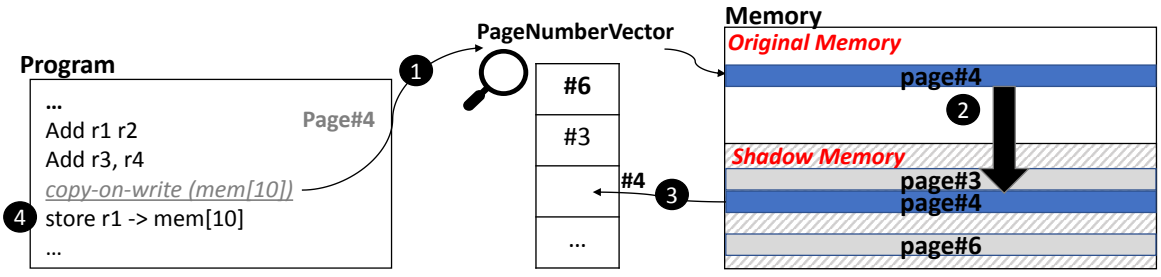


Figure 3.3. copy-on-write backup: ❶ page number vector (PNV) lookup, ❷ copy the page to shadow, ❸ PNV insertion, ❹ memory write

quently reading mem[a]. However, it ends up reading not the original value but the one updated by write#3, thereby leading to incorrect recovery.

To address the memory inconsistency, during each checkpoint interval, Elastin tracks memory writes on a per-page basis and backs up the original page; in the wake of power failure, Elastin first reverts all the writes (including anti-dependent ones) performed in the interrupted interval using the backup page and then jumps back to the recovery PC (Section 3.3.1) where the interval started. That way Elastin can restart the interrupted interval with original memory status as if it were being started for the first time.

To achieve this, Elastin leverages a conventional page protection mechanism of operating systems which tracks writes to non-writable page as a page fault and backs up the page with a copy-on-write mechanism [99]. In general, energy-harvesting systems do not run OS due to the scarce power supply, and thus we implemented custom page protection library; in a sense, Elastin can be regarded as a library OS that only supports page protection[1] and timer interrupt handling.

Interaction with Timer Based Checkpointing: When the watchdog timer is expired (i.e., the current checkpoint interval has just been finished), Elastin clears the write permission of all the pages written in the interval. In other words, when the upcoming new interval starts, no page has a write permission. This gets the new interval ready to track its own writes and trigger copy-on-write for backing up the corresponding pages. In this way, Elastin can ensure that each interval starts with clean memory status.

Custom Software-Controlled Page Protection: To track memory writes and trigger their copy-on-write if needed, Elastin instruments store instructions at compile time while maintaining a page number vector (PNV) to record which page has a write permission at run time. For each store, Elastin first checks if the target page has a write permission by consulting PNV (❶ in Figure 3.3). If not, i.e., the page number is not found in PNV, Elastin creates a copy of the page (i.e., copy-on-write) in shadow memory or radix tree based data structure (❷); Section 3.4.2 discusses the overhead of these alternatives. Then, to grant the page a write permission, Elastin inserts the page number (#4 in Figure 3.3) to PNV as a mark for the permission (❸). In this way, Elastin can preserve the copy of the page until the end of the current checkpoint interval so that the copy can

[1]↑ It is only for page backup and does not support virtual memory. The MCU of energy harvesting systems lacks MMU/TLB due to power constraint.

be used to recover from possible outages during the interval. Finally, Elastin performs the write (❹).

On the other hand, if the page being stored has a write permission [2], i.e., the page number is found in PNV, Elastin skips both the page copy and the PNV insertion. In summary, for a store to writable pages, Elastin takes only two steps (❶→❹) while a store to non-writable pages goes through all four steps (❶→❷→❸→❹). Note that any power outage between these steps does not cause a memory inconsistency problem during the recovery as long as their order is enforced.

In the wake of power failure, Elastin reverts all the written pages (i.e., those populated in PNV) by using their original copy in shadow memory along with restoring all registers as shown in Figure 3.1. Obviously, this software-controlled page protection mechanism consumes the harvested energy for both page backup and restoration; Section 3.3.3 describes how Elastin adjusts the page size to minimize the copy-on-write overhead, and Section 3.3.5 shows how Elastin bounds the energy consumption to ensure forward progress.

Discussion: PNV is small enough to keep the lookup cost low. In energy-harvesting systems, the common case is that they encounter frequent power failures, e.g., in a few tens of milliseconds. During the short power-on period (one charge cycle run time), PNV is populated with only a handful number of pages in reality. Another reason for the small size of PNV is spatial locality; many stores fall into a few previously-populated pages during the short period of intermittent execution [5].

In particular, the size of PNV never grows unboundedly. To avoid stagnation not only at run time but also at boot time, Elastin bounds the number of pages, that can be populated at run time, by taking into account their restoration cost at boot (i.e., recovery) time. Section 3.3.4 shows how Elastin bounds the number of the populatable pages.

3.3.3 Adaptive Execution

To enable energy efficient intermittent computation, Elastin dynamically adjusts the checkpoint interval and the page size at both timer expiration time and boot time if needed.

[2]↑In other words, the page has already been accessed before in the current checkpoint interval.

Checkpoint Interval Adaptation: Elastin reconfigures the checkpoint interval by taking into account the condition (quality) of the energy harvesting source. This harvesting condition is an important factor for Elastin to determine whether the checkpoint interval should be adjusted or not. If the quality of the harvested energy is sufficiently good, there is no need to frequently checkpoint at run time; not doing so can make a better forward progress by saving the high energy of NVM writes required for the register checkpoint and the page backup. In contrast, if the harvested energy is not enough, a system should checkpoint before the impending power outage. In light of this, Elastin leverages the timer itself to figure out the underlying energy harvesting condition.

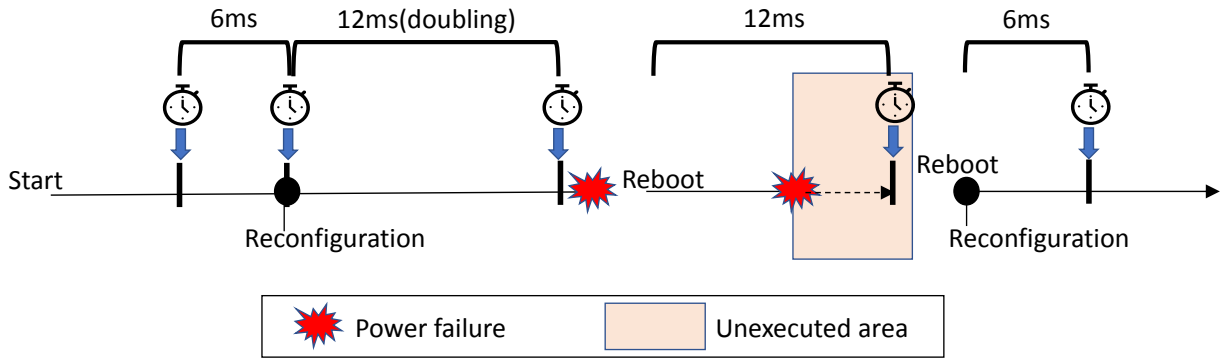


Figure 3.4. Timer reconfiguration example

Figure 3.4 describes how Elastin reconfigures the checkpoint interval. If the timer expires two times in a row [3] while the energy-harvesting system is active, Elastin assumes that the system have gone through good energy harvesting condition. Thus, it doubles the checkpoint interval at the second timer expiration. The rationale behind this heuristic is that at the second timer expiration, at least the first checkpoint interval turns out to be unnecessary because it did not encounter a power outage; the harvested energy was that sufficient. However, the second interval should not be considered as unnecessary because the next interval may encounter a power outage.

On the other hand, if the timer has never expired since the last reboot, i.e., *checkpoint counter* is 0, then Elastin assumes that the system is under poor energy-harvesting condition. The intuition here is that the harvested energy was insufficient to pass even the first checkpoint interval without interruption due to power failure. With that in mind, Elastin sets the interval as a half of the last

[3] To detect this, we use a metadata variable called *checkpoint counter*.

timer value in the wake of the power failure, i.e., at the reboot time. This particular approach (i.e., timer-halving mechanism) helps the system to overcome the stagnation problem for most of the time, though there are a few exceptional cases; Section 3.3.4 shows how Elastin handles them for stagnation-free intermittent computation.

Page Size Adaptation: To reduce the *copy-on-write* overhead, Elastin attempts to find the optimal page size; the spatial locality of memory writes is likely to vary due to program phase behavior [100], and therefore the best page size might vary for each phase.

In our current design, the memory page size cannot be changed at run time, which would otherwise cause significant metadata (e.g., PNV) updates overheads and a subtle correctness issue due to power failure between them. Instead, Elastin reconfigures the page size at reboot time as shown in Figure 3.1 to make the adaptation easier and still find the best size across power outages.

In the wake of a power outage, Elastin first restores all the pages populated in PNV. Then, it measures the cost of the current page configuration by the product of the page size and the number of populated pages, i.e., the size of PNV. Finally, Elastin resets the page size to the best-performing one using the decision logic of adaptive execution [101], [102].

That is, as decision runs, Elastin tries a set of page sizes to select the best among them across power outages; Section 3.4.5 shows how the set is determined. Even though the best page size is selected at the end of decision runs, it is not fixed for the upcoming reboot times. Instead, at every reboot time, Elastin measures the cost of its current pick, which is compared to the most recent costs of the other page sizes, to see if it is still the best or one of them becomes the new best.

3.3.4 Challenges in Forward Progress Guarantee

When the system repeatedly starts at the same recovery point due to stagnation, Elastin reduces the checkpoint interval by halving the watchdog timer value in the wake of each power outage (Section 3.3.3). In this simple way, Elastin can effectively avoid the stagnation problem.

However, there are a couple of challenges that must be addressed to ensure forward progress for stagnation-free intermittent computation; (1) an excessive pages populated during a checkpoint interval, and (2) capacitor (i.e., energy buffer) malfunction due to wear-out, environmental factors such as temperature change, physical access attacks, and so on. First, if there are too many

populated pages which must be restored at recovery time, the system may be stagnated in the middle of the recovery process. Second, if the capacitor is malfunctioning, the system may suffer from stagnation—e.g., the buffered energy is not enough to complete even single page backup or restoration.

To tackle these potential stagnation problems, Elastin defines thresholds for each condition: (1) the quota (i.e., maximum number) of populatable pages during a given checkpoint interval and (2) the lower bound of the power-on period (i.e., one charge cycle run time) of the energy harvesting system while its capacitor works fine. We assume that the lower bound as the worst case scenario to ensure forward progress even in the most harsh situation, i.e., the lower bound is called **WCPT** (the worst case power consuming time) [4].

3.3.5 Stagnation-free Adaptation Solution

In this section, we first delve into WCPT and then show how to use it for detecting the capacitor malfunction problem. Finally, we show how WCPT can be used as a basis for solving the other problem, i.e., how to bound the number of the pages populated in a checkpoint interval.

Worst Case Power Consuming Time: Specifically, we define WCPT as follows: *how long can an energy harvesting system sustain its execution under the maximum power consumption mode?* To figure this out for the target energy harvesting system, Elastin analyzes its capacitor, i.e., energy buffer [5]. This is motivated by the insight that energy harvesting systems do not boot until the capacitor (energy buffer) is fully charged as with commodity systems such as WISP [105]. In the wake of each power outage, it is thus assured that the program can make as much progress as the fully charged capacitor allows, even if no additional energy is harvested. Section 3.4.6 shows how Elastin calculates WCPT with this in mind and discusses how the calculation can be extended in case the system is equipped with other components such as sensors.

Energy Buffer Malfunction: Once WCPT is obtained, Elastin leverages it to detect the capacitor malfunction problem based on the following invariant: *the power-on period of an energy harvesting system should not be shorter than its WCPT*—as long as the capacitor works well. That is, if

[4] ↑Elastin can precisely bound WCPT due to the MSP430 MCU's simple architecture and execution environment, i.e., in-order core without cache/OS.

[5] ↑ A capacitor is used as an energy buffer [103], [104]. When an electric component depends upon a specific amount of power, the energy buffer is placed to provide the required power.

this invariant does not hold, the capacitor is malfunctioning. However, it is impossible for a timer to measure the power-on period because the timer value is reset on a power outage; Section 3.4.3 shows how Elastin checks the invariant without measuring the power-on period.

If the capacitor turns out to be malfunctioning based on the invariant checking, Elastin treats this situation as an exception and switches to its handling mode. At the reboot time, Elastin first decreases the page size to the minimum (2 bytes) and then sequentially restores the registers and pages one by one in case there is an insufficient amount of energy for their restoration in a the batch manner. With the exception handling mechanism, Elastin can avoid stagnation even if the capacitor malfunctions—provided the system can run at least a single read/write instruction without interruption [6].

Populatable Pages: Elastin also leverages WCPT to determine the maximum number of the populatable pages with their boot-time restoration cost in mind. To ensure that at recovery (boot) time, all the pages populated in the last checkpoint interval can be safely restored, the total page restoration time must be shorter than WCPT, i.e., $Number\_of\_Pages * Single\_Page\_Restore\_Time < WCPT$; otherwise, power failure may occur in the middle of the restoration process. For the threshold of $Number\_of\_Pages$, Elastin therefore uses the maximum value among those that satisfy the above inequality. In this way, when Elastin reconfigures the page size at boot time, the threshold is also updated according to the new page size. If the number of the pages populated in a checkpoint interval happens to exceed the threshold, which is detected by checking a metadata variable called *populated page counter*, Elastin makes an additional checkpoint right at the moment. This allows Elastin to safely restore all the pages at the next reboot time without interruption due to power failure.

3.4 Implementation

3.4.1 Register Checkpointing, Permission Clearing Protocol

As shown in Section 3.3.2, at each timer expiration, Elastin checkpoints all registers including PC with double buffering and invalidates out the write permission of all pages. For this purpose,

[6]↑If the capacitor cannot even secure energy required for one memory instruction, Elastin assumes that the system is completely unusable.

Elastin maintains two bits: (1) a *double buffer index bit* that is toggled at the end of the register file checkpointing and (2) a *PNV valid bit* whose reset invalidates the write permission of all pages. Note that these two bits must be atomically updated. Otherwise, a power outage between the two separate updates leads to incorrect recovery; in the wake of the power outage, Elastin ends up reverting the pages written in the formerly finished interval though it is about to start a new interval from the checkpointed PC, not the former. To avoid the incorrect recovery, Elastin updates the two bits in a single store instruction that guarantees failure atomicity [106]. Once they atomically updated, Elastin clears out all page numbers in PNV by using a single DMA operation [7] as will be shown in Section 3.4.4; the amount of the DMA write is determined by *populated page counter*. Once it is successfully done, Elastin finally sets the counter to zero before starting the new interval.

3.4.2 Memory Organization

Elastin divides the whole nonvolatile memory into four areas: main (original) memory, shadow memory, register double buffer, and reserved memory for PNV and the rest of various metadata, i.e., the checkpoint counter, valid bits for checkpoint and PNV, a performance table of page sizes, the populated page counter, thresholds for the number of populatable pages and WCPT, and so on.

The biggest problem with shadow memory is that it occupies a half of the total memory size, thus failing to run those applications that have high memory footprints. To overcome this challenge, Elastin proposes another design choice, radix tree memory management; as OS implements the page table using a radix tree, we used the same kind of data structure. By using radix tree as backup page storage, Elastin can increase available main memory size for applications at the expense of the increased page search overhead. Section 3.5 evaluates the performance overhead of both shadow memory and radix tree.

3.4.3 Invariant Checking for Capacitor Malfunction Detection

To detect capacitor malfunction, Elastin uses the invariant of an intact capacitor, i.e., the power-on period (one charge cycle run time) should not be shorter than WCPT; see Section 3.3.5. How-

[7] ↑Even if the DMA operation fails due to power failure, Elastin does not lead to incorrect recovery. In the wake of the power failure, Elastin simply starts the DMA operation over and follows the rest of the protocol.

ever, Elastin cannot use a timer to measure the period because the timer value will be reset on power outage. To achieve the invariant checking without a timer, Elastin relies on the following observation: when the first checkpoint is not made—due to power outage—since the last reboot, we can infer that the power-on period must be less than the checkpoint interval; this is the reason Elastin to halve the interval (Section 3.3.3). With this in mind, Elastin detects the capacitor malfunction as follows.

While the capacitor malfunctions, Elastin keeps decreasing the checkpoint interval due to frequent power outages. Thus, after many outages and resumptions, the interval would eventually become WCPT at some recovery time. At the moment, if it turns out that no checkpoint was performed since the last boot, i.e., *checkpoint counter* is 0, then we know that the power-on period is definitely shorter than the interval (WCPT). Thus, we conclude that capacitor is malfunctioning. In short, when a checkpoint interval is the minimum (WCPT), if the checkpoint is not made before power failure, Elastin switches to the exception handling mode (Section 3.3.5).

3.4.4 DMA-Based Fast Page Copy

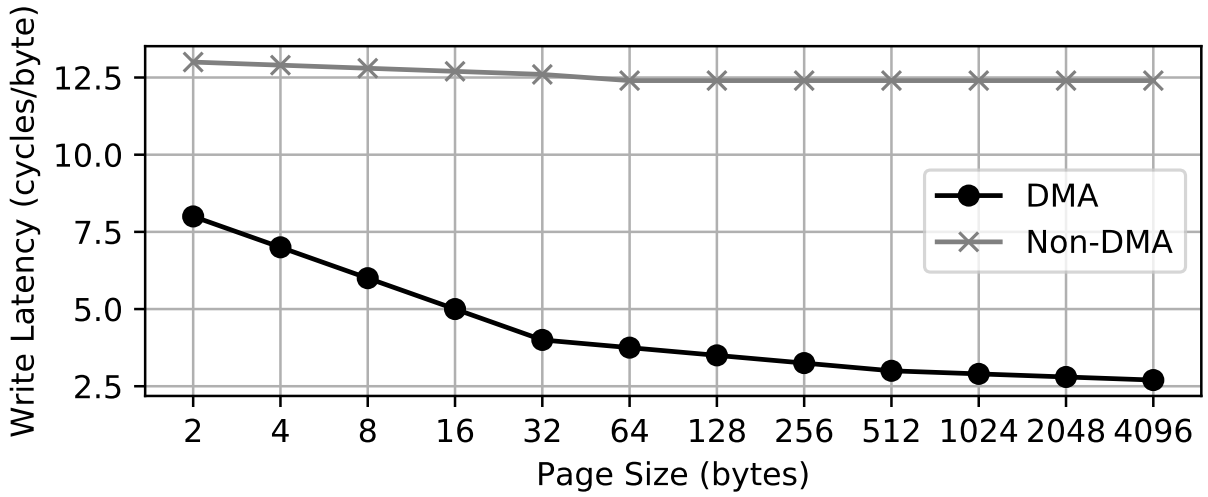


Figure 3.5. NVM write latency with DMA (Cycles per byte)

Elastin's copy-on-write mechanism entails NVM writes for the page copy. However, due to the nature of NVM, memory writes incur very significant latency [107]–[116]. To reduce the overhead,

Elastin leverages direct memory access (DMA) hardware accelerator available in the target energy harvesting system. Figure 3.5 demonstrates that a single byte DMA transfer takes only 8 cycles which is about 1.5X faster than the standard memory copy without DMA. When the copy size is larger than a single byte [117]. DMA copy becomes 4∼5X faster.

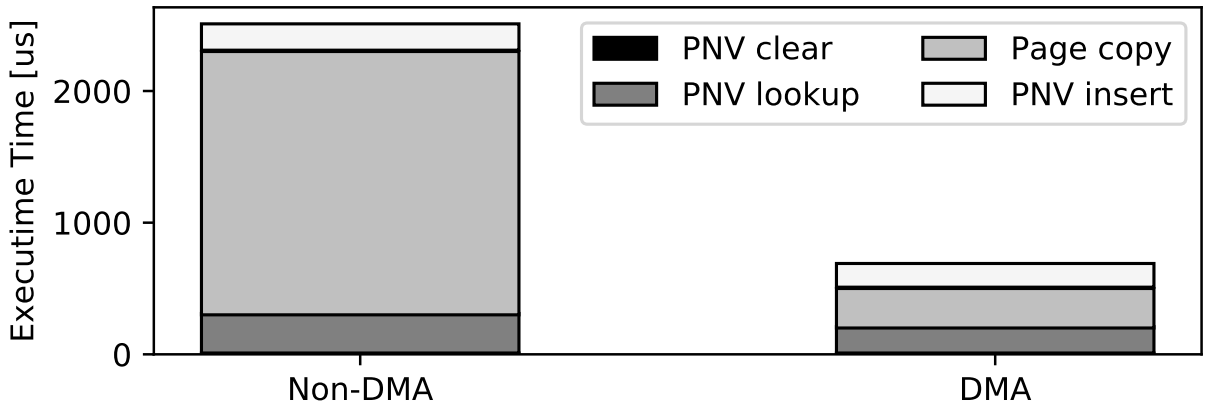


Figure 3.6. Copy-on-write overhead breakdown in stable power input case. With the page size of 256 bytes, the major overhead comes from the page copy.

We also measured the impact of DMA on Elastin's per-page based copy-on-write mechanism for all of our benchmarks. Figure 3.6 shows the average execution time breakdowns of the copy-on-write for a 256B page with and without DMA. The page copy overhead occupies the most significant portion, and the PNV lookup overhead follows. With DMA, the overall execution time becomes about one third of the original time without DMA; this results from the large reduction of the page copy overhead which is about 6X speedup. In particular, the PNV clearing overhead is negligible because all page numbers are cleared by one DMA operation. In contrast, PNV insertions cannot be batched, since they are far apart from each other. Even though DMA can be leveraged for them, the DMA initialization and completion costs offset the benefit. Consequently, Elastin takes advantage of the DMA only for the page copying and the page clearing.

3.4.5 Page Size Adaptation Range

Elastin's page size selection is based on a series of decision runs for testing each size; see Section 3.3.2. Since most of them are suboptimal, Elastin tries to minimize the decision runs by limiting the range of page sizes to be tested.

To find the optimal page size, it is necessary to understand the tradeoff between the cost of PNV copying and the cost of page clearing. For example, if the page size is too small, it may incur frequent page copies causing expensive PNV clearing cost at reboot time or timer expiration. In contrast, if the page size is too large, the system may consume too much energy for copying even on page. In addition, the spatial locality of memory writes is another important factor. The high locality lets Elastin skip the page copy and PNV insertion since the memory writes are likely to be concentrated on a few pages. while the low locality increases them since many writes tend to touch many different pages.

The tradeoff is affected by the locality, e.g., with the high locality can amortize the cost of a large page copy by many subsequent writes whose address falls into the same page. To a large extent, the locality significantly varies across applications due to their different pattern of memory writes. With that in mind, we empirically measured the performance of each page size for all of our benchmarks. Figure 3.7 shows the average execution time overhead of the best 4 page size configurations, i.e., 32B, 64B, 128B, 256B. As a result, Elastin's adaptive execution uses them for decision runs, i.e., the page size adaptation range is 32$\sim$256 bytes.

3.4.6 Worst Case Power Consuming Time

In this work, WCPT is defined as: how long a program can sustain its execution under the maximum power consumption mode of the microcontroller (MCU) which drains the energy from the capacitor at the highest rate. To measure WCPT, Elastin needs to know the energy buffer size (capacitance), because the MCU may rely on only the buffer without any input from harvesting energy sources in the worst case. For a given capacitance of the energy buffer (e.g., 47μF in

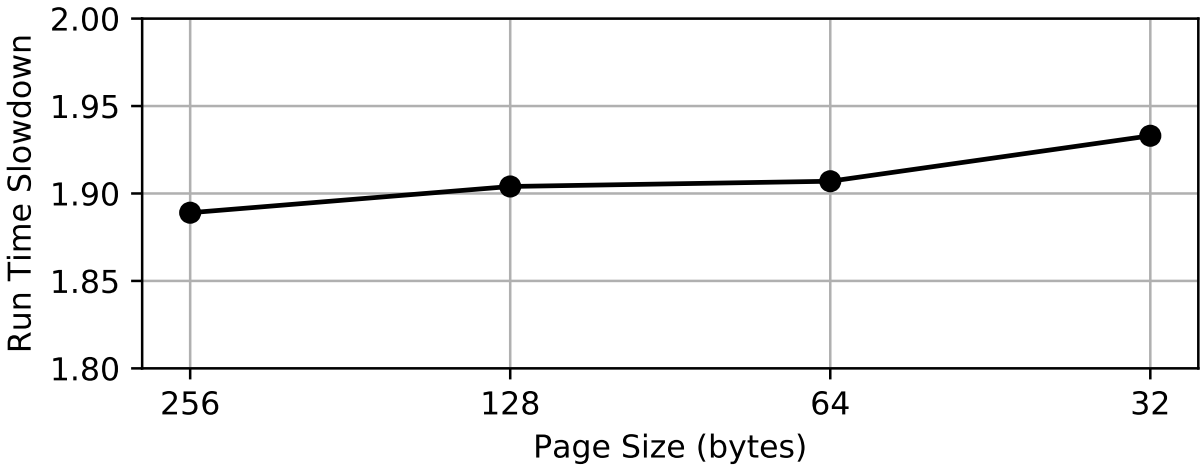


Figure 3.7. Average execution time overhead of the best 4 page size configurations for all benchmarks when DMA and shadow memory are used with stable power input, i.e., no power failure.

WISP5), it provides the MCU with the operating voltage from its starting point (V_{max}) to the power outage point (V_{min}). Then, Elastin estimates the available energy input as follows:

$$Available\ Energy\ Input = \frac{1}{2} C_{buf} * (V_{max}^2 - V_{min}^2). \tag{3.1}$$

For the maximum power consumption estimation of MCU, Elastin leveraged the following equation [118] :

$$E_{tot} = P_{tot} t = V_{dd} I_{leak} t + C_{msp} V_{dd}^2 \tag{3.2}$$

where V_{dd}, I_{leak}, and C_{msp} are input voltage to MCU, leakage current, and the MCU capacitance, respectively. Elastin considers the input voltage to MCU by taking into account the capacitor discharge behavior, since the capacitor cannot consistently provide the same amount of power. Elastin models the input voltage variation while the energy buffer is discharged with a simple equation: $v_o(t) = V_o e^{\frac{-t}{CR}}$ for which the capacitance (C) is already given by *Equation* 3.1, and the resistance (R) can be calculated by Ohm's law, $R = V/I$. Elastin views the MCU as a huge constant resistor, R, under the maximum power consumption mode. That is, Elastin refers to the MCU manual to figure out the maximum current (I)—that the device can consume—and use it to calculate the resistance (R). For I_{leak} and C_{msp}, Elastin refers to the manual as well; If a certain

MCU's manual does not specify them in any case, Elastin can adapt the typical leakage current and capacitance model as one used in [118]. With all these findings, the available energy input obtained by *Equation* 3.1 should be always greater than the energy consumption of the underlying MCU given by *Equation* 3.2. With that in mind, Elastin calculates the WCPT by calculating a threshold t in the following equation:

$$\frac{1}{2}C_{buf} * (V_{max}^2 - V_{min}^2) > V_o e^{\frac{-t}{CR}}(I_{leak})t + C_{msp}(V_o e^{\frac{-t}{CR}})^2 \tag{3.3}$$

In particular, this is applicable to commodity energy harvesting devices. For instance, WISP5 consists of 47μF energy buffer and MSP430FR5969. This MCU consumes 2650μA at 3.0V, 16MHz, in an active mode [119]. The MCU starts to operate at 2.4V and performs down to 1.8V while the resistance value of the MCU is 1133Ω. Therefore, the resulting WCPT is approximately 11.6ms.

Discussion: Thanks to the simplicity of the above analytical model, it is easy for Elastin to incorporate other system components in the WCPT calculation. For example, if the system is equipped with other components, e.g., sensors and actuators. For this purpose, Elastin needs to update the resistance part of *Equation* 3.3, i.e., $R = \frac{V}{I_{MCU}+I_{Sensor}+I_{Actuator}}$. To figure out the maximum current of the components, Elastin simply refers to their manuals as usual.

3.5 Evaluation

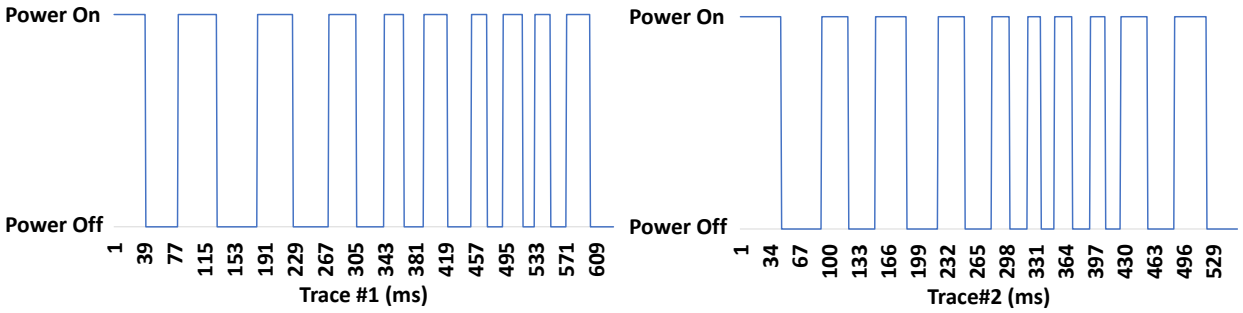
We conducted all the experiments on TI's MSP430FR5994 Launchpad development kit board[8] and implemented Elastin described in Section 3.3 as a runtime library. To instrument nonvolatile memory (NVM) writes (Figure 3.3), we implemented a source-to-source translator using the LLVM compiler infrastructure [120]. Then, the instrumented program and the runtime library are compiled and linked using TI's MSP430 GCC toolchain to generate the binary executable.

To compare Elastin with Ratchet [4], the state-of-the-art region based work, we ported it to MSP430 since it was originally implemented for ARM [121]. Note that we omitted Ratchet's timer based checkpointing, because it does not work—i.e., it may cause incorrect recovery—for those idempotent regions that contain WARAW (Write-After-Read-After-Write) dependence as admitted

[8] ↑FRAM is used as main memory, and we do not use SRAM at all.

by the author [4]. We evaluated both Elastin and Ratchet for total 11 benchmarks comprised of a subset of MiBench applications [122], [123] and others from prior works [90], [94]. All the benchmark applications were compiled with standard -O3 optimization.

3.5.1 Intermittent Computing Platform



(a) Power trace 1 (b) Power trace 2

Figure 3.8. Realistic intermittent power traces (simplified)

We developed a special power generator board with TI's MSP430FR5969 to mimic various power outages and resumptions as with prior work [18]. The power generator board provides supply voltage between 0 to 3.3V, directly to the target evaluation board (i.e., TI's MSP430FR5994) through GPIO pins to power it on/off at will based on power input traces. Unlike prior works [4], [5], [12], [89], [90] that do not vary the power failure frequency, we randomly increase and decrease the power-on period to model various energy sources and environments, which serves to stress-test Elastin. The minimum bound of the power-on period is set to 15ms [9] while the minimum bound to a half of the execution time of the smallest application among our benchmarks. With that in mind, we synthesized two power traces for our intermittent computing experiments as shown in Figure 3.8. At power-on, the power generator board provides voltage to the target board while it cuts the voltage at power-off for outage.

[9] ↑WISP5 [105], a commodity energy-harvesting system, has an a capacitor of 47μF, and it can sustain about 15ms as one charge cycle run time [90].

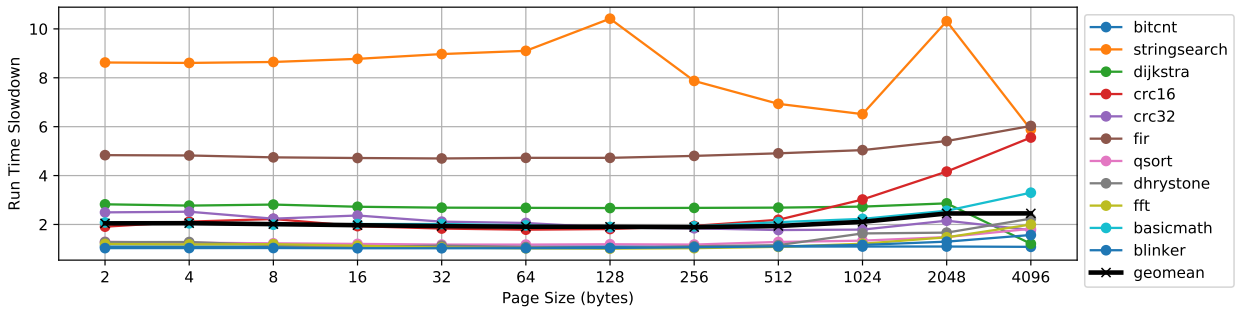


Figure 3.9. Normalized performance overhead of Elastin (shadow memory design)

3.5.2 Execution Time Overhead Analysis with No Power Failure

We first analyze Elastin's execution time overhead when the power source is stable, i.e., there is no power failure. Here, we set the baseline to the uninstrumented binaries that have no checkpoint/restart support. We measured the overhead of Elastin for 11 benchmark applications varying the page size from 2B to 4KB and alternating the backup page storage between a shadow memory and a 2-level radix tree data structure. Figure 3.9 shows the normalized overhead of Elastin compared to the baseline when shadow memory is used while 3.10 shows that when the 2-level Radix tree is used. Overall, the average overhead of Elastin is 88% with the best page size of 256 bytes in shadow memory (see Figure 3.7) while 2-level Radix tree results in 243% with the same page size as the best. Nevertheless, it would be a mistake to take this to mean that Elastin incurs such a significant overhead for intermittent computation. Recall that frequent power failures are the norm in energy-harvesting systems, and this particular experiment has no power failure at all; Section 3.5.3 evaluates the performance impact of Elastin on intermittent computation with frequent power outages.

As shown in the figures, some applications such as bitcnt, dijkstra, and stringsearch prefer larger page size. Even if the copy-on-write of a large page size is expensive due to high volume of NVM copy, the cost is amortized by high spatial locality in the following memory writes. In contrast, the other applications show performance degradation when the page size is bigger than 256 bytes. That is because the cost of such a large page copy cannot be paid off in the applications due to their low spatial locality. The takeaway is that the best page size varies depending on application characteristics. Furthermore, the best page might vary even during program execution

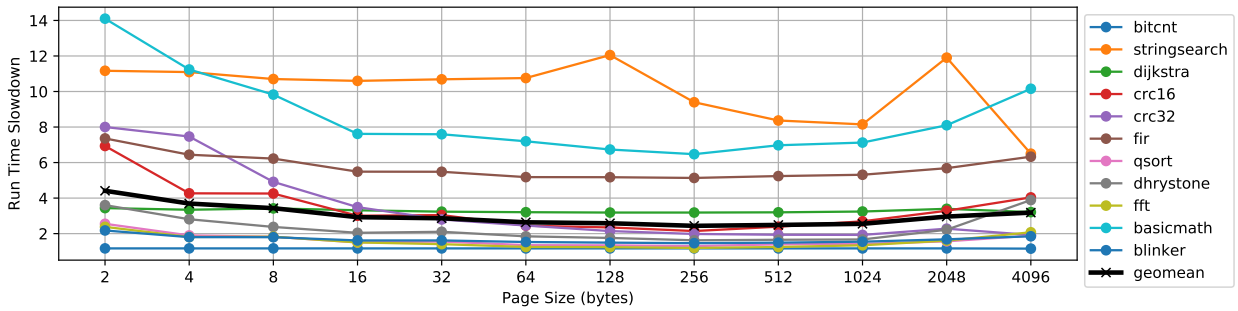


Figure 3.10. Normalized performance overhead of Elastin (2-level radix tree design)

due to phase behavior [100]. In such a case, Elastin's boundary-free adaptive execution can find the right page size across power failures.

3.5.3 Execution Time Overhead Analysis with Power Outages

To evaluate the forward execution progress in the presence of power failures, we measured the application's completion time i.e., the execution time taken to complete the application across power failures. In Figure 3.11 and Figure 3.12, total three cases are compared using the two power traces shown in Figure 3.8: the state-of-the-art [4], i.e., Ratchet in the legend, Elastin with timer only adaptation, i.e., Elastin (timer), and Elastin with both timer and page size adaptation, i.e., Elastin (timer+page). Note that in Figure 3.11 (trace#1) and Figure 3.12 (trace#2), we set the baseline to our approach, i.e., Elastin (timer+page) because Ratchet makes many applications stagnate.

As shown in the figures, Ratchet [4] incurs stagnation problem in five applications on both traces. For the rest applications where Ratchet does not stagnate, Elastin (timer+page) outperforms Ratchet on average by 3.5X and 3X for trace#1 and trace#2, respectively. Also, it turns out that Elastin (timer+page) improves Elastin (timer) on average by 40% and 8% for trace#1 and trace#2, respectively. This confirms that Elastin's boundary-free 2-dimensional (timer and page size) adaptation works effectively.

Interestingly, Figure 3.11 shows that Ratchet could outperform Elastin for basicmath. That is because in basicmath, Ratchet happens to have the optimal size of regions which corresponds to the input power cycle characteristics, i.e., trace#1. Thus, when a different power trace is used, Ratchet

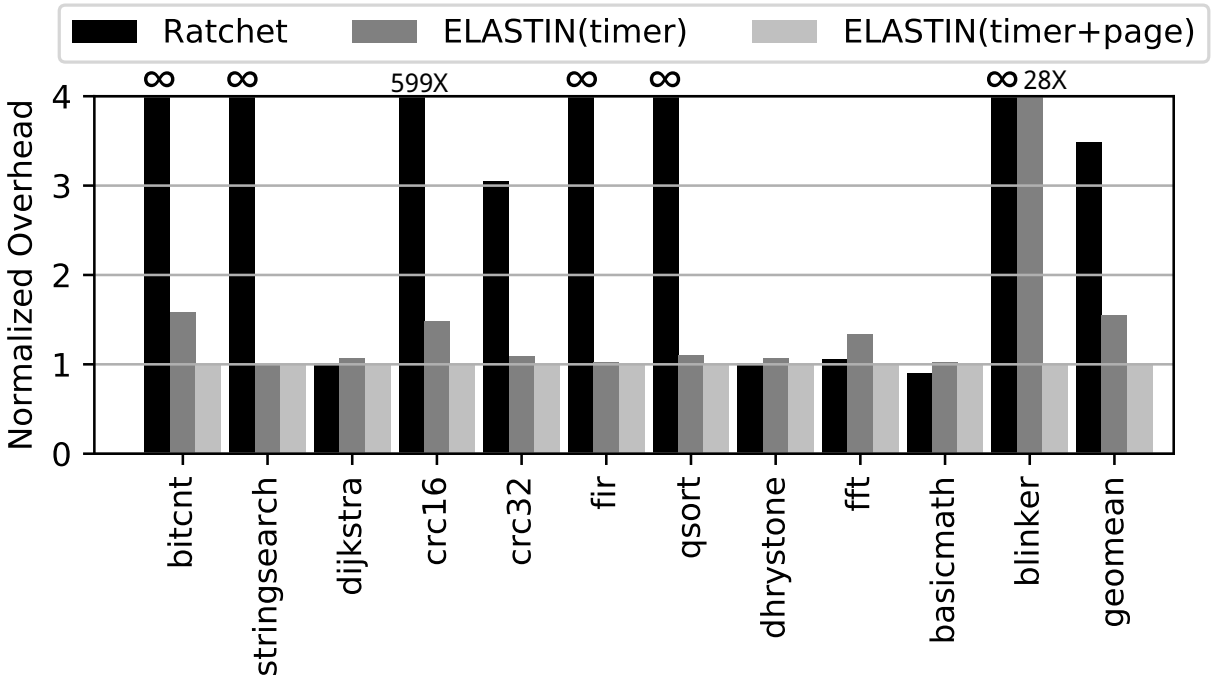


Figure 3.11. Application completion time in the presence of power failures using trace#1: the bar of stagnated applications reaches ∞, and the geomean of Ratchet is calculated only for non-stagnated applications.

cannot beat Elastin, which is confirmed by our experiment with trace#2. As shown in Figure 3.12, for the same application (basicmath), Elastin significantly outperforms Ratchet under trace#2.

3.5.4 Energy Consumption Breakdown across Power Outages

We also analyzed the energy consumption breakdown across power outages. Figure 3.13 shows that *copy-on-write* and the register checkpoint (ckpt in the legend) do not consume significant amount of energy, i.e., less than 1% on average. That is because the average number of copied pages in intermittent computation is only about 1 or 2 thanks to spatial locality and high power outage frequency; as shown in Figure 3.6, the PNV lookup overhead is trivial as well, and thus overall copy-on-write overhead is not significant.

Overall, the overhead of Elastin comes from the re-execution cost; in the legend, 'forward' means the energy consumption for a portion of execution time that has never been restarted, thus it is not an overhead technically. Even if Elastin reconfigures the checkpoint interval by halving

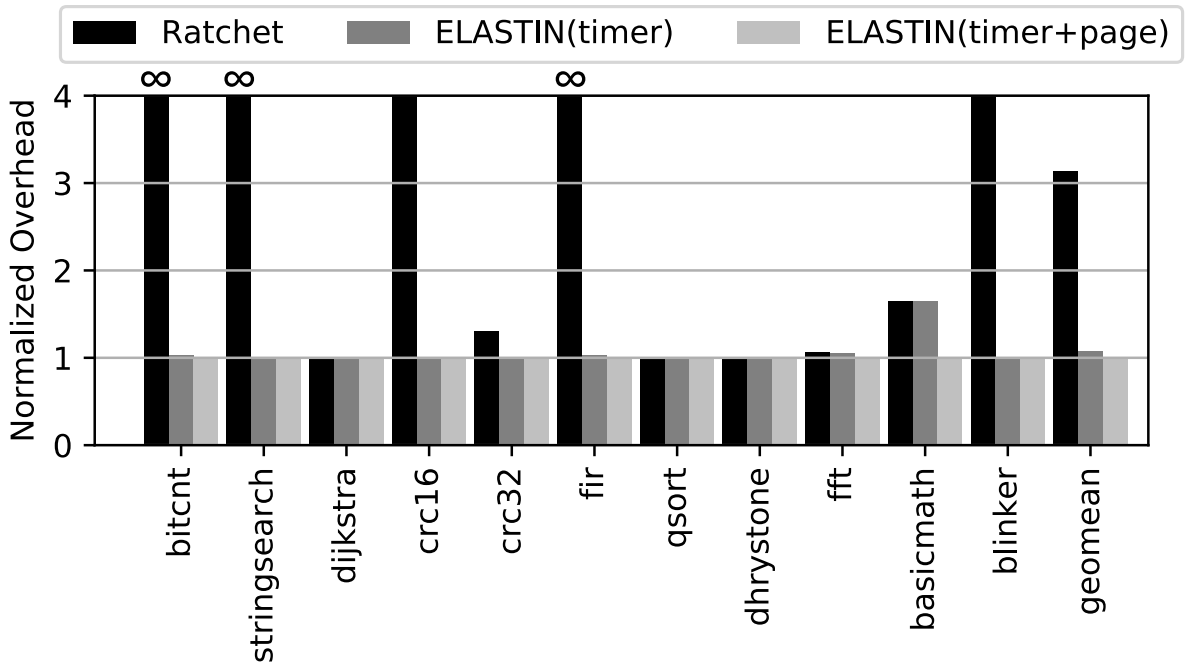


Figure 3.12. Application completion time in the presence of power failures using trace#2: the bar of stagnated applications reaches ∞, and the geomean of Ratchet is calculated only for non-stagnated applications.

the previously selected interval when the system dies without forward progress, the re-defined checkpoint interval may not help for the first time to make progress. For example, to get out of stagnation, Elastin might need to perform multiple times of checkpoint interval halving across power failures, and the re-execution of the reduced intervals consumes the harvested energy. As shown in Figure 3.13, this re-execution overhead is about 39% on average.

3.5.5 Exception Handling for Capacitor Malfunction

Capacitor (i.e., energy buffer) can malfunction either by natural worn-out or physical security attacks [124]. For example, when cracked, the capacitor leaks the buffered energy more quickly and the original capacitance is significantly reduced [19]. To have a scenario of the malfunctioning capacitor, we set the power-on period to 5X lower than the normal minimum bound mimicking the cracked energy buffer. That is, the system only runs for 3ms intermittently. As shown in Figure 3.14, Ratchet was unable to complete all of benchmark applications. In contrast, Elastin

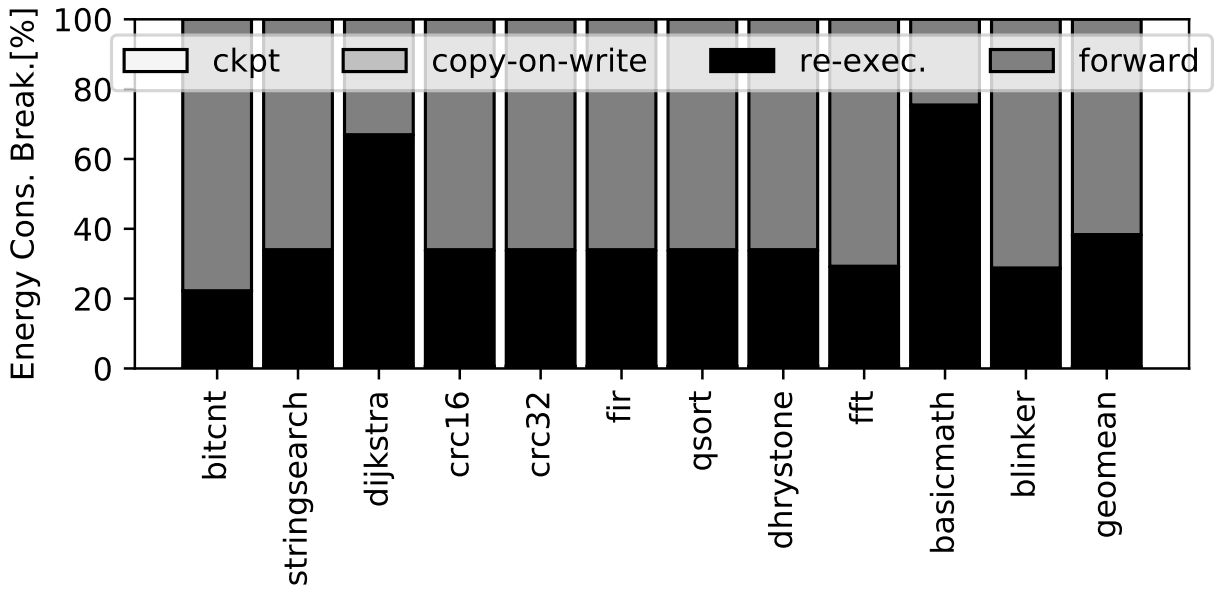


Figure 3.13. Energy consumption breakdown of Elastin

successfully completes them all. This implies that Elastin is not only robust against capacitor malfunction but also capable of working with smaller capacitance, e.g., less than 10μF. Consequently, we believe that Elastin enables using a smaller capacitor, which should be a desired approach for smaller chips required for IoT industry such as wearable markets.

3.6 Summary

We present Elastin, a stagnation-free intermittent computing system that ensures forward progress in the presence of frequent power outages. Elastin leverages both timer-based checkpointing of volatile registers and copy-on-write mappings of nonvolatile memory pages to restore them in the wake of power failure. Unlike prior works, Elastin does not partition program into recoverable regions or tasks. The boundary-free nature allows Elastin to realize full potential of adaptive execution, adjusting both the checkpoint interval and the page size at will. Consequently, Elastin can achieve stagnation-free intermittent computation and maximize forward progress across power outages.

During each checkpoint interval, Elastin tracks memory writes on a per-page basis and backs up the original page—i.e., the copy-on-write granularity, not a virtual memory page—using software-

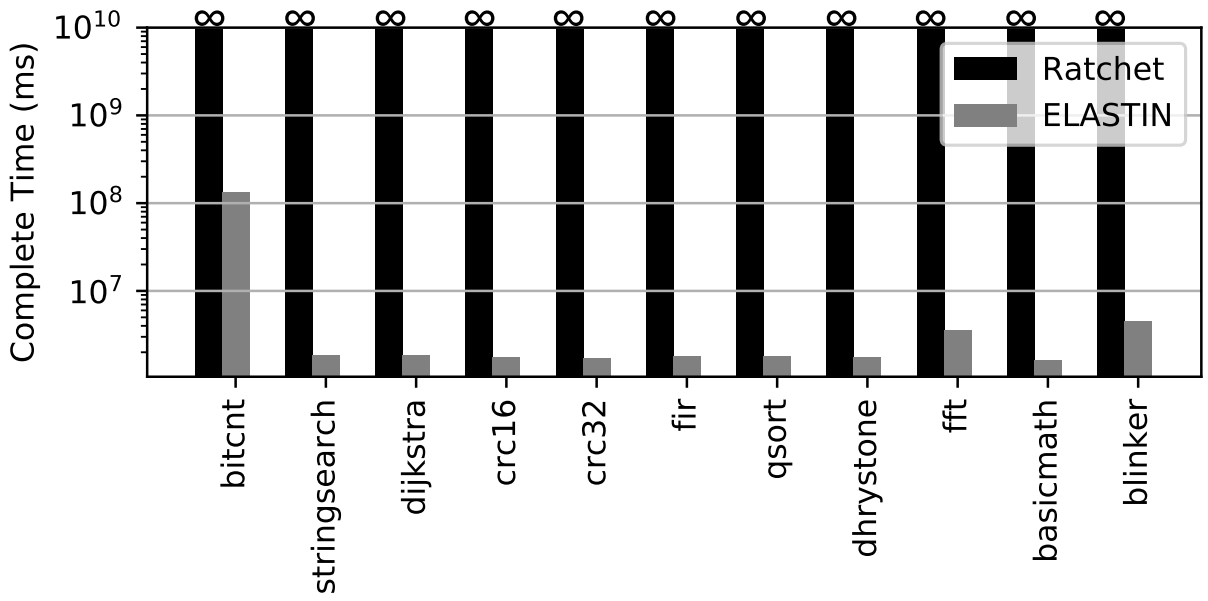


Figure 3.14. Elastin's robustness against capacitor malfunction

controlled memory protection without MMU or TLB; Elastin can be regarded as library OS that only offers memory protection and timer interrupt. In general, energy-harvesting devices do not run OS. When a new interval starts at each timer expiration, Elastin clears the write permission of all the pages written in the previous interval and checkpoints all registers including a program counter as a recovery point. Elastin reconfigures the checkpoint interval and the page size based not only on the underlying energy harvesting quality but also on the observed forward progress. Consequently, Elastin achieves stagnation-free intermittent computation, ensuring forward progress across power outages.

4. ROCKCLIMB: COMPILER-DIRECTED HIGH-PERFORMANCE INTERMITTENT COMPUTATION WITH POWER FAILURE IMMUNITY

In this chapter, we present ROCKCLIMB, a rollback-free and memory-log-free software recovery solution. ROCKCLIMB runs program without having power failure interrupt, achieving high-performance intermittent computation (TS1). ROCKCLIMB outperforms the state-of-the-art by 5%—550% on average in various energy harvesting conditions.

4.1 Introduction

Software-based recovery solutions statically partition the program into a series of recoverable tasks so that their re-execution can always result in the same and correct output. However, they can cause a severe issue, i.e., stagnating the system while consuming hard-won energy. Supposedly, if power outages repeatedly occur within a particular task before it ends, the solutions continually attempt to re-execute the same interrupted task, i.e., *stagnation*. In other words, although the software-based recovery schemes ensure correct recovery across power failure, they cannot provide any service to end-users at all. Therefore, we see there is an urgent need for recovery solutions to address the stagnation problem.

To overcome the challenge, we introduce power failure immunity (PFI), a novel program execution property for achieving energy-efficient intermittent computation. PFI ensures that each code region can fail at most once, i.e., a single in-region outage, regardless of power failure frequency. If a region ever encounters power failure, it never fails again during the re-execution—as if it was immunized after the first failure. The *never-fail-again* nature makes it possible for intermittent computation schemes to take the aggressive region formation (i.e., long regions) without the expensive re-execution penalty.

In particular, we leverage PFI to achieve *rollback-free* intermittent computation without expensive hardware support, e.g., just-in-time (JIT) checkpointing of nonvolatile processors which preserves their volatile states when power is about to be cut off [14], [15], [17], [21], [125]. To achieve the rollback freedom, we propose ROCKCLIMB guarantees that PFI-enforced regions never

fail, i.e., there is no in-region outage at all. At a high level, RockClimb checks if a fully buffered energy is secured at each region boundary to ensure the completion of the next region without power failure. If it is not secured, RockClimb waits at the boundary until the energy buffer is fully charged before executing the following region. The upshot is that the rollback-free nature of RockClimb obviates the need to perform logging for each memory write—required for prior work [1], [2], [11]–[13], [98] to achieve rollback recovery of power failure.

Although no region is power-interrupted, a power outage can still occur while RockClimb waits for the energy buffer to be fully charged at a region boundary; volatile states, i.e., registers, can still be lost upon an outage. To address the issue, our compiler leverages a novel optimization called *distributed checkpointing* that saves only essential registers without compromising the recovery guarantee. Unlike prior rollback recovery schemes [2], [4], [11], [12] that insert checkpoints (i.e., store instructions saving registers to NVM) at the beginning of each region/task boundary, *distributed checkpointing* spreads them out where each register is defined, thereby eliminating unnecessary checkpoints and their energy consumption.

Consequently, PFI+RockClimb achieves high-performance intermittent computation, ensuring forward execution progress and maximizing it even in the presence of frequent power outages. Our real board experiments demonstrate that PFI-enforced program never suffers from stagnation, and PFI+RockClimb outperforms the state-of-the-art intermittent computation work by 5%—550% on average depending on power failure behaviors.

- We define PFI as a basic program execution property for achieving energy-efficient intermittent computation and implement its compiler-directed enforcement.
- Our compiler automatically enforces PFI forming stagnation-free regions in 1.4 seconds on average, unlike dynamic approaches that take many hours of testing but can only remove the stagnation found on tested paths.
- We propose RockClimb that can ensure PFI-enforced regions never fail, i.e., there is no in-region outage at all.
- We propose a new compiler optimization called distributed checkpointing that can remove unnecessary checkpoints thus extending forward progress with their saved energy.

4.2 Background and Challenges

4.2.1 Expensive Centralized Checkpointing

Software-based recovery schemes achieve crash consistency by handling antidependences and checkpointing registers [4], [7], [18], [44]–[63]. In particular, no matter how antidependence is addressed, the prior works [2], [4], [11], [12] checkpoint every volatile input at the beginning of each region, thus being called *centralized checkpointing*. More precisely, they checkpoint all live-in [65] registers—that hold live [65] values at the beginning of a region—for every region at its entry in case it is interrupted due to power failure; in the wake of the failure, the interrupted region must be restarted from the entry for recovery. Unfortunately, many of live-in register checkpoints tend to be unnecessary wasting harvested energy that could otherwise be used to make further execution progress; it is important to note that because of the NVM write latency/energy, checkpoints are the most expensive instruction in energy harvesting MCUs.

We observe that the live-in registers are often not used in the current region, but read in the later regions; the more regions the live-range [65] of registers spans, the more redundant checkpoint stores the centralized checkpointing generates. As an extreme example, if registers are defined at the beginning of program and read at the end of it, they must be checkpointed at every single region entry. The takeaway is that although the registers are not used in the current region, the centralized checkpointing has no choice but to save them, otherwise their values are lost upon power failure; *this is why all the prior works end up checkpointing all live-in registers every time a region starts*. With that in mind, we propose a new compiler optimization called *distributed checkpointing*. It spreads out checkpoint stores to where volatile registers become live-out [65], i.e., at their last-update point in each region.[1] In this way, they do not have to be checkpointed repeatedly in the following regions unless they are updated and live-out again. Section 4.5.2 details the distributed checkpointing.

[1]↑ Distributed checkpointing is akin to incremental checkpointing that can be achieved with either hardware [126] or runtime support [127], because they checkpoint only updated registers. However, all updated registers do not require checkpointing, e.g., some registers could be re-defined before their use, in which case the incremental checkpointing wastes harvested energy by persisting such dead registers unnecessarily. In contrast, our distributed checkpointing preserves only essential (live-out) registers by taking into account their liveness at compile time.

Table 4.1. Comparison of prior software solutions for a stagnation problem in energy harvesting systems. Partitioning time means how long it takes to form stagnation-free regions/tasks. H/W Support represents an energy debugger requirement while User Intervention means whether a programmer must use a special programming language. Log indicates whether the work requires a logging mechanism for memory restoration. Re-execution shows whether the work involves re-execution inherently.

	Auto-tuning [11]	Chinchilla [12]	**PFI+RockClimb**
Partitioning Time	Very long	Very long	Short
Analysis	Dynamic	Dynamic	Static
Memory Log	Yes	Yes	No
HW Support	No	Yes	No
User Intervention	No	Yes (energy debugging)	No
Checkpoint Type	Centralized	Centralized	Distributed
Re-execution	Yes	Yes	No

To address the stagnation problem, the state-of-the-art works use dynamic testing approaches by using an energy debugger [12], [13], [66], [67] or an auto-tuning framework [11] as summarized in Table 4.1. However, such dynamic testing approaches have several limitations.

The energy debugger based approach requires multi-step *expert-level* user interventions for precise diagnosis [12], [13], [66], [67]. First, users should manually measure basic blocks' energy consumption with randomized inputs. Second, users need to compare the energy consumption of a given basic block to the total energy availability obtained by estimating the storage capacity of the energy buffer. Third, if the basic block consumes more energy than the available amount thus being vulnerable to stagnation, users should rewrite the code or let the prior work [13] split the block to smaller pieces with inserting checkpoint stores and memory logs therein.

The crux of the problem with the debugger based schemes is that the energy profiling [13] takes about 30 minutes on average even for toy applications, which makes the schemes impractical. Furthermore, the resulting program can still suffer stagnation provided some of untested program paths is taken; this is technically possible since users cannot cover all possible paths with dynamic testing due to its inherent *unsoundness* [128]. Due to the issue, the state-of-the-art work Chinchilla [12] ends up inserting its region boundary at each basic block, rendering the regions too small—though it can adaptively skip register checkpointing when energy harvesting condition is good.

Table 4.2. Comparison of stagnation-aware region formation schemes in terms of the time taken to complete the region formation

Benchmark Application [3], [11], [12], [25]	Auto-tuning [11] (in hours)	Energy-debugging [12], [13], [66] (in hours)	Compiler-directed PFI enforcement (in seconds)
basicmath	124.2	9.3	2
blinker	11.2	0.5	1
bitcnt	9.6	9.7	4
crc16	2.3	1.7	1
crc32	18.1	3.7	1
dijkstra	7.2	5	1
fft	8.9	19.3	2
fir	6.0	2	1
dhrystone	1.0	5	1
stringsearch	6.2	22	2
qsort	53.5	4	1
geomean	**9.6 hours**	**4.7 hours**	**1.4 seconds**

On the other hand, the auto-tuning based dynamic approach first tests a user-defined range of region sizes (instruction counts) for a given power failure trace and then picks the best-performing size for all regions of each program. However, it is based on *one-size-fits-all* assumption, i.e., every region size is identical. Also, due to the large search space, the tuning time takes a while. Unfortunately, users must go through the same tuning procedure again for the change of the energy harvesting condition, which is frequent and unpredictable in reality.

To evaluate the usability of the prior dynamic testing approaches, we measured the total elapsed time for completion of their recoverable region formation; two different test inputs were used for each of 11 applications. For the auto-tuning approach, we tested 200 different region size variants as suggested in the original work [11] with two different power failure traces. As shown in Table 4.2, the auto-tuning (2nd column) and the energy-debugging (3rd column) approaches take a considerable amount of time for each application (up to more than 5 days as in *basicmath*). Note that although both dynamic approaches finally form regions after many hours, such a high cost has to be paid anew for different program/input combinations and various power failure behaviors. Unfortunately, this is a serious problem in that energy harvesting systems inevitably encounter the significant change of the failure behavior—because the underlying harvesting condition often

unpredictably varies over time. In contrast, our proposal only requires a few seconds of compilation time as shown in the 4th column in the table. On average, PFI compiler is five orders of magnitude faster than the both dynamic testing approaches. More importantly, unlike the approaches, our static analysis can guarantee stagnation-free execution regardless of program paths and inputs.

4.3 Design

The goal of this work is to achieve high-performance energy harvesting systems. As the first step to achieving the goal, we define power failure immunity (PFI), a basic execution property of intermittent program. PFI ensures that each recoverable region of program never fails more than once i.e., at most single in-region outage (Section 4.3.1). Thus, PFI-enforced regions are robust against both stagnation and expensive re-executions across power failure, achieving energy-efficient power failure recovery.

The second step is leveraging PFI to achieve rollback-free and high-performance intermittent computation—that we call RockClimb—where no region is power-interrupted (Section 4.3.2). That is, RockClimb not only ensures that hard-won energy is never wasted for region re-execution, but also maximizes the forward execution progress fully utilizing the energy only for computation. Furthermore, since all regions are sure to finish thanks to RockClimb, it can eliminate expensive per-region memory logging—that is required by prior work for crash consistent rollback recovery—without compromising the correctness guarantee. The rest of this section details the PFI and its compiler-directed enforcement, the workflow of which is shown in Figure 4.1, and shows how it guarantees that no region ever fails.

4.3.1 PFI: Power Failure Immunity

To prevent repetitive in-region outages, we leverage two important observations. First, energy harvesting systems do not start to operate their microcontoller (MCU) until the energy buffer (capacitor) is fully charged as with virtually all commodity systems, e.g., WISP [129]. That is, when the MCU is ready to resume the execution in the wake of power outage, the capacitor is always sure to have the fully buffered (charged) energy at the starting point of the resumption. The implication is that the power-interrupted region can make as much progress as the full energy buffer

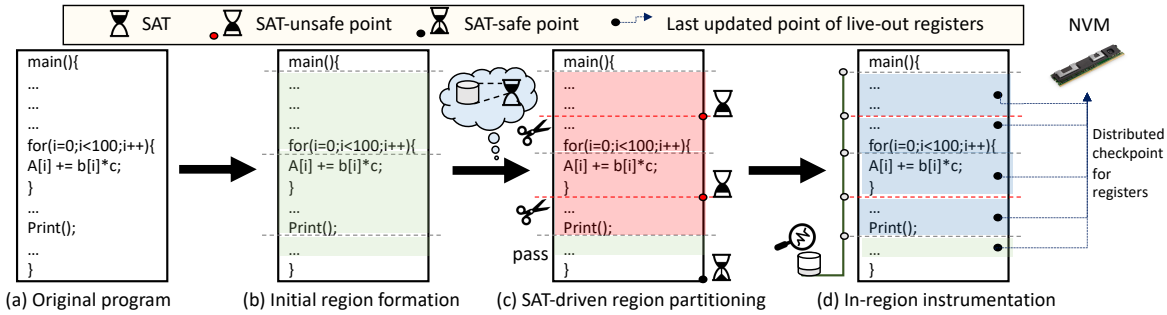


Figure 4.1. Workflow of PFI compiler: it partitions program into a series of PFI-enforced regions and instruments them to achieve rollback-free and high-performance intermittent computation.

allows, even if there is no additional energy is harvested. We refer to the minimum progress time, for which the MCU can be sustained under the fully buffered energy, as safe active time (SAT).

The second observation is that if the worst-case execution time (WCET) of any region is shorter than the SAT, the region is assured to finish with no power failure under the fully buffered energy.

$$\text{PFI enforcement constraint: } WCET\,(r) < SAT\,(\mu) \tag{4.1}$$

With that in mind, for a given region (r) and the underlying MCU (μ), we formulate the problem of ensuring the forward execution progress as Eq.4.1 above.

Thus, PFI can achieve stagnation freedom by partitioning the original program into *SAT-safe regions*, each of which satisfies the PFI constraint Eq.4.1; even if the SAT-safe regions may encounter power failure, they **never fail again** upon recovery from the failure. In other words, when power comes back, the previously interrupted region never retreats before reaching the end of the region, which ensures forward progress to the next region without exception.

SAT Calculation under Worst-Case Execution Scenario: To calculate the SAT for a given MCU's full energy buffer in a *sound* way (where all regions satisfy the PFI constraint Eq.4.1), we must consider the worst-case scenario of MCU operation, which would otherwise fail to achieve PFI for those regions power-interrupted under the scenario. Hence, our PFI compiler considers the most harsh environmental setting where there is no harvested energy, and the MCU consumes the maximum amount of energy all the time. That is, the compiler calculates the SAT by analyzing how long program can sustain its execution, under the maximum power consumption mode of the

MCU—which drains the energy from the capacitor at the highest rate—to take into account the worst-case scenario.

PFI Region Formation: To form PFI-enforced regions that satisfy the above constraint Eq.4.1, the compiler takes a 2-step approach. First, it forms initial regions at function call boundaries and loop headers. As shown in Figure 4.1 (b), the input program (a) gets to have a region boundary at a `Print()` callsite and the entry of a `for` loop.

Second, after finishing the initial region formation, the compiler performs per-region WCET analysis to check if the initial regions satisfy the PFI constraint Eq.4.1. If so, the regions remain the same—until they are instrumented later for rollback-free intermittent computation; otherwise, the compiler partitions the SAT-unsafe region, that violates the PFI constraint Eq.4.1, into a series of SAT-safe regions; the SAT-driven partitioning might need to be repeated if the remainder of the cut is still too long to satisfy the constraint. For example, as shown in Figure 4.1 (c), the first two initial regions in (b) are both cut at the point where their WCET hits SAT. As a result, every program point belongs to one of SAT-safe regions where PFI is enforced.

4.3.2 RockClimb: Never Fail Whatsoever!

Once SAT-safe regions are formed, our compiler enables RockClimb that leverages the PFI as a basis for achieving rollback-free intermittent computation, i.e., extending the PFI to much stronger guarantee that **no region ever fails**. In fact, the name RockClimb is inspired by rock climbing; climbers divide their route into multiple sections, and at the entry of each section they usually rest eating energy bars until they get powered up enough to pass the section. Similarly, to complete each PFI-enforced region with no power failure, RockClimb checks the energy buffer at each region boundary. If the buffer is not fully charged, RockClimb waits for the buffer to secure the full energy before starting the next region; otherwise, it is immediately started with the guarantee of failure-free completion—because a PFI-enforced region can always finish with a fully buffered capacitor.[2]

In particular, RockClimb's guarantee of no in-region failure simplifies achieving crash consistency obviating the memory logs in each region. As discussed in Section 4.2.1, a root cause of

[2]↑We assume that the energy harvesting system does not suffer accidental reliability issues such as energetic particle striking or capacitor malfunctioning in the circuit.

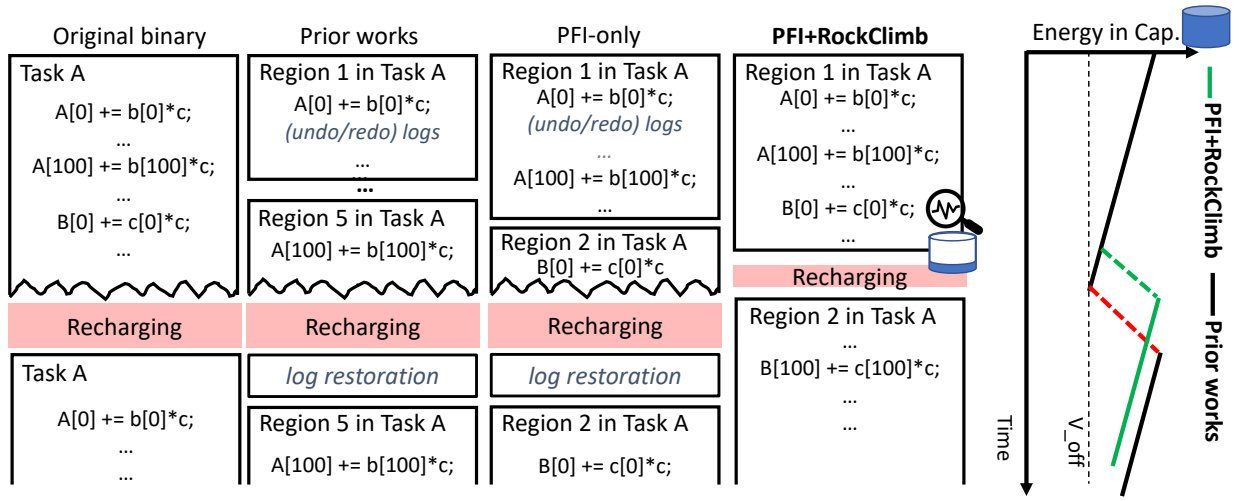


Figure 4.2. Comparison of intermittent computation schemes: Each scheme runs the same program (Task A). While prior works form many regions, e.g., Region 1∼5 in Task A, PFI generates a few regions, and PFI+RockClimb further lengthens the region size and eliminates the re-execution.

the memory inconsistency is that in the wake of power failure, program control rolls back across antidependence, reading values updated by stores left behind the failure. That is why prior work logs memory inputs of each region to make a copy of the original values before they are overwritten by antidependent store instructions.

On the contrary, since RockClimb's regions are never power-interrupted (thus no rollback), they do not have to log memory inputs at all; the absence of rollback recovery means no need to handle the restoration of memory inputs. The upshot is that RockClimb's rollback-freedom saves the high energy/latency of the NVM logging stores, thereby achieving an energy-efficient and high-performance energy harvesting system. Figure 4.2 highlights RockClimb compared to prior works that partition program into several regions with memory logs for recovery. While the prior works keep spending their energy for logging, restoring, and re-executing as shown in the figure, RockClimb here makes a further forward progress due to its log-free and re-execution-free intermittent computation.

Region Instrumentation: When a program control reaches the end of a region, RockClimb checks the energy availability (full capacitance) before starting the next region. For this purpose, as shown in left side of Figure 4.1 (d), the compiler thus inserts—at each region boundary—the

voltage-level checking code offered by commodity energy harvesting systems, e.g., TI-MSP430's power management library [38], [130] supports the checking through a voltage comparator interrupt; Section 4.5.1 offers more details.

Finally, our compiler, if necessary, inserts checkpointing stores to save volatile registers in some regions. Although PFI-enforced regions are never interrupted by power failure, it can still occur at region boundaries while RockClimb waits the capacitor to be fully charged. In this case, unlike NVM resident data, volatile registers are lost on power failure. To this end, as shown in the right side of Figure 4.1 (d), the compiler checkpoints registers using a novel compiler optimization called *distributed checkpointing* that saves only essential registers without compromising the recovery guarantee (Section 4.5.2).

4.4 Implementation

4.4.1 SAT Calculation

To obtain the safe active time (SAT), our PFI compiler first measures the available energy input as:

$$Available\ Energy\ Input = \frac{1}{2} C_{buf} * (V_{max}^2 - V_{min}^2), \tag{4.2}$$

where $C_{buf}, V_{max}, V_{min}$ are capacitance, MCU power-on voltage level, and MCU power-off voltage level, respectively. Then, the compiler measures MCU's energy consumption during operation as [118]:

$$E_{tot} = P_{tot} t = V_{dd} I_{leak} t \ + \ C_{msp} V_{dd}^2, \tag{4.3}$$

where $V_{dd}, I_{leak}, C_{msp}$ are input voltage to MCU, leakage current, and the MCU capacitance, respectively. However, the input voltage (V_{dd}) is not constant; it decreases as the energy buffer is discharged (Section 4.3.1). With that in mind, we consider the MCU as a resistance-capacitor (RC) circuit, i.e., our PFI compiler substitutes the input voltage with $v_o(t) = V_o e^{-t/CR}$, where the capacitance (C) is the same as C_{buf}, and the resistance (R) can be estimated as $R = V/I$. Here, we can readily get the V and I as the operating voltage and the maximum current, respectively, from the MCU manual. If a certain MCU's manual does not specify them, our compiler can adapt the typical leakage current and capacitance model [118].

The key insight of PFI is that, to guarantee the forward progress, the available energy input obtained by Eq. 4.2 should be always greater than the energy consumption of the underlying MCU given by Eq. 4.3. In light of this, RockClimb obtains the SAT by calculating a threshold time t in the following formula (Eq. 4.4).

$$\frac{1}{2}C_{buf} * (V_{max}^2 - V_{min}^2) > V_o e^{-t/CR}(I_{leak})t \; + \; C_{msp}(V_o e^{-t/CR})^2 \tag{4.4}$$

In particular, SAT should also cover the system recovery cost to safely restore the checkpointed registers at the system reboot time in the wake of power failure. We use the simple energy profiling model—which can be further improved by using a recent advanced model [130], [131]. For simplicity, our PFI compiler conservatively updates SAT (Eq.4.2) as $SAT = SAT - Recovery\_Cost$ by assuming all registers are restored upon recovery as with prior work [4].

4.4.2 WCET Analysis

First of all, our WCET calculation is straightforward for two reasons: (1) the energy-harvesting MCU architecture is simple, i.e., a cache-free single in-order core, unlike multi-core systems backed with out-of-order execution and deep cache hierarchy [132], [133], and (2) unlike traditional WCET calculation [134], [135], ours does not require whole program analysis.

To analyze WCET for each initial region shown in Figure 4.1(a), our PFI compiler navigates all possible paths in region-based control flow subgraph; while whole-program-analysis based WCET calculation is challenging, our **region-based** (intra-region) analysis makes it possible to run the WCET analysis for all the benchmarks we tested.

Also, our compiler identifies a basic block that has initial region boundaries in the middle of it, and splits it into different basic blocks. This allows that region boundaries always start at the beginning of basic blocks, thus facilitating the next SAT-driven region partitioning.

For instruction-level WCET calculation, we build a cost model by referring to the MCU manual which gives the execution cycles of each instruction [136][3]; if executing some instructions takes a range of cycles, our compiler takes the worst latency. Since the worst-case execution cycles of

[3]↑For example, a simple register update instruction takes 1 cycle while a load instruction takes 5 cycles at least.

instructions are fixed, the timing cost model is simple and safe unlike the profile-based energy consumption model [13] that can be inaccurate in different execution environments.

4.4.3 SAT-Driven Region Formation

Once the SAT is obtained, our PFI compiler statically converts it to the MCU cycles and splits those initial regions, that are SAT-unsafe, into SAT-safe regions. As shown in Figure 4.3(a), the compiler keeps accumulating the execution time (cycles) of instructions on every path in a given initial region, i.e., the accumulated sum is the WCET of the path between the region entry and the current instruction just visited there.

During the instruction time accumulation on each path, if the sum becomes greater than or equal to the SAT (i.e., the current instruction and its successors are susceptible to power failure), then the compiler cuts the susceptible path by placing a new region boundary before the current instruction with zeroing the sum for a further partitioning.[4] This happens recursively until the last instruction of the path is reached. Figure 4.3(b) shows the final shape after partitioning the original region with SAT of 200 cycles.

Algorithm 1 Region Formation Algorithm

for *each basic block* bb_i *in CFG* **do**
  $Cycle_{bb_i} \leftarrow Cycle\_ori_{bb_i} + CkptCycles_{bb_i}$
  $IncomeCycle_{bb_i} \leftarrow 0$
end
for *each basic block* bb_i *in program topological order* **do**
  if bb_i *starts with region boundary* **then**
    accum_cycle $\leftarrow Cycle_{bb_i}$
  else
    accum_cycle $\leftarrow Cycle_{bb_i} + IncomeCycle_{bb_i}$
  end
  while $accum\_cycle > threshold_{time}$ **do**
    place boundary and split bb_i into bb_i' and bb_i
    recalculate $Cycle\_ori_{bb_i}$ and $CkptCycles_{bb_i}$
    accum_cycle $\leftarrow Cycle\_ori_{bb_i} + CkptCycles_{bb_i}$
  end
end

[4] Technically, the region boundary instruction is a checkpoint store for saving a program counter so that it serves as a recovery point in case the following region is power-interrupted.

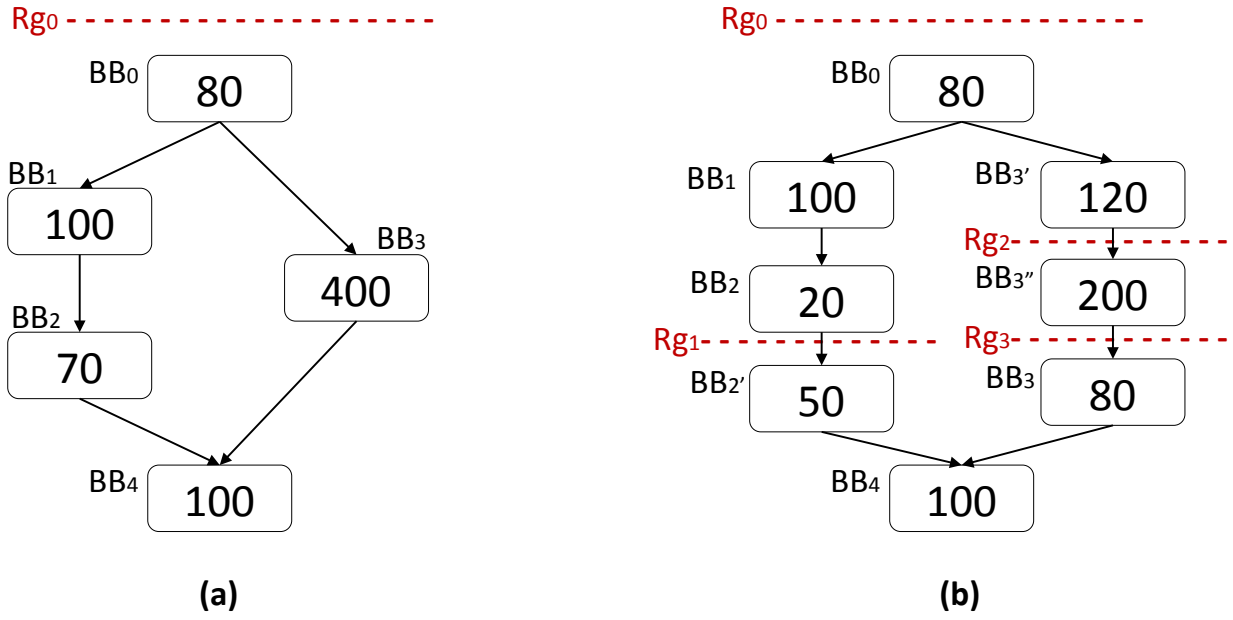


Figure 4.3. Region partitioning with the SAT threshold of 200; the number in each basic block (box) represents its total execution cycles, and dashed lines represent region boundaries.

Algorithm 1 details the overall region formation process highlighting how to cut and reform SAT-unsafe regions. In a given initial region, the PFI compiler traverses the CFG in a topological order. During the path traversal, it updates the sum of current and incoming basic blocks' cycles by using the instruction-level cost model from the beginning of the latest region boundary along the path (line 5∼16). If the sum becomes greater than SAT before the next region boundary is reached, the compiler places a cut (line 11∼15). After re-partitioning, the compiler inserts register checkpoints with distributed checkpointing (Section 4.5.2).

Loops: To handle loops, our PFI compiler inserts a region boundary in the loop header. The compiler splits the loop body if its WCET is greater than SAT. Otherwise, the compiler tries to extend such a short loop body since it can cause more live-out registers and checkpoint stores across the region boundary in the loop header. That is, to address this issue, our compiler repeatedly unrolls such a loop as long as the WCET of the loop body is smaller than SAT. By maximizing the loop body in this way, the compiler can minimize the number of live-out registers in the loop, thereby reducing the number of checkpoint stores to be inserted. Currently, for those loops whose

iteration count is statically unknown, the compiler just inserts a region boundary in the loop header without performing the unrolling.

IO Operation: Our PFI compiler treats an IO operation as a separate region. Since RockClimb always secures the full capacitance before starting a region, the IO becomes failure-atomic operation. This is exactly what IO operations pursue to ensure the freshness of the IO results. However, it is the IO designer's responsibility to ensure the operation can be completed in one capacitor charge cycle, i.e., the fully charged energy should afford to finish the IO operation.

Although individual IO operations are failure-atomic, their combination can violate the PFI constraint (Eq.4.1), provided they are performed in parallel. To address the issue, we should conservatively estimate the SAT by covering the total current consumption of the IO operations. For this purpose, our compiler updates the resistance (R) of Eq.4.4 with the sum of each IO's maximum current, i.e., $R = \frac{V}{I_{MCU}+I_{IO_1}+I_{IO_2}+...+I_{IO_n}}$; we assume that the required values can be found from the IO manuals.

4.4.4 Discussion

As discussed in Section 4.4.2, to meet the PFI enforcement constraint (Eq.4.1) for region formation, our compiler simply adds up the worst-case instruction latencies and places a region boundary at the point where the accumulated sum becomes greater than the SAT—without considering the instruction pipeline. Thus, such a conservative WCET analysis tends to generate smaller regions than traditional WCET analysis. The rationale behind is that the smaller region leaves residual energy in the capacitor when the region ends. In particular, the residual energy can serve as a safe margin[5] to guarantee the soundness of PFI-enforcement even when the full capacitance is not secured due to reliability issues. For example, even if a capacitor malfunctions holding a smaller amount of energy than normal due to its physical properties, e.g., abnormal leakage or aging-induced damage, PFI-enforced regions do not exceed the SAT.

[5]↑The safe margin does not harm the performance because the residual energy is going to be picked up by RockClimb, i.e., the charging time before executing the next region becomes shorter.

4.5 Optimization

4.5.1 Securing Full Capacitance for Rollback-Free Computation

At each region boundary of SAT-safe regions that satisfy the PFI constraint Eq.4.1, our compiler enables RockClimb that dynamically checks if the full capacitance is secured, which would otherwise wait for the energy buffer to be fully charged, before starting the next region. For this purpose, RockClimb leverages a voltage emergency interrupt of commodity energy harvesting systems, e,g., TI-MSP430 [38], which can compare a current voltage level to a certain voltage threshold that can be controllable by software. That is, at each region boundary, RockClimb enables the voltage interrupt by controlling the interrupt vector and immediately starts the next region unless the interrupt is generated; otherwise, RockClimb puts the microcontroller (MCU) to a power-down mode so that it can be rebooted when the buffer is fully charged [130]. The implication is two-fold. First, no-interruption here means that the MCU is directly powered by the energy harvesting source with the energy buffer fully charged, and therefore the next region is guaranteed to finish without power failure. Second, the voltage interrupt should be disabled before executing any instruction of the next region. That is, at the beginning of a region, the compiler inserts the code that disables the interrupt.

4.5.2 Compiler Optimization: Distributed Checkpointing

Unlike centralized checkpointing, our compiler does not checkpoint all live-in registers at each region boundary. Instead, it spreads the checkpoints (store instructions saving registers in NVM) out where each register is defined; they are often distributed to many regions, thus being called *distributed checkpointing*. More precisely, the compiler checkpoints the updated register of each region, which is used in following regions and therefore called *live-out* [65], right after the register update. In data flow terms, we define the output of region r as the live-out registers defined in the region—the values that are written and downward-exposed [65]: $ckpt_r = Def_r \cap LiveOut_r$, where Def_r is the set of registers defined in r and $LiveOut_r$ is the set of live-out registers of r. For each region r, the compiler checkpoints only the registers belonging to $ckpt_r$ as soon as they are defined in the region.

The distributed checkpointing has several implications. First, if the register is defined multiple times in a region, the compiler only checkpoints the last-updated value used by later regions, i.e., the live-out value. Second, since the compiler checkpoints such a live-out register right after its update, each checkpoint shares the same register and the value with the preceding instruction that updates the register. Here, the benefit is two-fold: (1) minimal dynamic switching activity in the circuit; in contrast, two consecutive instructions with different operands/values increase the power consumption by 20% due to the switching activity [137], and (2) rare chance to increase the peak power consumption—typically made by consecutive checkpoints (NVM stores)—since no two checkpoints are consecutively placed in our distributed checkpointing; even if registers are updated in a row, their checkpoints are interleaved with the instructions that update the registers, i.e., checkpoints are always separated. Third, at the beginning of each region, all input registers to the region are already sure to have been checkpointed; thus, no action is required when each region entry is reached—unlike traditional JIT checkpointing that must save all registers before impending power failure [3], [7], [17], [125]. Fourth, once a register is checkpointed, no further checkpoint is necessary across regions unless the register is redefined and becomes live-out again. That is, unlike centralized checkpointing, the live-in registers of each region which are not written in the region, do not have to be checkpointed; again they are known to have already been checkpointed somewhere before the region entry. Finally, the compiler can minimize the number of necessary checkpoints.

Figure 4.4 shows how the compiler removes a majority of the checkpoints introduced by the centralized checkpointing of prior works [2], [4], [11], [12]. Here, $r1$ is defined on the top region (box) and used in other regions. In particular, $r1$ is live at the entry of all regions except for the top one. In Figure 4.4(a), the centralized checkpointing inserts five checkpoints, each of which saves the live register $r1$ (and other live-ins if exist) at the entry of the bottom five regions. Note that checkpointing $r1$ at the entry of the 2nd region on the left branch is redundant since the predecessor region has already checkpointed $r1$. Even worse, in the 2nd region on the right branch, the centralized checkpointing stores $r1$ though it is not even used there. In contrast, our distributed checkpointing stores $r1$ only once right after it is defined on the top region as shown in Figure 4.4(b). Here, our SAT-driven region formation is aware of the additional code for both volt-

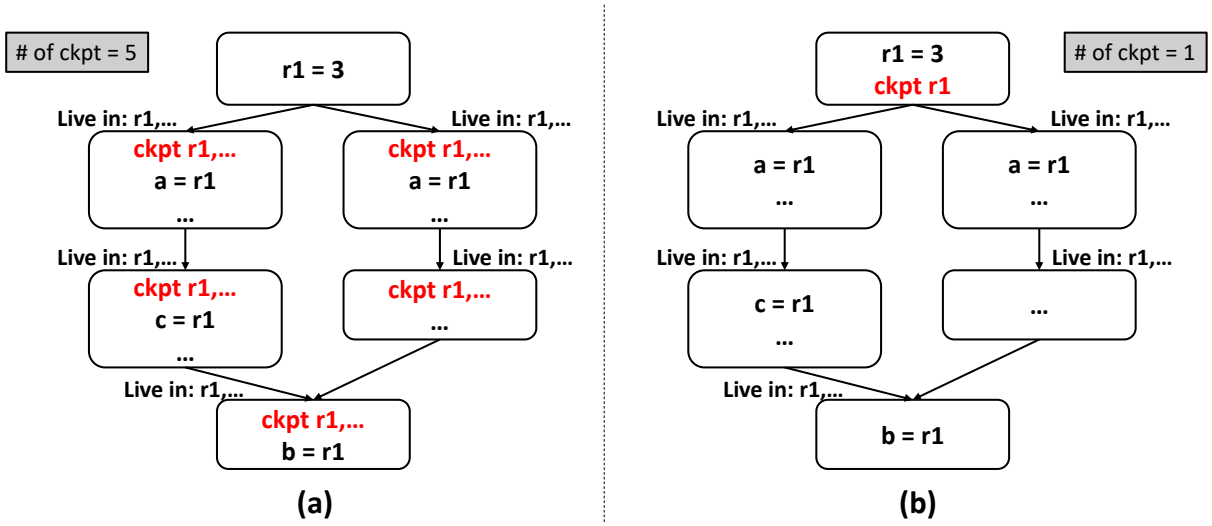


Figure 4.4. Checkpoint reduction by distributed checkpointing: (a) centralized checkpointing and (b) distributed checkpointing. Each box represents a program region.

age interrupt controlling and distributed checking already. Thus, the compiler strictly maintains the PFI-enforcement for all regions.

4.6 Evaluation

4.6.1 Experimental Setting

We implemented novel compiler techniques described in Section 4.4 in the LLVM compiler infrastructure [120] and conducted experiments by running compute-intensive 11 benchmarks that are used in prior works [3], [25], [122], [123]; sensing applications are ruled out on purpose, because they mostly consist of I/O or sensor tasks that must be power-failure-atomic. This implies that the tasks must be formed that way by the system designer in the first place; once they are formed, it is technically impossible to partition them[11].

To evaluate the effectiveness of PFI+RockClimb in preventing stagnation, we conducted experiments using TI's MSP430FR5994 [138] evaluation board with Powercast P2110-EVB RF energy harvester [139] as our energy harvesting system testbed, following the same convention used by prior works [2], [6], [7], [12], [140]–[142]; we equipped the board with a 10μF capacitor which is used as energy storage of commodity systems such as WISP [129]. To power the energy

harvesting system, we used Powercast TX91501-3W transmitter emitting RF signal at 915 MHz center frequency to the system supplying 6.1 dBi patch antenna. We placed the RF transmitter as real energy source 50cm away from the energy harvesting system by default; we also varied the harvesting condition for sensitivity analysis (Section 4.6.3).

4.6.2 Stagnation Analysis

To analyze the stagnation problem, we conducted experiments by running the benchmark applications with 4 different schemes as shown in Figure 4.5: (1) Ratchet [4], (2) Chinchilla [12] the state-of-the-art software solution for stagnation freedom, (3) PFI-only scheme that partitions program to SAT-safe regions but leave memory logs and checkpoints therein for recovery, and (4) PFI+RockClimb as shown in the figure. Here, we assumed that the stagnation occurred if program had not finished within an hour; all benchmarks should have been finished within a couple of minutes. We avoid testing auto-tuning [11] since it requires too much tuning cost in real harvesting situation.

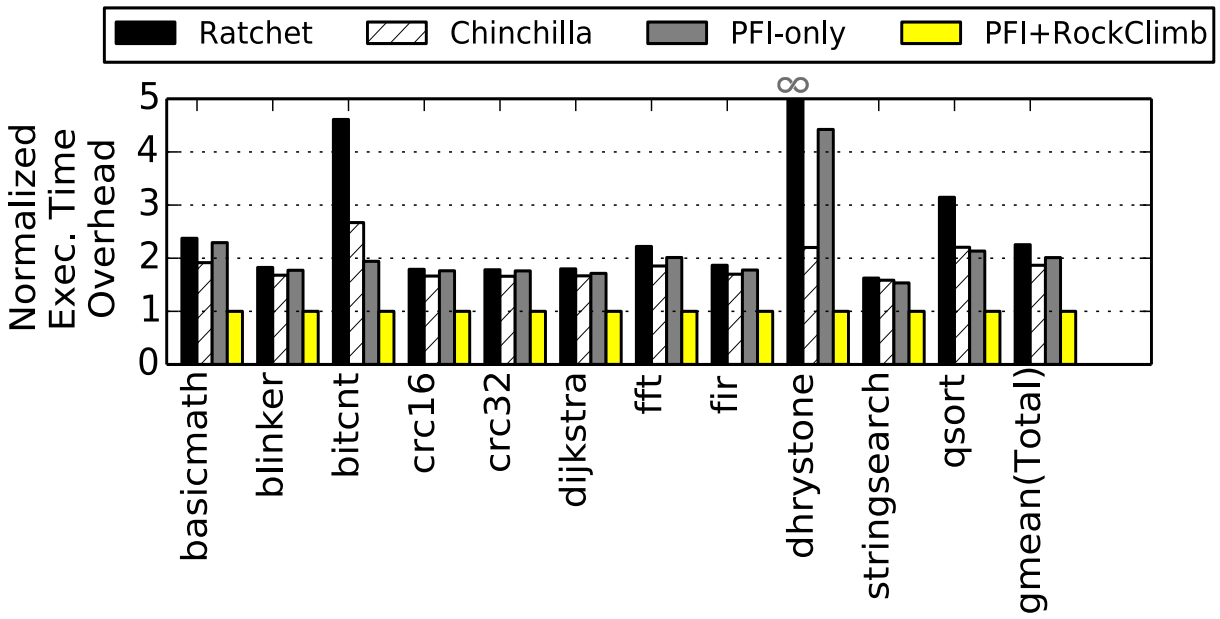


Figure 4.5. Performance results in real energy harvesting situation. We compare PFI+RockClimb with Ratchet and Chinchilla. Y-axis shows the normalized execution time to PFI+RockClimb. ∞ represents the stagnation problem.

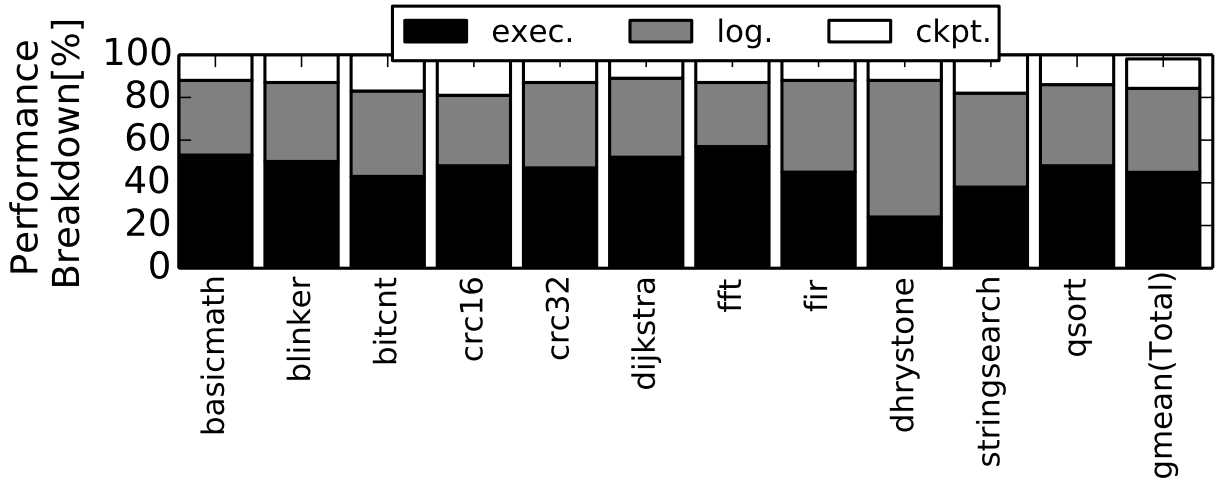


Figure 4.6. Performance Breakdown of PFI-only.

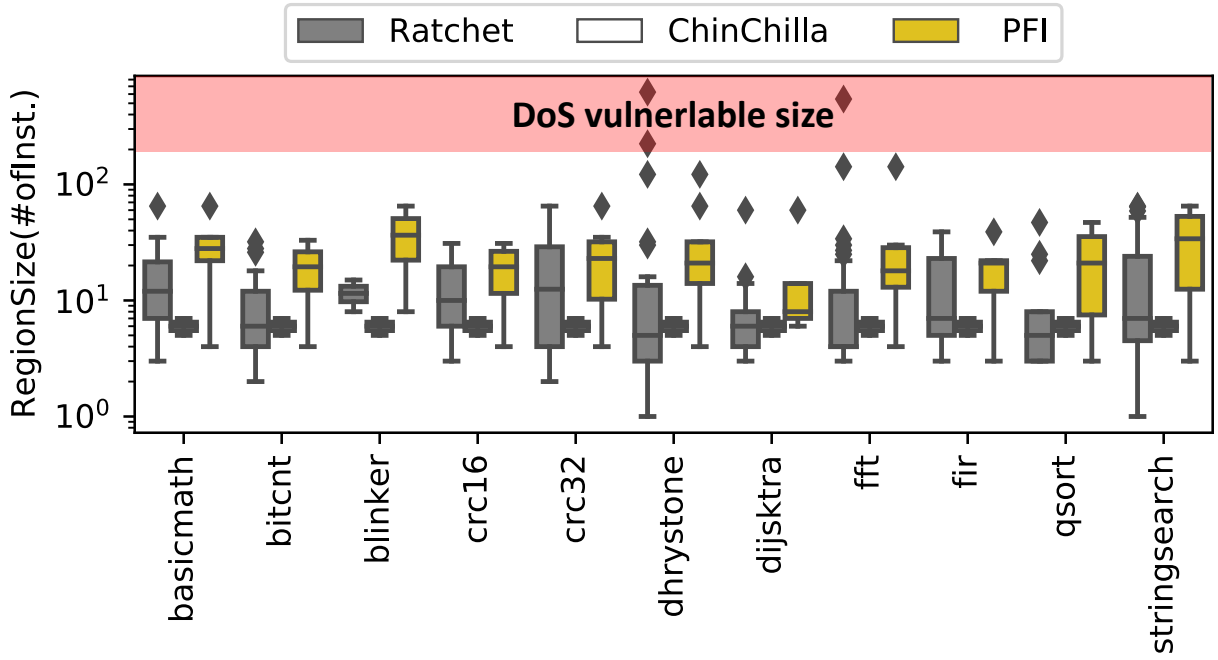


Figure 4.7. Region Size Comparison of Ratchet/Chinchilla/PFI.

Ratchet turned out to be the worst among the tested schemes; there was one stagnating application, i.e., *dhrystone*. Ratchet has many short idempotent regions generating a number of checkpoint stores; the more regions, the more their inputs in total. Thus, Ratchet causes relatively higher execution time overhead than others; the region size analysis is discussed in the next section.

On the other hand, Chinchilla, PFI-only, and PFI+RockClimb completed all applications. However, PFI-only and Chinchilla are 2x and 1.85x slower than PFI+RockClimb on average, respectively; Ratchet is 2.2x slower than PFI+RockClimb on average. This is mainly because they cause the overheads of re-execution, logging, and checkpointing—though Chinchilla was faster than PFI-only thanks to its adaptive execution that can skip checkpointing sometimes by considering energy source condition. Overall, PFI+RockClimb outperforms Ratchet, PFI-only, and Chinchilla thanks to the re-execution-free and memory-log-free nature.

In particular, when RockClimb is enabled for PFI-only, it becomes 2x faster. That is because RockClimb eliminates the logging (and their restoration) overhead. To see the benefit of RockClimb in more detail, we analyzed the performance breakdown of PFI-only into three parts: execution, checkpointing, and logging. As shown in Figure 4.6, the logging overhead is more than 40% on average, i.e., RockClimb can avoid the expensive cost and make a further forward progress.

Region Size Characteristics: To figure out the reason for the stagnation and characterize the regions of different schemes, we measured the region size of each application for Ratchet, Chinchilla, and PFI. Here, we counted the number of instructions executed during the execution of each region. As shown in Figure 4.7, there are two outliers, i.e., excessively long regions, in *dhrystone* for Ratchet, leading to stagnation. For other applications, Ratchet forms shorter regions than PFI on average. This is because Ratchet requires each region to be idempotent by cutting all antidependent store-load pairs, which makes the region size small [34], [35].

Unlike Ratchet, two schemes Chinchilla and PFI do not generate stagnating regions. In particular, PFI forms relatively longer regions than others on average; this trend demonstrates that PFI can aggressively increase the region size as long as it does not violate the constraint Eq.4.1. On the other hand, Chinchilla generates very short regions because it considers each basic block as a region, the entry of which—if not skipped by the adaptive execution—checkpoints all registers, to address the stagnation problem. Although the region size is short, Chinchilla outperforms Ratchet. That is because Chinchilla's adaptive execution can skip the register checkpointing according to the underlying power outage behavior.

Performance Modeling and Analysis: To analyze the performance benefit of RockClimb, we set the cost model of PFI+RockClimb and PFI-only schemes as following: $PFI\_only =$

$orig.exec + checkpoint + logging + reexecution + \sum_0^m T_{recharging}$, $PFI + RockClimb = orig.exec + checkpoint + \sum_0^n T_{wait}$. That is, PFI-only execution time (PFI_only) consists of original execution time ($orig.exec$), checkpoint time ($checkpoint$), logging time ($logging$), reexecution time ($reexecution$), and the sum of recharging time across the m number of power outages ($\sum_0^m T_{recharging}$). On the other hand, PFI+RockClimb execution time ($PFI + RockClimb$) is comprised of original execution time, checkpoint time, and the sum of waiting time across the n number of waits at region boundaries ($\sum_0^n T_{wait}$).

This implies that PFI+RockClimb can be technically slower than PFI-only when the total waiting time is higher than the sum of logging, reexecution, and total recharging time across the "m" number of power outages, i.e., $\sum_0^n T_{wait} > logging + reexecution + \sum_0^m T_{recharging}$. However, this is not practically impossible to happen because each waiting time is less than the recharging time, i.e., $T_{wait} < T_{recharging}$.

Moreover, even if the number of waits n could be greater than the recharging count m (i.e., the number of power outages), the total waiting time can be easily paid off by avoiding logging and reexecution time overheads. This is confirmed by our experiments; it turns out that PFI+RockClimb waited 10∼15 times while PFI-only had 3∼5 power outages on average, but PFI+RockClimb is almost 1.7x faster than PFI-only as shown in Figure 4.5. The results with various energy harvesting settings show the same trend (Section 4.6.3).

Distributed Checkpointing: To analyze the impact of distributed checkpointing, we measured the number of checkpoint stores of PFI at compile time and run time for both centralized and distributed checkpointing schemes. Figure 4.8 shows that when the distributed checkpointing is enabled, it can reduce the number of checkpoint stores of the variant of PFI—which uses centralized checkpointing on purpose— by 42% and 21% on average at compile time and run time, respectively.

To see the correlation between the checkpoint reduction and performance benefit, we also measured the performance of PFI+RockClimb without enabling the distributed checkpointing. When it is optimized with distributed checkpointing, PFI+RockClimb managed to improve the performance, achieving 1.16x speedup on average and up to 1.92x for *bitcnt*. Note that we refer to RockClimb as the optimized version that enables the distributed checkpointing by default.

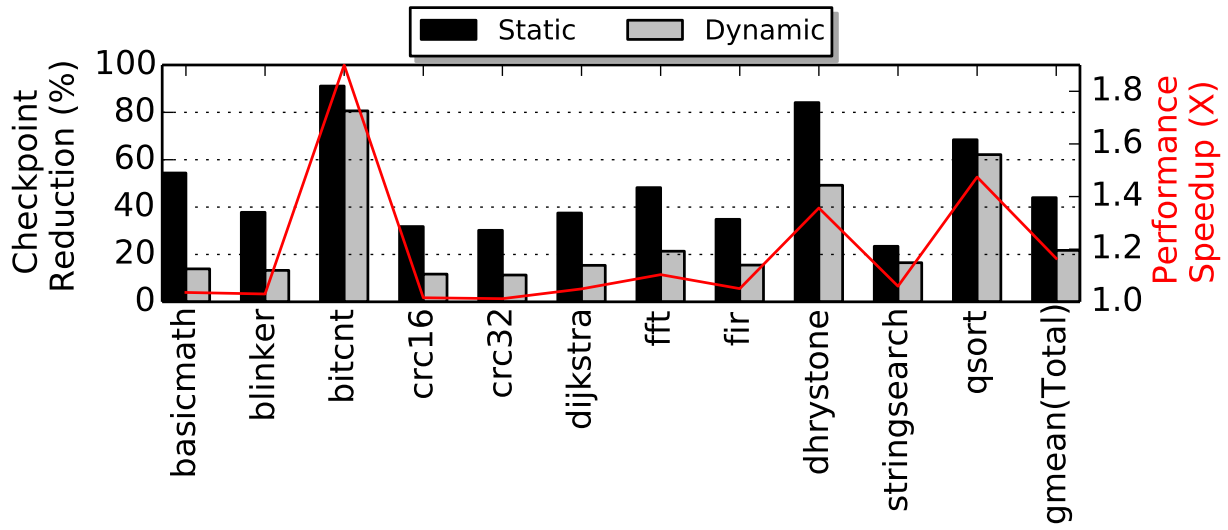


Figure 4.8. Checkpoint reduction by distributed checkpointing.

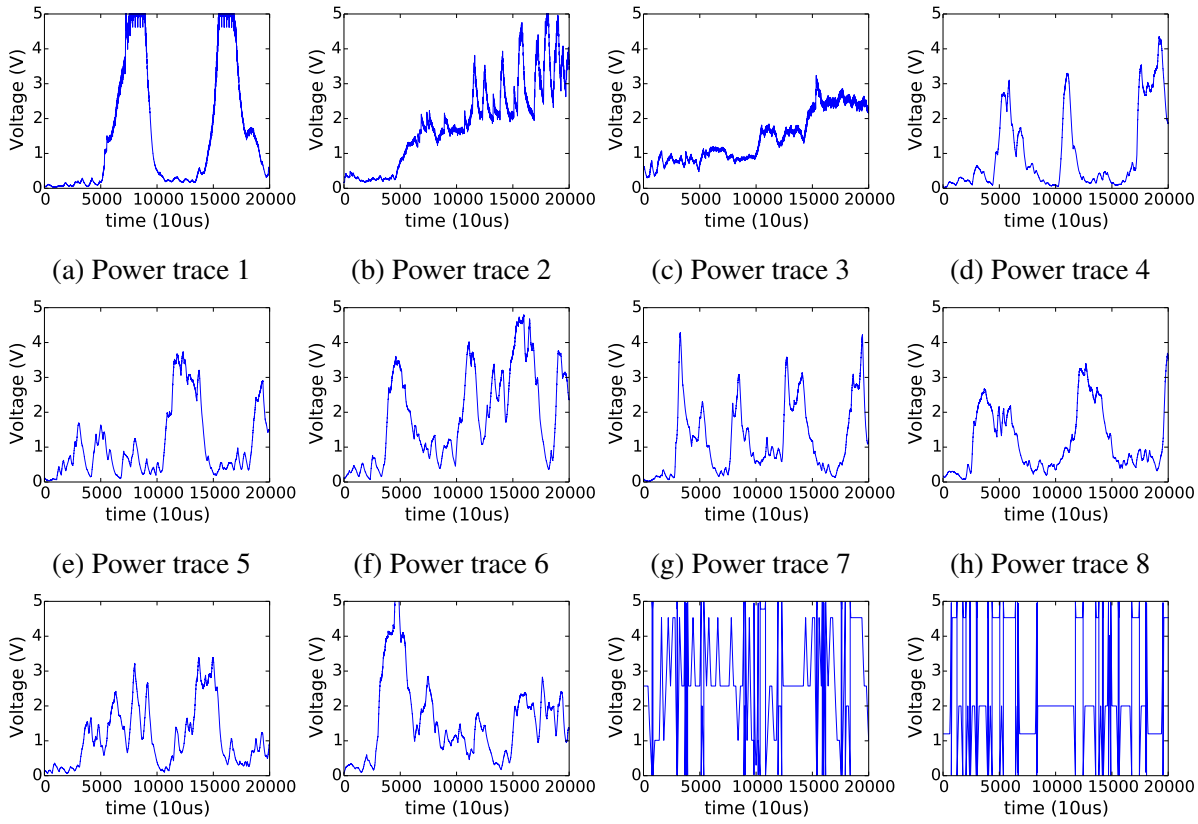
4.6.3 Sensitivity Analysis

Experimental Setting: Rather than placing the RF transmitter in the same position (50cm away from the system), which is conducted by prior works [2], [6], [7], [12], [140]–[142] but considered to be unrealistic, we performed additional experiments to analyze the performance of PFI+RockClimb compared to PFI-only and Chinchilla with the same 11 benchmarks in various energy harvesting situations including an outage-free case and many other unpredictable power failure cases as shown in Figure 4.9. Each power trace in the table causes a different power outage pattern. We found that the trace 10 caused only one power outage while the trace 12 incurred 12 power outages in one second as shown in Table 4.3. With these different power failure patterns, we will discuss how much performance improvement PFI+RockClimb can achieve by comparing it to prior works for each pattern.

Table 4.3. The number of power failures per second in traces.

Trace	1	2	3	4	5	6	7	8	9	10	11	12
# of P.F (s)	2	3	2	4	5	4	8	4	3	1	9	12

For realistic experiments, we developed a power generator board with MSP430FR5969 to generate various power inputs with the power traces collected by prior works [25], [95] in real energy



(a) Power trace 1 (b) Power trace 2 (c) Power trace 3 (d) Power trace 4

(e) Power trace 5 (f) Power trace 6 (g) Power trace 7 (h) Power trace 8

(i) Power trace 9 (j) Power trace 10 (k) Power trace 11 (l) Power trace 12

Figure 4.9. Energy harvesting trace; the plots in this table show voltage input fluctuations to MCU during 12 different movements from an RF energy harvesting reader [25], [95]

harvesting settings; the power generator provides supply voltage to our target energy harvesting system board through GPIO pins according to the traces.

Overall Performance Trend: Figure 4.10 shows the performance results on average in all different situations, e.g., no power failure, and various power patterns from trace 1 to 12 shown in Figure 4.9, including the Powercast RF transmitter (discussed in Section 4.6.1). The Y-axis is the normalized execution time to PFI+RockClimb as a baseline. In summary, PFI+RockClimb is always faster than others, achieving 1.9x and 2.7x average speedups over Chinchilla and PFI-only, respectively. PFI-only shows the worst performance across all traces. Due to the logging and re-execution overheads, it ends up with lower performance compared to others. Overall, Chinchilla shows relatively better performance overhead than PFI-only, since Chinchilla can skip checkpointing at some points depending on energy source condition.

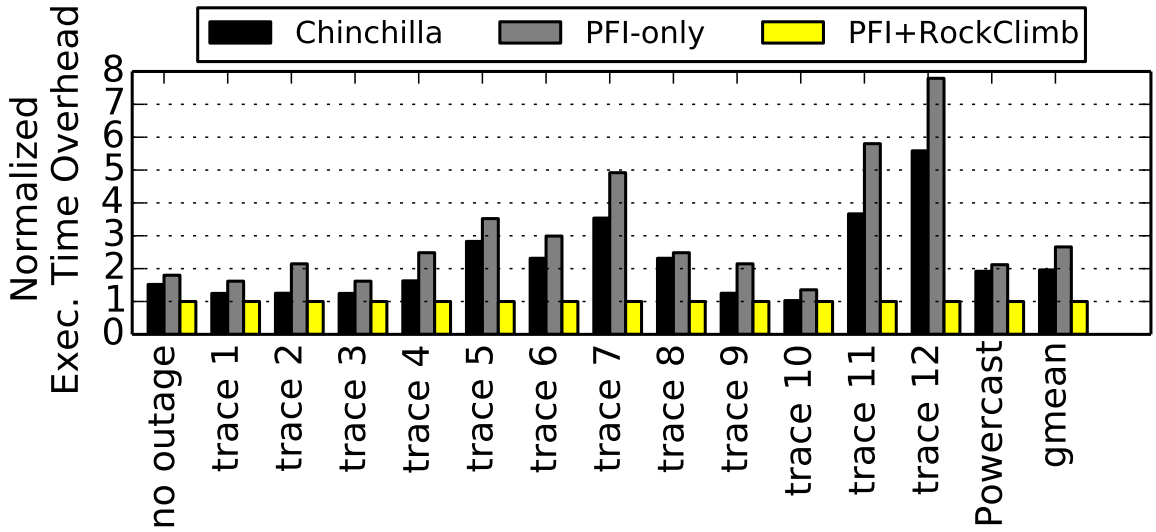


Figure 4.10. Performance results in various situations; Y-axis shows the normalized execution time to the baseline.

No Power Failure: When there is no power failure, Chinchilla and PFI-only cause about 1.5x and 1.9x slowdowns, respectively, compared to PFI+RockClimb as shown in Figure 4.10. Here, Chinchilla recognizes that energy source condition is good, and thus it skips most of checkpoint stores. Nevertheless, since it cannot avoid memory logging overhead, it should check log entries at every memory update and flush the logged data at some points, which causes a significant performance overhead. This is why Chinchilla underperforms PFI+RockClimb even in the power-failure-free case.

Various Power Failure Patterns: When there is frequent power failure, both Chinchilla and PFI-only cause memory logging and re-execution overheads unlike PFI+RockClimb. In particular, with trace 12, Chinchilla and PFI-only show 5.7x and 7.7x slowdowns, respectively, compared to PFI+RockClimb. The reason is that the trace causes the most frequent power outages among all traces, i.e., generating 15x more outages than trace 10. This implies that when there are frequent power outages, the both prior schemes likely cause a much higher performance overhead. Here, Chinchilla cannot skip register checkpointing due to the frequent power outages—since it recognizes that the harvesting condition is poor. On the other hand, for trace 10, Chinchilla is comparable to PFI+RockClimb. That is because the trace causes the smallest number of power outages and happens to cause power failure at a right time, i.e., right after register checkpointing,

minimizing the waste of Chinchilla's rollback recovery, Thus, PFI+RockClimb is only 5% faster than Chinchilla for trace 10.

4.7 Summary

We introduce power failure immunity (PFI) that ensures each code region can fail at most once, thus minimizing the re-executions of power-interrupted regions. In the virtue of PFI, this work presents RockClimb, a rollback-free intermittent computation scheme, ensuring that PFI-enforced regions never fail. To improve the performance further, this work proposes distributed checkpointing, a new compiler optimization that eliminates unnecessary register checkpoints without compromising the recoverability. Consequently, PFI+RockClimb achieves high-performance intermittent computation.

5. COSPEC: COMPILER-DIRECTED SPECULATIVE INTERMITTENT COMPUTATION

Researchers introduce hardware-based recovery solutions for EHS devices. However, the hardware-based solutions require non-trivial hardware modifications, e.g., a voltage monitor, nonvolatile flip-flops/scratchpad, dependence tracking modules, etc., thereby causing significant area/power/-manufacturing costs (PS2). For low-cost yet high-performance intermittent computation, in this chapter, we introduce CoSpec, a new architecture/compiler co-design scheme that works for commodity processors used in energy-harvesting systems (TS2). Our experiments on a set of real energy harvesting traces with frequent outages demonstrate that CoSpec outperforms the state-of-the-art scheme by 1.8∼3X on average.

5.1 Introduction

Energy harvesting systems continue to grow at a rapid pace due to their batteryless nature. However, since ambient energy source is unreliable, the systems suffer frequent power failure. To address the challenge, they use a small capacitor as an energy buffer and intermittently compute only when sufficient energy is secured in the capacitor; when it is depleted, the systems die. This is so-called *intermittent computation*. With the intermittent nature in mind, researchers adopt a low-power in-order processor with byte-addressable nonvolatile memory (NVM) as main memory and offer a crash consistency mechanism to checkpoint necessary data and restore them across power outages.

For the crash consistency, prior works introduce a nonvolatile processor (NVP) [21] that checkpoints volatile registers to nonvolatile flip-flops (NVFFs)—when it is about to be interrupted by power failure—and restores the checkpointed registers from NVFFs in the wake of power failure. Since the NVP restarts exactly at the power interruption point, program states in both NVM and NVFF remain the same across power failure, thereby achieving crash consistency.

Unfortunately, the NVP requires non-trivial hardware modifications. To checkpoint the entire register file right before power failure, they require not only the NVFFs but also a voltage monitor, checkpoint/controller logic, and additional capacitors for the monitor itself. Even worse, the

voltage monitor has stability issues such as excessive leakage or capacitor aging effects leading to reduced capacitance and voltage detection delay with unexpected cold-start glitch. To mitigate the issues, the prior NVP works aggressively increase the voltage threshold of the system wake-up/backup, which is energy-inefficient due to the inability to make forward progress unless such a high voltage is secured to wake up the system.

With that in mind, we propose CoSpec, an architecture/compiler co-design scheme that can realize low-cost yet performant intermittent computation for commodity in-order processors used in energy harvesting systems. To realize crash consistency without the voltage monitor based checkpointing, CoSpec leverages speculation assuming that power failure would not occur and thus hold all committed stores in a store buffer (SB)—as if they were speculative—in case of mispeculation; we call this **power failure speculation**. CoSpec compiler first partitions a given program into a series of recoverable regions with the SB size in mind, so that no region overflows the SB during the region execution. When the program control reaches the end of each region, the speculation turns out to be successful; therefore, CoSpec releases all the stores of the region, which have been buffered in the SB, to NVM.

If power failure occurs during the execution of a region, all its stores buffered in the SB disappear because it is volatile. The implication is that such mispeculated stores—left behind power failure—cannot affect any program state in NVM at all. Consequently, the interrupted region can be restarted with consistent program states in the wake of power failure.

While CoSpec provides crash consistency, the region-based speculation window causes pipeline stalls at the end of each region due to the SB release. Since it consists of NVM writes that are the most time-consuming instruction, the stalls are rather long leading to a significant performance overhead. To hide the long NVM write latency of the SB release, CoSpec overlaps the SB release of the current region with the speculative execution of the next region. Such instruction level parallelism (ILP) gives an illusion of out-of-order execution on top of the in-order processor, achieving high-performance intermittent computation.

Our contributions can be summarized as follows:

- Unlike prior works, CoSpec does not require any expensive hardware modifications. CoSpec's intelligent compiler-architecture interaction provides commodity microarchitecture with crash consistency, achieving truly recoverable intermittent computation at a low cost.

- CoSpec achieves high performance intermittent computation. The proposed ILP techniques allows CoSpec to effectively hide the long latency of NVM writes in a program. Overall, CoSpec outperforms the state-of-the-art nonvolatile processor by 11% on average and up to 26% when there is no power outage.

- In the context of voltage-monitor-free energy harvesting systems, CoSpec can decrease the wake-up voltage by 1.5∼3X compared to the state-of-the-art work [21], [143], which leads to a much higher energy efficiency. The experimentation with frequent power outages demonstrates that CoSpec consumes 2∼3X less energy than the state-of-the-art work.

5.2 Background and Challenges

For the crash consistency, prior works including nonvolatile processors (NVP) approaches rely on voltage monitor based checkpoint schemes [3], [14]–[18], [144]. They checkpoint volatile registers—when the voltage monitoring system detects the voltage drop below a defined threshold—by using the buffered energy in the capacitor. In addition to the voltage monitor, the schemes require non-trivial hardware modifications such as nonvolatile flip-flops, that must be laid out next to volatile flip-flops for fast backup/restoration, special hardware checkpoint/controller logic, and additional capacitors for the voltage monitor.

Even worse, the voltage monitor may cause stability issues such as excessive leakage or crack of the capacitors leading not only to reduced capacitance [19], [20] but also to voltage detection delay with unexpected cold-start glitch [21]. To mitigate the issues, existing works aggressively increase the voltage threshold of the system wake-up/backup [1]. Consequently, they waste hard-won energy with making no forward progress until such a high voltage is secured to wake up the system for sure.

[1]↑Without the voltage monitor, the wake-up voltage can be set between 1∼1.8V [14], [15], [138], which is about 1.5∼3X lower than that of the state-of-the-art work [21], [145].

With that in mind, Hicks [5] proposes a voltage monitor free crash consistency scheme called Clank by implementing idempotent processing [4] in hardware. In detail, Clank monitors all memory accesses (load/store) at run time with several memory buffers such as write-back, read-first, write-first, and address prefix buffers. By sweeping read-first and write-first buffers, Clank keeps track of antidependent load-store pairs that make it impossible to perform idempotent processing and thus lead to memory inconsistency in the wake of power failure.

In particular, once an antidependent store is detected, Clank holds it in the write-back buffer; non-antidependent stores are directly merged into NVM. If any of the buffers is about to overflow and unable to accommodate any further memory instruction (address), Clank alerts the processor to checkpoint all its registers, flushes the write-back buffer to nonvolatile scratchpad emptying out other buffers as well, and copies the flushed data eventually to nonvolatile main memory. Note that since it holds the antidependent store in the write-back buffer, Clank requires every load to check the write-back buffer first in case of the store-to-load forwarding.

Unfortunately, Clank suffers from two significant problems that prohibit its adoption. First, although Clank takes advantage of nonvolatile scratchpad—much faster than NVM—for performance reason, there is no current technology to realize nonvolatile yet fast SRAM in reality. Clank may leverage NVSRAM, a 3D stacking based hybrid design of SRAM and NVM [146], which copies SRAM data to the slow nonvolatile part right before power failure. However, NVSRAM also requires the voltage monitor and the necessary checkpointing/controller logics, rendering Clank vulnerable to the same voltage monitor issues. Second, Clank may involve frequent checkpoints due to overflows in its memory buffers, thus degrading the performance significantly.

While Clank proposes to increase the size of the buffers for less overflows, it presents another—potentially more serious—problem in terms of the resulting hardware and energy costs. To a large extent, enlarging the buffers puts significant pressure on the design of CAM (content addressable memory) structure for Clank's associative searches of the buffers; in fact, the size of load/store queues has scarcely increased at all in the last decade for the same reason. Apart from the additional power consumption on the larger buffers, their wire delays might lead to significant energy consumption, possibly making Clank inappropriate for energy-harvesting systems.

With the reasonable size of the buffers, the performance overhead of Clank can be more than 20% even with the unrealistic assumption of the nonvolatile scratchpad [5]. With the deficiencies

of all above prior works in mind, we seek to develop a practical crash consistency solution that works for commodity processors without a significant run-time overhead.

5.3 Overview

CoSpec is a low-cost architecture/compiler co-design scheme that enables reliable crash consistency without significant energy and performance overheads. This section first presents the basic design of CoSpec: (1) hardware design, (2) compiler support, and (3) architecture/compiler co-design. The optimization techniques of CoSpec are deferred to Section 5.4.

5.3.1 CoSpec Hardware Design

The design philosophy of CoSpec is to leave the commodity microcontroller (MCU) architecture [38], [147], [148]—used in energy harvesting systems—almost as is and enable high performance intermittent computation without expensive hardware modifications.

Store Buffer for Power Failure Speculation: Store buffer has been adopted for other commodity in-order MCUs [147], e.g., ARMv8-A core implementations [148], mainly to handle mispeculation such as branch misprediction [2]. To achieve lightweight crash consistency, CoSpec proposes to exploit such a store buffer (SB) for a different type of speculation.

The difference is that CoSpec uses a region-level speculation window, guessing whether each region is likely to finish without interruption due to power failure. In other words, CoSpec leverages the store buffer (SB) to hold committed stores of each recoverable code region during its execution—since they are treated as speculative—until the program control reaches the end of the region (i.e., the region boundary) where the speculation turns out to be successful and thus all the buffered stores are released.

Note that this speculation approach never allows the stores of any regions being interrupted by power failure to be written to primary main memory (NVM). If power failure occurs, all buffered stores in the SB disappear because it is volatile. It is therefore impossible for the mis-speculated stores to affect NVM. Consequently, the interrupted region can be restarted with consistent pro-

[2]↑Currently, the processors used in energy-harvesting systems have no branch predictor as in MSP430 MCUs, NVPs (nonvolatile processors), Clank, and CoSpec.

gram states in the wake of power failure. The takeaway is that speculative stores cannot be released to NVM until they become non-speculative, i.e., their region finishes without power failure. As a result, CoSpec can completely eliminate memory inconsistency without adding multiple non-trivial microarchitectural components required by nonvolatile processors and Clank.

It is important to note that CoSpec splits the store buffer into two parts to enable instruction level parallelism as will be shown in Section 5.4.1. During the program execution, any two consecutive recoverable code regions exclusively occupy one of the two parts in the SB. That is, each statically partitioned code region commits its stores to a different part of the *SB* at run time. When the program control reaches each region boundary, CoSpec drains to NVM only the stores in the part of the SB which is used by the region being finished.

As the major challenge in achieving correct crash consistency, CoSpec should maintain failure atomicity of the *SB* draining; otherwise, any partial draining may result in the memory inconsistency problem [4], [5], [21]. To overcome the challenge, CoSpec leverages a 2-phase SB release mechanism; CoSpec first drains the committed stores from the *SB* to a proxy buffer in NVM, and then copies the drained results from the buffer to the primary main memory area in NVM. With the help of the 2-phase SB release, either the buffer or the main memory can always remain intact no matter when power is cut off. This will be discussed in Section for more details 5.3.3.

5.3.2 CoSpec Compiler

To partition the program into such regions, CoSpec compiler first counts the number of stores while traversing the control flow graph (CFG) of the program. When the number of stores hits a threshold, i.e., a half the SB size, CoSpec compiler cuts the current basic block—where the last store is counted—by placing a region boundary. Then, the compiler analyzes the live-out registers of the resulting region and inserts a checkpoint instruction to save them into a designated register file (RF) checkpoint storage in NVM. In particular, a PC register is saved at the end of each region—which serves as a recovery point in the wake of power failure—so that the forthcoming power failure will be recovered by restarting the next region.

Note that all inserted checkpoint instructions are normal store instructions. Thus, according to CoSpec's power failure speculation, they are first committed to the SB and then drained to

NVM provided the speculation turns out to be successful. Indeed, the region formation is a tricky problem due to the circular dependence during the partitioning process. That is, the live-out register checkpointing essentially adds store instructions to a region, and of course the number of (added) stores determines the region boundary, which in turn affects the live-out registers provided the region boundary changes.

Region Formation: To conduct the region formation, we leverage the algorithm used in our own prior work [92]. In the following, we describe the high-level idea; for more details, readers are referred to the work [92].

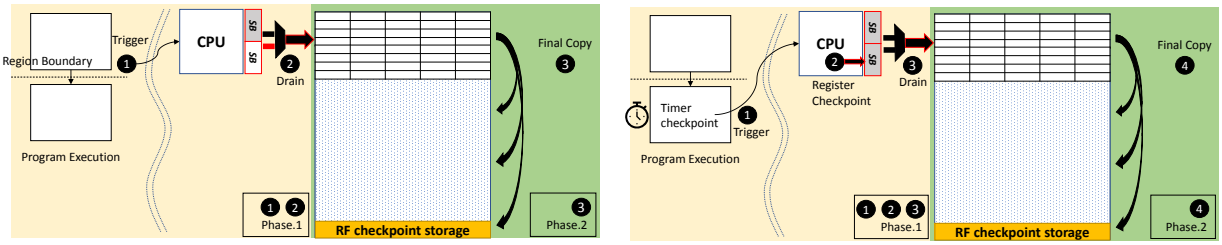
CoSpec first partitions an input program into common program structures such as calls and loops. For this purpose, CoSpec places a region boundary at all the entry and exit points of functions. Likewise, a boundary is placed at the beginning of each loop header. Next, CoSpec identifies the basic block that has region boundaries in the middle of it, and splits it into separate basic blocks. This allows the region boundaries to always start at the beginning of basic blocks, which helps the next step to compute the initial checkpoint instructions. After finishing the initial region formation, CoSpec analyzes the regions to place live-out register checkpoints (i.e., store instructions).

Then, CoSpec compiler traverses the CFG in a topological order trying to combine those initial regions into larger regions as much as possible. The region combining can eliminate many checkpoints because the live-out registers of preceding region(s) are often no longer live after being combined with following regions. During the traversal of each control flow path, CoSpec updates the sum of current and incoming basic blocks' stores from the beginning of the latest region boundary along the path. If the sum becomes greater than a half of SB size (threshold) before the next region boundary is reached, CoSpec places a boundary to cut the region. After that, CoSpec compiler analyzes the re-partitioned regions again to insert live-out checkpoints and possibly repeats the re-partitioning process as long as there is a region that has more stores than the threshold.

In this way, it is guaranteed that the each partitioned region has at most as many stores as a half of the SB size, i.e., the threshold. It would be a mistake to take this to mean that all regions have exactly the threshold number of stores; rather many regions could have less stores than the threshold due to the re-partitioning process.

I/O Operations: To the best of our knowledge, to support non-recoverable operation such as I/O operation has remained as the open problem. That being said, since CoSpec compiler places a

region boundary at function calls, the function that implements I/O operations is treated as a separate region—though it cannot be recovered due to the I/O operation. We believe that CoSpec can deal with I/O operations by simply checkpointing necessary status—just before each I/O operation starts—so that the interrupted I/O operation can be restarted in the wake of power failure.



(a) Region-based Checkpoint (normal case) (b) Timer-based Checkpoint (exceptional case)

Figure 5.1. CoSpec's checkpoint protocol for a normal case (a) and an exceptional case such as stagnation (b)

5.3.3 Architecture/Compiler Co-design

2-Phase Store Buffer Release Protocol: To achieve failure-atomic store buffer (SB) release, which is required for safe power failure recovery without memory inconsistency, CoSpec drains SB to NVM using a 2-phase mechanism. When each region is ended, i.e., program control reaches the end of each region boundary, CoSpec first drains the committed stores to a proxy buffer allocated in NVM and in turn moves the drained data from the buffer to the primary main memory in NVM. Figure 5.1 describes how the 2-phase SB release protocol works for (a) a normal case (region based) and (b) an exceptional (watchdog timer based) case. First, in the normal checkpoint case, the system (❶) triggers the SB release when each region boundary is reached during the program execution. Then, CoSpec (❷) drains one part of *SB*—which corresponds to the region being finished—to the proxy buffer in NVM. As soon as the draining is completed, CoSpec (❸) copies all the buffered data to primary main memory locations.

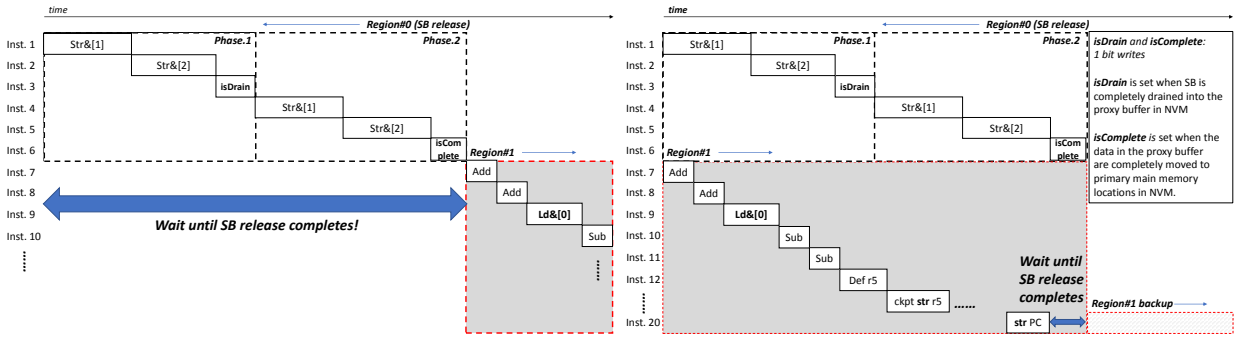
Second, CoSpec also supports an exceptional (watchdog timer based) case. In particular, if a compiler-partitioned region is excessively long, the system might be unable to make forward execution progress because of re-executing the interrupted region again and again across power outages, i.e., *stagnation*. To avoid the stagnation, CoSpec dynamically checkpoints registers to SB

at the expiration of a watchdog timer—which can be adjusted at run time taking into account the dynamic power failure behaviors as will be shown in Section 5.4.2. Figure 5.1(b) shows how the dynamic checkpointing works with the 2-phase SB release. When the watchdog timer (❶) expires, CoSpec (❷) immediately checkpoints (stores) all registers and commits them to the **idle part of SB**–not used by the current region. CoSpec (❸) then drains full *SB* to the proxy buffer in NVM. When two parts of *SB* are completely drained, CoSpec (❹) makes the buffered data moved to the primary NVM locations in the same way as a normal case; note that, the watchdog timer is disabled during the 2-phase SB release process.

The 2-phase SB release mechanism protects both proxy buffer and the primary data in NVM—by managing a check bit for each—against the partial SB draining that may fail to recover from power failure. The first bit *isDrain* is devised for 'Phase 1' release, and it is set when the part of *SB*, which corresponds to the region being ended, is completely drained into the proxy buffer in NVM; in the exceptional case shown in Figure 5.1(b), the bit is set when both parts of SB are drained completely. The second bit *isComplete*—devised for 'Phase 2' release—is set when all the data in the proxy buffer are completely moved to primary main memory locations in NVM. These two check bits help CoSpec to restore correct data in the wake of power failure, and the next section discusses more details about the recovery protocol.

Recovery Protocol: CoSpec provides a safe recovery protocol to address potential memory inconsistency problem across power failures. There are three possible cases of power failure that differ in terms of their failure point in the timeline. First, a power failure can occur during *SB* draining (Phase 1 release), i.e., *isDrain* bit is not set. In this case, CoSpec simply ignores the SB data drained to the proxy buffer in NVM; the SB contents all disappear due to the volatility of the SB. To resume the interrupted region in the wake of the power failure, CoSpec first restores the saved register values including the recovery PC from the RF checkpoint storage in NVM and jumps to the PC. Note that it points to the beginning of the interrupted region at the moment. Although at the end of the region, a compiler-inserted checkpoint successfully saved a new recovery PC that points to the beginning of the next region, it was not written to neither the proxy buffer nor the RF checkpoint storage in NVM because of the power failure occurred during the 'Phase 1' release.

Second, a power failure can occur during the copy from the proxy buffer to the primary main memory in NVM (Phase 2 release). In this case, since the *isDrain* bit has been set, i.e., the recovery



(a) CoSpec without ILP (b) CoSpec with ILP

Figure 5.2. Performance benefit thanks to ILP. DMA is not enabled here, though it can accelerate the 2nd phase of the SB release.

PC checkpoint at the end of the current region was successfully written to the proxy buffer, CoSpec does not rollback to the beginning of the current region. Instead, CoSpec does redo the Phase 2 release, i.e., moving the proxy data to the primary main memory in NVM. Then, as usual, CoSpec restores the saved register values including the recovery PC from the RF checkpoint storage in NVM and jumps to the PC for recovery.

Third, a power failure can occur outside of the 2-phase release, i.e., in the middle of a region, CoSpec recognizes such a case by checking the both bits, i.e., *isDrain* and *isComplete* are set. Here, the recovery process is simpler compared to the above two cases. CoSpec just restores the saved register values including the recovery PC from the RF checkpoint storage in NVM and jumps to the PC that should point to the beginning of the region interrupted by the power failure. The takeaway is that according to the status of the two check bits, CoSpec takes appropriate actions for correct recovery, thereby ensuring truly-recoverable intermittent computation no matter when power is lost and how often it occurs [3].

5.4 Implementation

To avoid potential memory inconsistency, the 2-phase SB release mechanism requires double persistent writes for all stores. Unfortunately, this incurs significant performance overhead consuming hard-won energy—for such expensive NVM writes—that would otherwise could be used for making further forward execution progress. To address the overhead problem, CoSpec optimizes the 2-phase SB release by enabling instruction level parallelism (ILP). That is, CoSpec does not wait until its 2-level SB release is finished; rather it speculatively executes the next region's instructions while the SB release is pending. This section describes the implementation details of such an optimization: (1) how to reliably enable the ILP execution on an in-order processor without memory inconsistency and (2) how to adapt the ILP for the intermittent computation where the frequency of power outages varies.

5.4.1 Instruction Level Parallelism

Enabling ILP execution: CoSpec enables instruction level parallelism to hide long NVM write latency by overlapping them with the next code region execution. Figure 5.2 shows how the instruction level parallelism (ILP) works when two consecutive regions (i.e., Region#0 and Region#1) are executed. Figure 5.2(a) describes a non-ILP case; when *SB* starts its draining to NVM using the 2-phase release mechanism, the system needs to wait until the both phases finish to ensure the SB data is safely written to the primary memory. Since partial SB release can cause memory inconsistency, the power failure recovery might fail. Figure 5.2(b) shows how CoSpec hides such a long latency of NVM writes. By overlapping the NVM writes during entire 2-phase SB release with the speculative execution of the next code region, CoSpec is able to execute more instructions within a given time; as shown in Figure 5.2(b), the ILP approaches executes 13 more instructions than the non-ILP approach that encounters stalls at the instruction #6 due to the 2-phase SB release [4].

[3]↑The recovery protocol can be further optimized by using only one bit. For safe recovery, the check bit is set to 1 when the 'Phase 1 release' is finished, and it is reset to 0 when the 'Phase 2 release' is finished. This implies that the bit is always zero when a new region starts. In the wake of power failure, if the bit is 0, CoSpec simply restarts the interrupted region by restoring registers and jumping to the recovery PC; otherwise, CoSpec first redoes the 'Phase 2 release' and then restarts the region as usual.

[4]↑ Similarly, TSO_ATOMICTY [149] leverages the overlapped region execution for atomic-region based dynamic optimizations. However, that is devised for multi-core out-of-order processors to achieve more thread interleaving. Also the store queue design and the region formation algorithm are different from those of CoSpec.

Note that once the speculative execution of Region#1 is completed, CoSpec should wait until the SB release of the Region#0 is finished rather than executing the next region (Region#2 not shown in the figure). This is necessary for achieving correct crash consistency. The next section shows how CoSpec solve this problem.

Achieving ILP without Breaking Correctness: There are a few challenges CoSpec must overcome to achieve the ILP optimization for correct recovery. First, CoSpec should avoid inserting a store to the *SB* during its draining; otherwise, it may incur the data hazard or race condition on the store buffer. To address this challenge, CoSpec lets each code region alternatively use a different part of *SB*. Recall that CoSpec splits the SB to two parts for exclusive use of any two neighboring code regions. For example, if a current region inserts its stores to one part of *SB*, then the next region inserts its stores to the other part of *SB*. That is, any two consecutive regions exclusively use a different part of *SB* all the time. However, it is still possible to insert a store to the same part of *SB*. For example, if the speculative region execution finishes too fast even before the 2-phase SB release of the previous region is completed, then executing the following region may overwrite data in the part of SB which is pending (being drained) for its 2-phase release. To avoid this problem, CoSpec conservatively waits at the end of the speculative region while the previous region's SB release is pending.

Second, load instructions should read the up-to-date data for correct execution. Suppose that a current load instruction needs to read data, but the required data is placed in the part of SB which is being drained. In this case, the load instruction should be stalled for correctness purpose. To avoid such a delay, both parts of SB must be available for correct execution. With that in mind, CoSpec does not invalidate the *SB* entries being drained until the program control reaches the end of the speculative region, i.e., the one following the prior region whose 2-phase SB release is pending. That way, the load of the speculative region can read any written data of the prior region from its part of the SB—which is being drained—without any stall. Of course, when the load in a region is to read the data written by the same region, its load can be served as usual using the conventional store-to-load forwarding through its own SB.

Discussion: One might argue that adding a SB in a simple in-order pipeline could reduce the core clock frequency as with modern processors where their SB must provide a dependent load with data within L1 hit time to avoid complicating their scheduling logic. However, we believe

that CoSpec is free from this concern thanks to its architecture characteristics. Apart from the use of in-order pipeline and low clock frequency (∼25MHz) in energy harvesting systems, CoSpec does not have a cache (Section 2.1). The implication is that the SB search has only to finish within NVM (i.e., FRAM) access time. Note that this is always doable because each SB entry access is orders-of-magnitude faster than FRAM access latency. Consequently, CoSpec causes neither clock frequency reduction nor scheduling logic complication [5]. In addition, CoSpec can bypass SB searches for the majority of following loads; Section 5.4.3 details the SB bypassing and necessary compiler analysis.

5.4.2 Stagnation-Free Intermittent Computation

CoSpec should address the **stagnation** problem (Section 5.3.3), which would otherwise waste the harvesting energy in vain without making forward execution progress. To ensure the forward progress in the presence of frequent power outages, CoSpec proposes adaptive execution techniques [8], [101], [102] that take into account dynamic power failure behaviors.

The use of ILP optimization and the region-level speculation window may increase power consumption compared to non-modified design, possibly causing more power failures during intermittent computation. In light of this, CoSpec adaptively turns on/off the ILP and adjusts the speculation window according to the power failure patterns in a reactive manner.

When the system suffers from power failures, CoSpec first turns off the ILP execution. Then, if the power failure happens in the same region more than twice, which might be a sign of stagnation, CoSpec turns on the watchdog timer checkpoint. Once the timer is expired, CoSpec checkpoints registers to the store buffer (SB) and performs the 2-phase SB release as shown in Figure 5.1(b). Since the timer is set for it to be expired in the middle of the stagnating region, CoSpec can resume from the timer expiration point in the wake of power failure—rather than jumping back to the beginning of such a long region. If the region still encounter another power failure, CoSpec decreases the watchdog timer to a half of the previous value. This in effect doubles the frequency of the register checkpointing (and the 2-phase SB release) and can be repeated to get out of any long stagnating region across power outages.

[5] ↑Technically, accessing 40 SB entries takes less than 1 cycle [25], [150]

On the other hand, if the system continues to make progress without a power outage in which case CoSpec assumes the system is under a good energy harvesting condition, then it enables ILP and disables the watchdog timer approach. With this simple adaptive execution heuristic, CoSpec can address the stagnation problem and improve the performance by spending more harvested energy for forward execution progress rather than wasting it for the re-executions of stagnating regions.

5.4.3 Energy-Efficient Store Buffer Search

In case of store-to-load forwarding, every load should consult the store buffer (SB). However, this involves expensive CAM (content addressable memory) based associative search in the SB. To address this issue, CoSpec (1) bypasses unnecessary SB searches and (2) designs a cost effective SB search logic.

First, CoSpec compiler statically checks if each load can be may- or must-aliased to stores in the current and previous regions by leveraging alias analysis [120], [151]–[153]. When no alias is found, CoSpec compiler marks the load instruction so that it can bypass the SB. During the program execution, if the processor detects such a special load instruction, it avoids the SB search and directly accesses to primary main memory.

To see the impact of this compiler-directed SB bypass scheme, we conducted measured how many load instructions could avoid SB searches at both compile time and run time. The experimental result demonstrates that a significant number of loads is able to bypass the SB search. As shown in Figure 5.3, at compile time, more than 80% of total load instructions can be marked to bypass the SB search on average by using both basic alias analysis (BasicAA) [120] and advanced alias analysis called SVF (static value-flow analysis [153] [6]). At run time, 98∼99% of dynamic loads turn out to be from the SB search. That is mainly because many non-aliased loads are found in hot loops whereas aliased loads are not.

In particular, the promising results of high SB bypass rates motivate the different design of the SB search mechanism. In other words, CoSpec can afford a sequential search logic rather than the

[6]↑CoSpec compiler could run SVF—which is field- and flow-sensitive—successfully on top of program's region-based control flow sub-graph; while such an advanced analysis is very expensive for whole program analysis, our region-based (per-region) analysis makes it possible to run the SVF for all the benchmarks we tested.

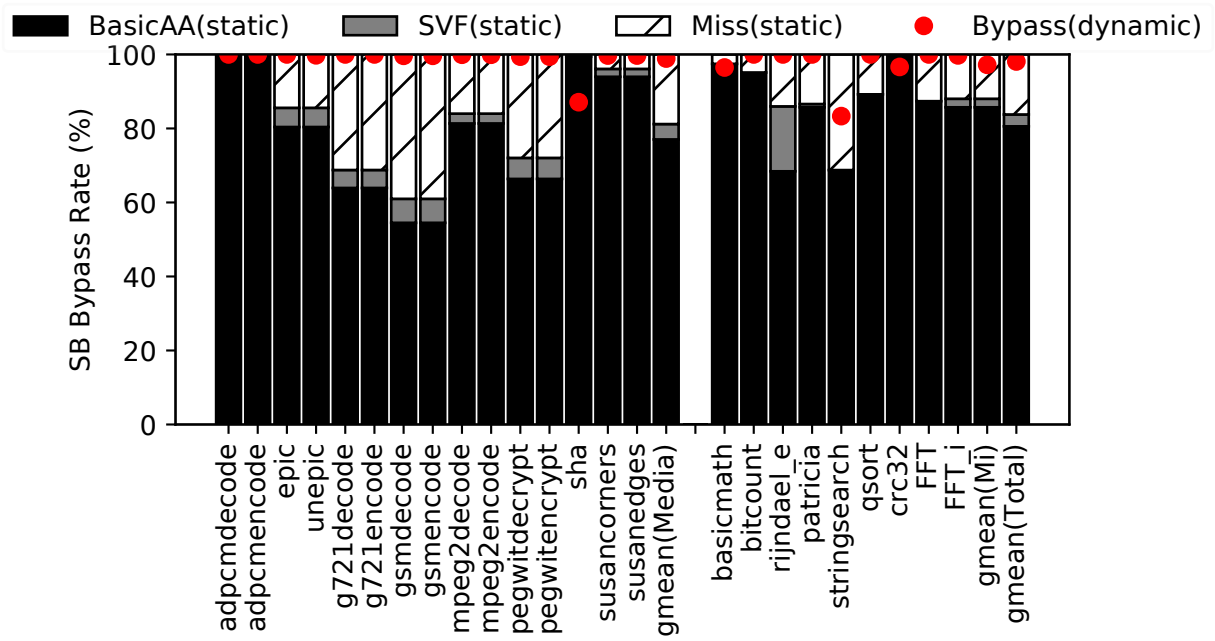


Figure 5.3. Store buffer bypass rates at compile time and run time. Both BasicAA and SVF are static alias analysis.

expensive CAM-based associative search. This gives a freedom to use the SB for energy harvesting system without worrying about the high power consumption required for the CAM search.

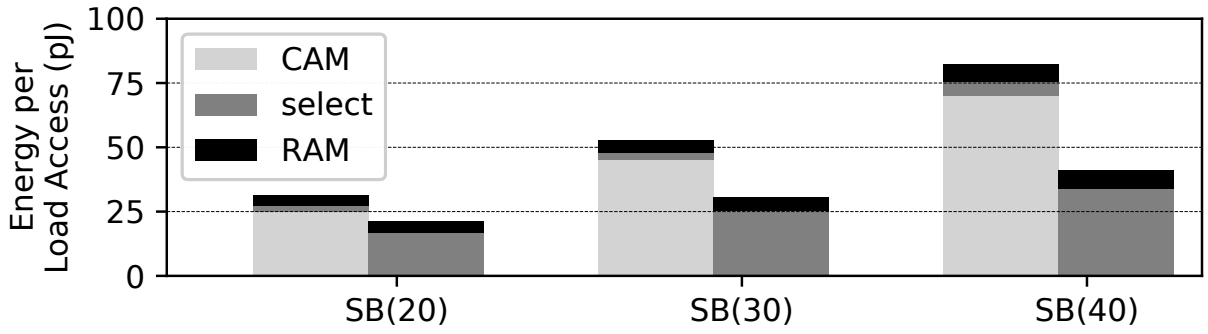


Figure 5.4. Energy consumption breakdown of different SB search schemes. For each SB configuration on x-axis, the first and second bars represent conventional CAM search and CoSpec's sequential search, respectively.

To this end, we estimated the energy consumption and performance of both conventional CAM-based associative search and sequential search by using CACTI [154] with 90nm technology [155]

in the same way as prior work [150]. While the conventional associative SB search is comprised of three components, i.e., CAM, select logic, and RAM (buffer), CoSpec can remove the CAM part thanks to the sequential search. Figure 5.4 describes the energy consumption breakdowns of the CAM-based associative search and the sequential search. When the SB is 40, the energy consumption of the associative SB search is about 2X greater than the sequential search.

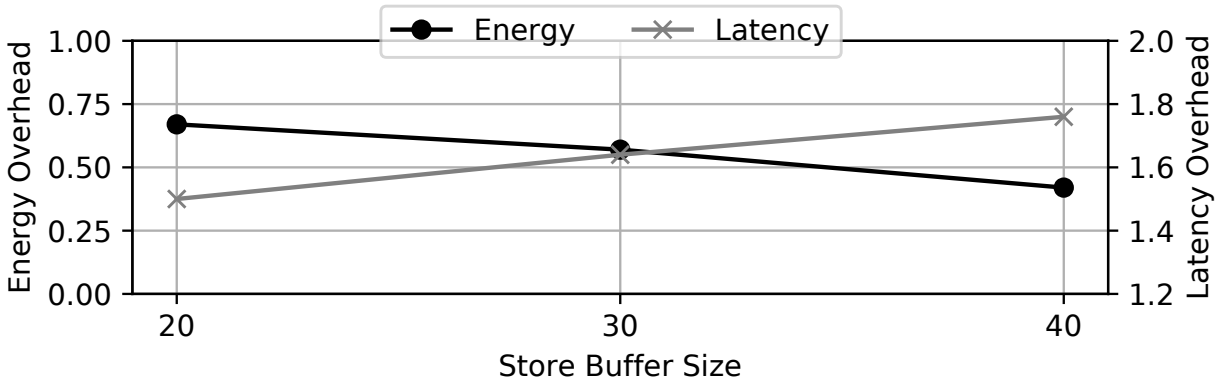


Figure 5.5. Normalized energy/latency overheads of the sequential SB search compared to the CAM based associative search

We also analyzed the latency overhead of CoSpec's sequential search compared to the CAM search. Figure 5.5 shows both the normalized access latency and the energy consumption overhead. The latency overhead is about 1.5∼1.8X when the store buffer size is 20∼40, while the energy consumption reduction is 40∼70%. Section 5.5 evaluates the impact of the both SB search schemes for various benchmark applications.

5.4.4 Direct Memory Access (DMA)

Although ILP execution can hide the long latency of the 2-phase SB release, it does not reduce the latency. To accelerate the SB release, CoSpec can opt for DMA processing available in commodity energy harvesting microcontrollers (MCUs), e.g., MSP430 series. In fact, the DMA engine of MSP430 MCUs [38] can speed up NVM data transfer, i.e., memory-to-memory copy, by ≈4X faster then normal read-write based copy [117]. In light of this, CoSpec can use the DMA to accelerate the second phase (i.e., Phase 2 data copy shown in Figure 5.1) of the 2-phase SB release.

However, care must be taken to perform the DMA processing because every data in the proxy NVM buffer needs to be copied to the corresponding primary main memory locations in a precise manner. Currently, CoSpec uses a single DMA channel multiple times in a row. That is, the number of DMA operations is the same as the number of the proxy buffer entries to be copied. Although a series of DMA copies seem to be not optimized, the DMA processing is still helpful thanks to its 4X faster NVM copy. It is important to note that due to the DMA processing can improve the ILP efficiency as will be shown in Section 5.5.2. That is because the prevention of the SB race condition lets the ILP mechanism conservatively wait at the end of the speculative region for the previous region to complete its 2-phase release (See Section 5.4.1).

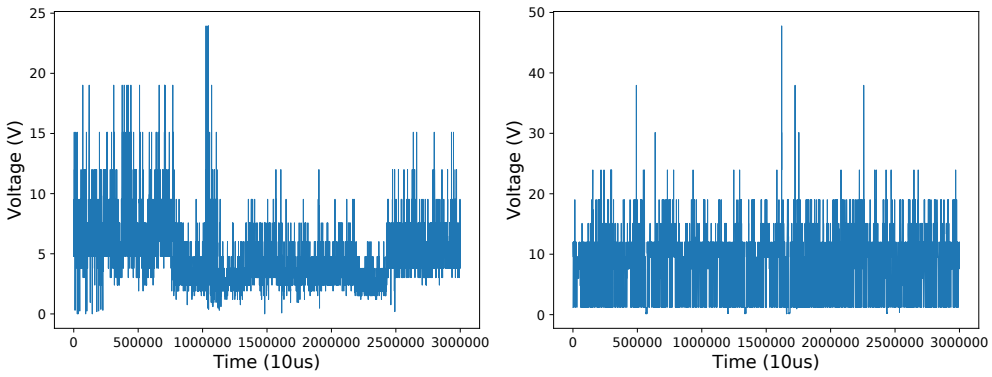
5.5 Evaluation

We implemented CoSpec compiler techniques described in Section 5.3.2 using the LLVM compiler infrastructure [120]. All the experiments were performed on the gem5 simulator [156] with ARM ISA, modeling as in NVP simulator [25]. We compared CoSpec to nonvolatile processor (NVP) [21], i.e., the state-of-the-art NVFF based checkpoint scheme, using the mixture of Mediabench and MiBench applications [122], [123], [157]. They were all compiled with standard -O3 optimization. As a default configuration, CoSpec uses the *SB* size of 40 entries with the sequential search logic (Section 5.4.3) [7]. Table 5.1 describes the hardware specifications of the baseline NVP and CoSpec.

To evaluate CoSpec for harsh environment with frequent power outages, we used two power traces of the NVP simulator which were collected from real RF energy-harvesting systems [25]. Figure 5.6 describes the shape of the two power traces: (a) home and (b) office. In the following, we provide the detailed analyses of CoSpec on (1) hardware cost, (2) execution time with and without power failure, and (3) energy consumption breakdown.

Table 5.1. Simulation configuration

	NVP	**CoSpec**
Capacitor	100nF	100nF/No
Computing Power	100uW/MHz	100uW/MHz
Voltage Monitor(VM)	18uA	No
Store Buffer	No	Yes (Section 5.4.3)
DMA	No	Optional
Von/Voff	3.3/2.8	1.8/1.8
Ckpt/Restore V	3.1/2.9	No/1.8
Write/Read (latency)[8]	120ns/20ns	120ns/20ns
Write/Read (power)	2mW	2mW
Sleep/Wakeup T	46/14us	212/310us[14]
Recovery Point	VM hit	Boundary
ILP	No	Yes



(a) Power trace#1 (Home) (b) Power trace#2 (Office)

Figure 5.6. Energy harvesting voltage traces. Trace#1 and#2 incur ≈20 and ≈400 power outages in every 30 seconds, respectively.

5.5.1 Hardware Cost Analysis

This section analyzes the hardware cost of prior works [5], [21], [159] and highlights the low cost of CoSpec. Table 5.2 provides the major hardware cost comparison. First, NVP [21] requires the voltage monitor, NVFF, and extra energy buffer. The voltage monitor consumes a signifi-

[7]↑Since the target microcontroller [119] has 16 registers, the SB size must be at least two times bigger than the register file size (16) to safely enable the watchdog timer based checkpoint scheme shown in Figure 5.1(b).
[8]We configured the NVM write/read latency based on the commodity design [119] and the state-of-the-art works [25], [158]

Table 5.2. Hardware cost comparison: In the first column, the entries in bold are non-commodity hardware components, i.e., the bold marks represent expensive hardware modifications. Others have already been adopted to commodity hardware designs.

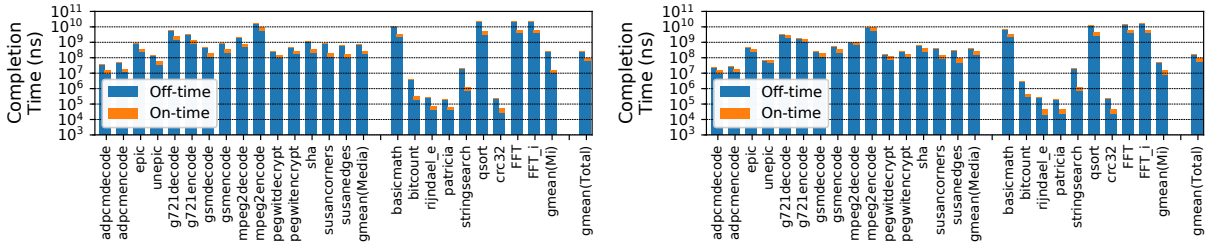
Schemes	NVP [21]	Clank [5]	TCCP [159]	CoSpec
Buffers	No	4 buffers	*SB*	*SB*
DMA	No	No	No	Optional
ISA Change	No	Yes	Yes	Optional
Double Backup	No	Yes	No	Yes
Counter(s)	No	No	**Yes (2)**	No
NV Scratchpad	No	**Yes**	No	No
NVFF	**Yes**	No	**Yes (+NVSB)**	No
Extra Energy Buffer	**Yes**	No	**Yes**	No
Voltage Monitor	**Yes**	No	**Yes**	No
Total Cost	High	High	High	Low

cant amount of energy and occupies a nontrivial portion of die size [5][9]. Also, integrating the NVFF (nonvolatile flip-flops), that must be laid out in close proximity to the volatile flip-flops, in the core microarchitecture is complex and expensive due to the manufacturing cost. Overall, the hardware cost of NVP is high. Second, Clank [5] introduces new hardware components such as nonvolatile scratchpad and idempotence violation (i.e., antidependence) detector with several memory buffers. Since the dependence tracking has to monitor every single load/store and sweep the buffers for CAM based associative searches, it is fair to say that the total cost of Clank is high. Third, TCCP [159], a variant of NVP, builds up an out-of-order processor. As with NVP, TCCP requires the voltage monitor, extra energy buffer, and NVFF. In addition, TCCP introduces a nonvolatile store buffer (NVSB) as well as two threshold counters and their controller logic for varying the checkpoint interval. Given all this, TCCP is another high cost approach.

Finally, CoSpec re-purposes the existing *SB* and introduces its 2-phase release logic. Other than that, CoSpec does not modify core microarchitecture unlike above prior works. Although CoSpec currently assumes a special load instruction for bypassing the SB, this can be done without ISA change. The idea is to (1) set the least significant bit of the aliased load address operand—which must be zero due to the word granularity—and (2) let the pipeline architecture check the bit to reset it and enable the SB bypassing. Although the bit setting instruction must be inserted at

[9]↑ The die area occupied by the commodity voltage monitor is about $0.3mm^2$ [21].

compile time, the overhead will not be significant thanks to the small portion of aliased loads as shown in Figure 5.3. Although CoSpec can opt for a DMA engine, it has already been adopted by commodity in-order processors such as MSP430 series MCUs. Overall, the hardware cost of CoSpec is significantly lower than that of the prior works.



(a) Power trace#1 (Home)

(b) Power trace#2 (Office)

Figure 5.7. Completion time comparison. The 1st/2nd bars of each application represent the times of NVP and CoSpec, respectively.

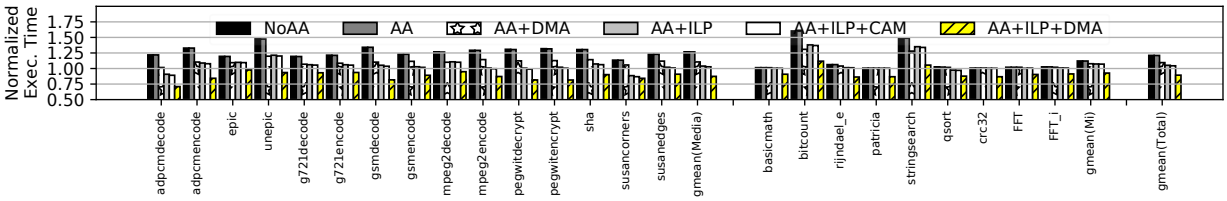


Figure 5.8. Normalized execution time of CoSpec compared to NVP [21]. As a default, CoSpec enables SB bypass, ILP, and DMA support for all other experiments

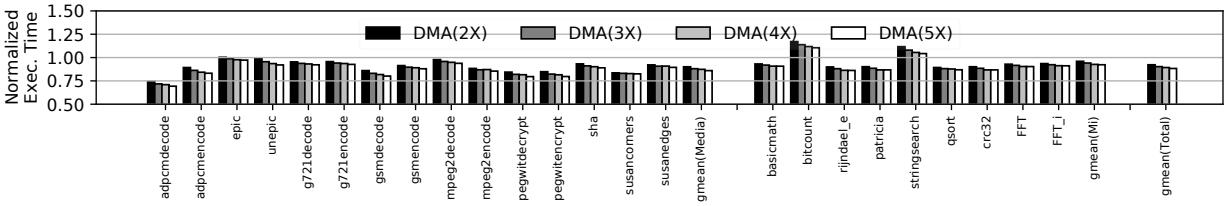


Figure 5.9. Normalized execution time of CoSpec compared to NVP [21] varying DMA speed. DMA(4X) is the default configuration for all other experiments.

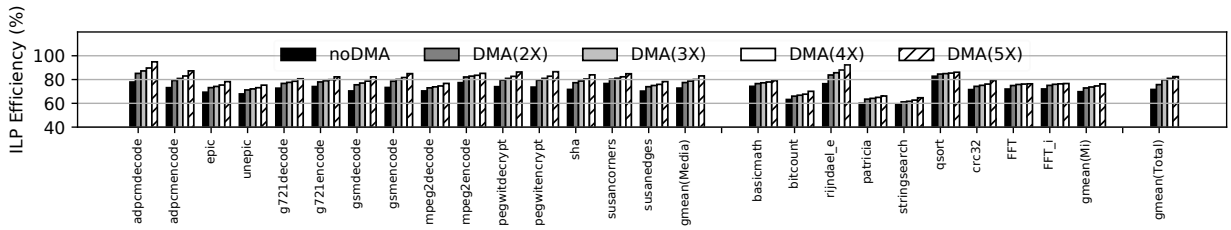


Figure 5.10. ILP Efficiency comparison varying DMA speed. DMA(4X) is the default configuration for all other experiments

5.5.2 Execution Time Analysis with No Outage

To analyze the execution time of CoSpec, we first set the baseline to the state-of-the-art NVP [21] with uninstrumented binaries. We measured the execution time of CoSpec for 24 benchmark applications with 6 configurations.

First, we analyzed the performance impact of the alias analysis based SB bypass by turning it off (NoAA) and on (AA). As shown in Figure 5.8, without the *SB* bypass (NoAA), i.e., the first bar in the figure, CoSpec incurs about 24% execution time overhead due to the region-based power failure speculation overheads such as the 2-phase SB release and the inserted register checkpoints. When the SB bypass is enabled (AA), i.e., the second bar in the figure, the resulting execution time reduction is only marginal. This implies that the SB search is not the main source of the execution time overhead.

Second, we also analyzed the impact of DMA and ILP on the execution time of applications. Recall that DMA is used for fast memory-to-memory copy, and therefore it can only speed up the second phase of the SB release. When both SB bypass and DMA are enabled (AA+DMA), i.e., the third bar in Figure 5.8, CoSpec causes about 10% execution time overhead; as with MSP430 microcontrollers, we set the DMA speed to 4X faster then normal memory copy as default. When both SB bypass and ILP are enabled (AA+ILP), i.e., the fourth bar in the figure, the resulting execution time overhead is only 4∼5% though DMA is not enabled. This confirms that ILP is the main reason for CoSpec's high performance.

Finally, we enabled all the optimizations to see the performance bound of CoSpec. When the best configuration is set (AA+ILP+DMA), i.e., the sixth bar in the figure, CoSpec rather outper-

forms the state-of-the-art NVP by 11% on average. As the next section shows, the use of DMA is able to improve the ILP efficiency. In this way, CoSpec can effectively hide the long latency of NVM writes involved in the 2-phase SB release.

Interestingly, Figure 5.8 shows that CAM search does not make a huge impact on the execution time on average. When the CAM search is enabled with both SB bypass and ILP (AA+ILP+CAM), i.e., the fifth bar in the figure, there is only marginal difference compared to the sequential search with SB bypass and ILP (AA+ILP). That is because 1∼2% of total loads access to the store buffer—as shown in Figure 5.3—thanks to the precise alias analysis of CoSpec's compiler. That is, only a few loads could get the CAM search benefit. Note that all the other bars except for AA+ILP+CAM in Figure 5.8 use the sequential SB search logic.

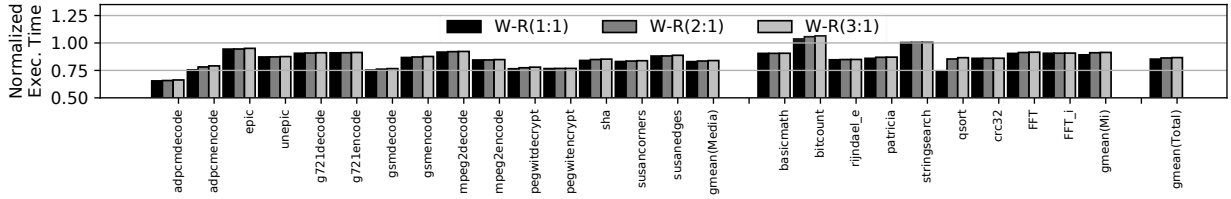
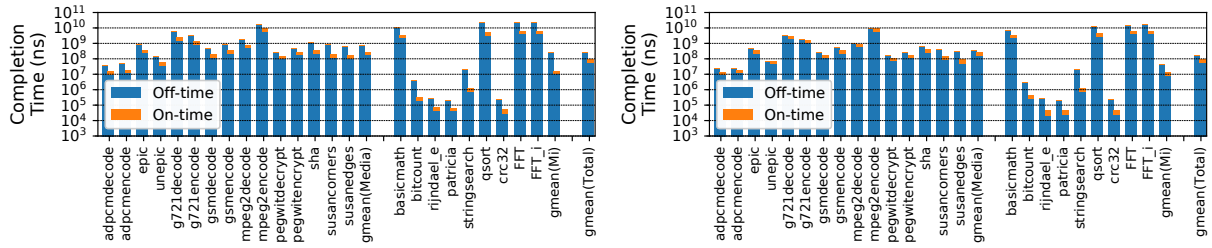


Figure 5.11. Normalized execution time of CoSpec compared to NVP [21] varying the write-to-read ratio of NVM. The ratio, 6:1, is the default configuration for all other experiments.



(a) Power trace#1 (Home)

(b) Power trace#2 (Office)

Figure 5.12. Completion time comparison. The 1st/2nd bars of each application represent the times of NVP and CoSpec, respectively.

Sensitivity Analysis: We explored the performance impact of the DMA and NVM technology with the highly optimized CoSpec (AA+ILP+DMA). First, we varied the DMA speed of data

transfer in NVM, i.e., 2X, 3X, and 5X faster than a normal NVM copy—and then measured the resulting execution times of NVP and CoSpec for the same set of benchmark applications.

Figure 5.9 shows the normalized execution time of CoSpec compared to NVP which is the same baseline used in the prior experiment. To a large extent, CoSpec becomes faster as the DMA speed is increased. When the DMA speed is 5X, CoSpec can achieve ∼6% speedup than the default speed of 4X.

To further analyze the correlation between the DMA speed and ILP execution, we measured the ILP efficiency varying the DMA speed. The ILP efficiency is defined as how much the time taken for the 2-phase SB release of a code region is overlapped with the execution time of the next region. For example, if the SB release time is completely overlapped with the next region execution, the ILP efficiency is 100%. Note that the perfect efficiency is achieved when the region execution time is greater than or equal to the SB release time; either way, the SB release time is fully hidden, the ILP efficiency is 100%. On the other hand, if the next region finishes while the SB release is still pending, the ILP efficiency is decreased. That is because CoSpec must wait—at the end of the next region—for the SB release to finish. As shown in Figure 5.10, 70∼82% of the 2-phase SB release can be overlapped with the next region execution when the DMA speed is 2X∼5X.

Table 5.3. Write-to-read ratios of different NVM technologies.

Memory	FRAM [38]	NVsim [160]	PCM [161], [162]	Re-RAM [158]
Ratio	1:1	2:1	3:1	6:1

Second, we varied the NVM write/read latency ratio, i.e., 1:1, 2:1, and 3:1, assuming different NVM technologies[158], [160]–[162] shown in Table 5.3. Figure 5.11 shows the normalized execution times of CoSpec compared to the same baseline NVP again. On average, CoSpec outperforms the NVP by about 13∼16% when the NVM write/read ratio becomes 3∼1:1.

5.5.3 Execution Time Analysis with Outages

To test the ability to make forward execution progress in the presence of a myriad of power outages, we measured the completion time of benchmark applications using two voltage traces

shown in Figure 5.6; they are collected from a real RF-based energy harvesting system when it is deployed in home (a) and office (a). Figure 5.12 shows the completion time of the baseline NVP (the first bar) and CoSpec (the second bar) with breaking down the time to 2 parts, i.e., power-off-time and power-on-time. As shown in the figure, the system off-time dominates the completion time of both NVP and CoSpec. However, NVP is designed to wake up at 1.5∼3X higher voltage level than the minimum supply voltage of MCUs, due to voltage monitor issues (see Section 5.2). This implies that NVP should stay in a sleep mode for a substantial amount of time without making forward progress. Unlike the NVP, CoSpec can start to operate once the minimum supply voltage is secured, thus achieving further forward progress. Figure 5.12 (a) and (b) highlights that CoSpec outperforms the NVP by 3.0X and 1.8X in the trace#1 (a) and trace#2 (a), respectively.

Interestingly, the NVP makes further forward execution progress in trace#1 than trace#2. As shown in Figure 5.12 (b), NVP's completion time using trace#1 is only 60% of that of using trace#1. Given that trace#2 has relatively less power outages than trace#1, NVP tends to prefer more reliable voltage trace. With that in mind, we expect that CoSpec can outperform the NVP more significantly when the energy source is more unreliable.

5.5.4 Energy Breakdown with Outages

Finally, we analyzed the average energy consumption breakdown across power outages [10]. The total energy consumption can be divided into two parts: one under ILP execution and the other under non-ILP execution. The ILP part is further broken down to successful- and mis-speculation, each of which is comprised of 3 parts: the Phase1/Phase2 of the SB release and the computation. On the other hand, the non-ILP part is two-fold: NoILP and re-execution. NoILP is simply the energy consumption of CoSpec when it executes without ILP excluding that of re-executing any interrupted regions.

Figure 5.13 indicates that the overhead of CoSpec mostly comes from the re-execution cost—i.e., Re-exec in the figure. Although CoSpec enables the ILP and the watchdog-timer-based checkpoint in an adaptive manner according to a dynamic power failure pattern, the adaptation may not help for the first time to make progress (see Section 5.4.2). For example, to avoid stagnation,

[10]↑CoSpec shows similar energy consumption trends in both power traces. On average, the total energy consumption of CoSpec in the presence of power failures using two traces is 2∼3X less than the NVP's.

CoSpec might need to perform multiple times of the adaptation in a reactive manner (involving the sequence of ILP off $->$ watchdog timer on $->$ timer halving). Thus, the re-execution consumes the harvested energy without making actual progress until CoSpec finally gets out of the stagnating region after the multiple adaptations. As shown in Figure 5.13, the re-execution consumes 40% of the total energy on average.

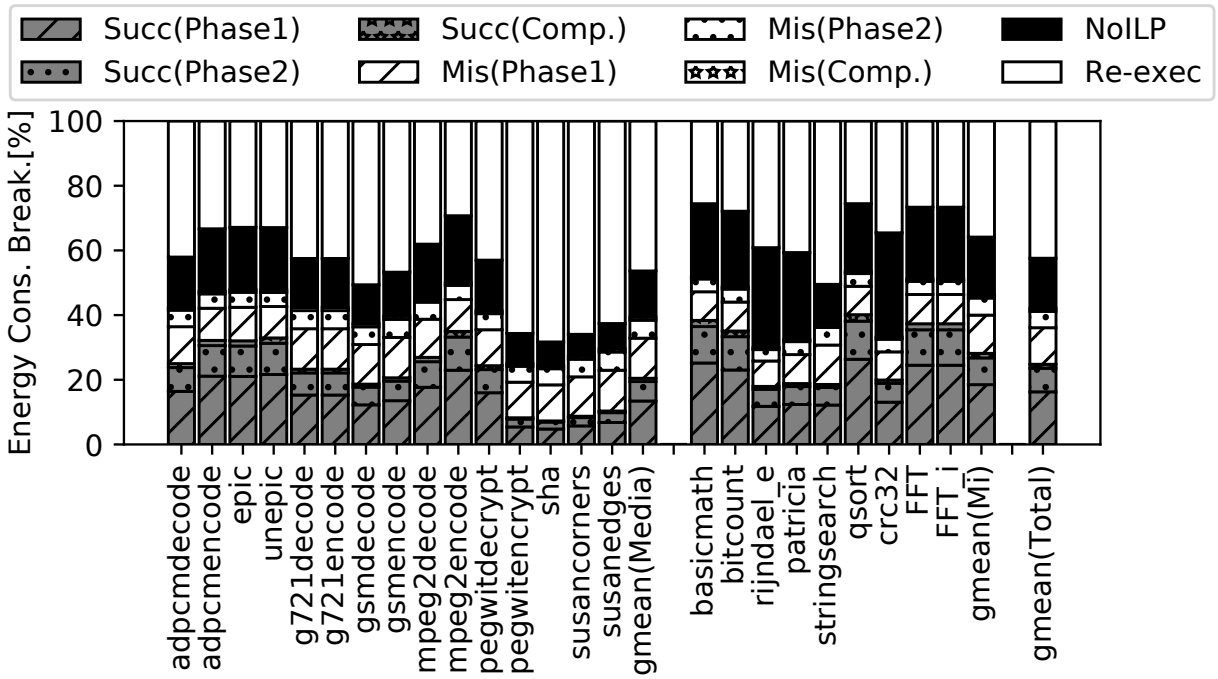


Figure 5.13. Energy consumption breakdown of CoSpec

On average, CoSpec consumes about 40% of its total harvested energy for ILP executions while it does the rest of the energy for non-ILP executions. NoILP consumes 20% of total energy on average due to the adaptation of CoSpec which throttles down its execution to escape (potentially) stagnating regions. In particular, extra NVM writes (i.e., the Phase2 of the SB release during successful- and mis-speculation) account for 18% of the total energy consumption on average. Also, it turns out that the wasted energy of mis-speculated execution (i.e., computation during mis-speculation) is negligible thanks to CoSpec's adaptive execution.

5.6 Summary

We present CoSpec, an architecture/compiler co-designed scheme, that can work for commodity in-order processors, to achieve low-cost yet performant intermittent computation. CoSpec takes advantage of power failure speculation to enable crash consistency without significant hardware and performance overheads. In particular, CoSpec realizes instruction level parallelism on top of the in-order processor pipeline to hide the long latency of nonvolatile memory writes, thereby improving the performance significantly. Our experiments on a real energy harvesting trace with frequent power outages demonstrate that CoSpec outperforms the state-of-the-art nonvolatile processor across a variety of benchmark applications by 3X on average.

6. WRITE-LIGHT CACHE: LIGHTWEIGHT CRASH CONSISTENT CACHE FOR ENERGY HARVESTING SYSTEMS

Energy harvesting system has huge potential to enable battery-less Internet of Things (IoT) services. However, it has been designed without a cache due to the difficulty of crash consistency guarantee, limiting its performance. We introduce Write-Light Cache (WLCache), a specialized cache architecture with a new write policy for energy harvesting systems, that can reduce hardware cost yet improve performance significantly (TS2). WLCache combines benefits of a write-back cache and a write-through cache while avoiding their downsides. Unlike a write-through cache, WLCache does not access a non-volatile main memory (NVM) at every store but it holds dirty cache lines in a cache to exploit locality, saving energy and improving performance. Unlike a write-back cache, WLCache limits the number of dirty lines in a cache. When power is about to be cut off, WLCache flushes the bounded set of dirty lines to NVM in a failure-atomic manner by leveraging a just-in-time (JIT) checkpointing mechanism to achieve crash consistency across power failure. Our experiments demonstrate that WLCache provides significant speedup compared to a non-volatile cache baseline. With no power outage, WLCache achieves about 3.1x speedup. With frequent power outages, WLCache leads to about 2.9x speedup, which is 1.4x faster than the state-of-the-art volatile cache design with non-volatile backup.

6.1 Introduction

Energy harvesting systems [163] offer battery-less computing that is deemed to be the next step in the evolution of IoT. Without a battery, energy harvesting systems can self-power their devices by collecting ambient energy from external sources (*e.g.,* solar power, thermal energy, etc.). They enable various applications such as wearables, sensors, and implantable medical devices [164]–[167] in which a battery-equipped design could be bulky, environment-unfriendly, and cost-inefficient—apart from regular battery replacements.

However, due to unreliable nature of ambient energy sources, energy harvesting systems suffer from frequent power failures. To mitigate the problem, existing energy harvesting systems leverage a small capacitor as an energy buffer and employ a non-volatile processor (NVP) that can instantly

checkpoint/restore all on-chip data, i.e., volatile registers to/from neighboring non-volatile flip-flops at a power failure/recovery point [14]. In addition, the systems use non-volatile memory (NVM), not Flash, as main memory to persist all off-chip data across power failure.[1] Notably, they do not make use of (volatile) cache since its states are lost across power failure causing a crash consistency problem [168]. That is, without a cache, they directly persist data on NVM at the cost of long NVM access latency for every memory operation, resulting in poor performance.

A cache has high potential to significantly improve performance for energy harvesting systems. Given an energy budget, they can make a further forward execution progress by avoiding NVM accesses on cache hits. Unfortunately, leveraging a cache in energy harvesting systems remains a challenge due to the difficulty of crash consistency guarantee. Different cache write policies (*i.e.,* write-though or write-back) have different implications on the crash consistency. A (volatile) write-through cache directly achieves crash consistency as it has no concern about losing volatile cache states across power failures. However, it requires updating both the cache and NVM on each memory store and thus consumes more power, offsetting the caching benefits.

In contrast, a write-back cache does not update NVM until dirty cachelines are evicted to NVM, i.e., write hits do not involve NVM access at all. However, to ensure crash consistency, it requires additional hardware support that can flush all the updated (dirty) cachelines to NVM before impending power failure. For example, prior works introduce a write-back NVSRAM cache [14], [24], [25] that uses a volatile SRAM cache backed with the same size non-volatile (NV) cache counterpart. When power is about to be cut off, the NVSRAM cache triggers just-in-time (JIT) checkpointing that can failure-atomically flush the entire cache [14], [24] or dirty lines [25] to the neighboring NV counterpart. This approach has two downsides: (1) its hardware modification cost is high—no such fabrication has been adopted for production yet; and (2) it requires reserving a large amount of extra energy enough to backup all cache lines (in the worst case all may be dirty). The reserved energy cannot be used for computation, significantly limiting the forward progress and the energy efficiency.

[1]↑ Flash memory requires (9x) higher voltage and incurs orders-of-magnitude slower write latency (1000x) than byte-addressable NVM such as FRAM [3], [117].

We present Write-Light[2] Cache (WLCache), specialized cache architecture with a new write policy for energy harvesting systems. In particular, WLCache builds upon traditional SRAM cache design without requiring non-volatile cache counterpart and combines the benefits of a write-through cache and a write-back cache while avoiding their downsides. Unlike a write-through cache, WLCache does not update data in NVM main memory at every store, but it holds dirty lines in a cache to take advantage of locality, saving energy and improving performance. Unlike a write-back cache, WLCache permits only a limited number of dirty lines in a cache. That way WLCache has only to secure the a small amount of energy—so that the bounded number of dirty lines can be JIT-checkpointed into NVM—without expensive backup/restoration costs.

To achieve this, WLCache tracks a set of dirty cache lines in a separate small hardware queue, called DirtyQueue. Then, WLCache uses two reconfigurable thresholds, named maxline and waterline. On the other hand, the maxline threshold ($\leq$DirtyQueue) defines the maximum number of dirty cache lines in WLCache. When the number of dirty cache lines reaches maxline, WLCache stalls a store instruction until a free slot in DirtyQueue becomes available. Importantly, maxline determines and bounds the amount of energy WLCache needs to reserve to failure-atomically checkpoint dirty cache lines—which is much lower than that of the NVSRAM cache—when power failure is impending.

The waterline threshold ($\leq$ maxline) determines when WLCache writes back dirty cache lines to NVM. When the number of dirty lines exceed waterline, WLCache picks one of dirty lines and asynchronously writes it back to NVM. The persisted cache line remains in the cache with a "clean" state for future references. The asynchronous write-back operation overlaps with the execution of following instructions, realizing instruction-level parallelism (ILP). The gap between maxline and waterline defines the potential ILP opportunity, not available in a write-through cache.

For optimization, WLCache interacts with a run-time system that adaptively adjusts the two thresholds depending on the energy harvesting quality. The run-time system measures a power-on period across power outages and estimates the quality of energy source at each reboot time. When the power-on time increases (*i.e.,* the energy source condition is seemingly good), the run-time raises the waterline/maxline and the JIT checkpointing threshold (V_{backup}) accordingly to hold

[2]↑As light can behave simultaneously as a particle and a wave, WLCache takes advantages of both write-through and write-back writing policies; and WLCache is "light"weight.

more dirty cache lines in WLCache at each reboot time, making it behave more like a write-back cache. On the contrary, when the power-on time decreases (*i.e.,* possible sign of poor energy harvesting), the runtime lowers the waterline/maxline and the V_{backup} threshold to hold a smaller number of dirty lines. That way WLCache starts to act more like a write-through cache and can use hard-won energy more for forward progress—rather than lavishing it on recurring JIT checkpointing of many lines across frequent outages.

Our experiments with 23 applications from Mibench [122] and Mediabench [157] benchmarks highlight that WLCache provides significant speedup compared to the non-volatile cache baseline. With no power outage, WLCache achieves about 3.1x speedup. With frequent power outages, WLCache attains about 2.9x speedup, which is 1.4x faster than the ideal state-of-the-art NVSRAM-based cache design.

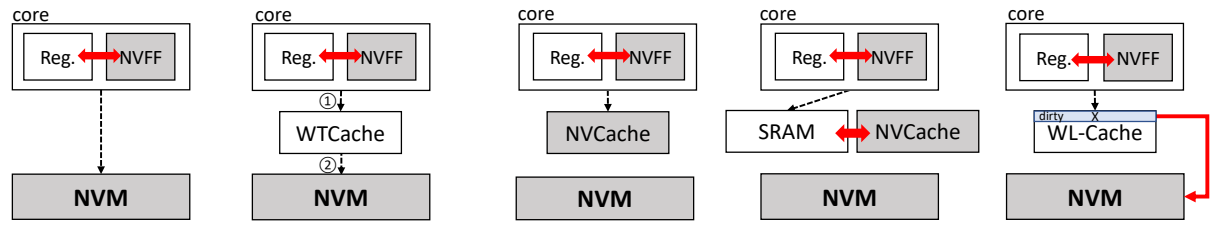
This work makes the following contributions:

- We present WLCache, a new energy-efficient and crash-consistent cache design for energy harvesting systems.
- WLCache introduces a new cache write policy that takes advantages of both write-back's efficiency and write-through's persistence. Like a write-back cache, it cheaply serves subsequent write hits and reduces write traffic to NVM. As a write-through cache, it ensures crash consistency without having an non-volatile counterpart for a volatile cache.
- WLCache adaptively behaves as a write-through or a write-back cache by changing its characteristic back and forth with the quality of energy source in mind.

6.2 Background and Challenges

6.2.1 Cache and Write Policy

A cache allows a system to exploit temporal and spatial locality and thus improves overall performance. Conventional caches employ either write-through or write-back policies, which affect the crash consistency design discussed in the next section. A write-through cache updates both the cache and the main memory at every store. A write-back cache updates only the cache and keeps track of dirty cache lines. It coalesces subsequent write hits on the cache and reduces the write



(a) NVP without cache volatile registers are checkpointed to NVFF in a core at a power failure point.

(b) NVP with WTCache, a write-though volatile cache. A processor updates data in both cache and NVM.

(c) NVP with NVCache, a write-back non-volatile cache. A processor updates data in cache, but it is slow.

(d) NVP with NVSRAMCache. Registers and cache data are checkpointed to NVFF and NVM.

(e) WL-Cache. A processor holds a few dirty lines in the cache and flushes them at a power outage.

Figure 6.1. Design comparison of cache architectures in NVP. Gray boxes are non-volatile while white boxes are volatile. Red arrows represent JIT checkpointing.

traffic to the main memory, achieving higher performance than the write-through design in many cases.

6.2.2 Crash Consistency with a Cache

A cache has a high potential to improve performance for energy harvesting systems. However, all dirty cache lines are lost upon a power failure so the NVM state upon power failure could be inconsistent, causing incorrect program behavior when the program resumed. We now discuss the limitations of existing cache solutions that ensures crash consistency, motivating the proposed approach WLCache.

Volatile Write-through Cache: A traditional SRAM-based write-through cache can be used in energy harvesting systems without modification (Figure 6.1(b)). The write-through policy naturally supports crash consistency by persisting data at every store in a synchronous manner while updating the same data in the cache. However, the requirement of synchronous writes prevents store buffer optimization. The system should pay the long store latency as in the case without a cache. Table 6.1 (second row) summarizes the pros and cons of write-through cache (WTCache). It does not require extra energy for JIT checkpointing, nor additional hardware (beyond a traditional write-through cache). However, as discussed earlier, all stores have to travel to NVM, so its performance improvement is limited.

Non-volatile Write-back Cache: A write-back cache addresses the performance issue of a write-through cache by holding dirty lines in the cache without having synchronous writes. However, it raises the crash consistency problem since the main memory could be outdated upon power

Table 6.1. Hardware complexity and performance comparison in prior cache schemes for energy harvesting systems.

	HW cost	**Energy Buf. Requirement**	**NVM Cache Req.(size)**	**Perf. Improve.**
WTCache	None	No	No	Low
NVCache [22], [23]	Low	No	Yes (Large)	Low
NVSRAM(full) [24]	High	Large	Yes (Large)	High
NVSRAM(ideal) [25]	High+	Large	Yes (Large)	High
NVSRAM(practical) [26], [27]	Medium	Medium	Yes (Medium)	Medium
ReplayCache [54]	None	Small	No	Medium
WLCache	**Low**	**Small**	**No**	**High**

outage. For crash consistency, NVCache [22], [23] is designed as a full non-volatile cache, instead of a traditional SRAM-based volatile one, as illustrated in Figure 6.1 (c). However, NVCache is inevitably slower and requires more energy than a traditional SRAM-based cache. As summarized in Table 6.1 (third row), NVCache does not need to reserve extra energy for JIT checkpointing, but it requires a full non-volatile cache design. The performance impact is limited due to long latency and additional energy consumption. Later we use NVCache as baseline for comparison.

NVSRAMCache: NVSRAMCache [24]–[27] couples a traditional write-back SRAM cache with an NVM counterpart (*e.g.,* ReRAM) as shown in Figure 6.1(d). It achieves crash consistency via JIT checkpointing; it monitors a remaining energy in a capacitor (energy buffer) and copies the SRAM cache states to the NVM counterpart right before a power loss. As listed in Table 6.1, NVSRAMCache can achieve higher performance improvement as it uses write-back policy and absorb write hits (unlike WTCache) and it uses a SRAM-based cache at runtime (unlike NVCache). Additionally, NVSRAMCache can resume from a warm cache (as in NVCache). Three versions (full, ideal, and practical) of NVSRAMCache designs have been proposed and they differ in how to achieve crash consistency and associated hardware complexity.

The original NVSRAMCache (full) [24] checkpoints the "entire" SRAM cache to the NVM counterpart, while the optimized NVSRAMCache (ideal) [25] reduces checkpointing overhead by magically copying "dirty" SRAM cache states only without requiring any additional support. However, note that because all cache lines could be dirty in the worst case, NVSRAMCache (ideal) still needs to reserve the same amount of large energy, enough to JIT checkpoint the entire cache.

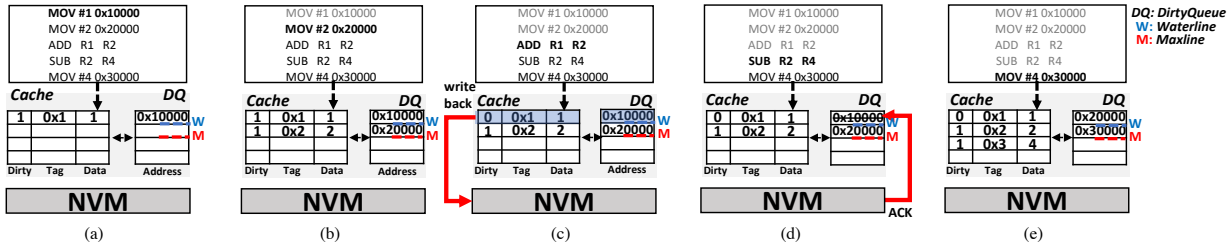


Figure 6.2. Running example of WLCache. WLCache holds dirty cache lines and keeps track of their memory addresses in DirtyQueue (DQ). When the number of dirty lines exceeds waterline (blue dashed line), WLCache asynchronously writes back a dirty line to NVM while a processor executes the next instructions. When the number of dirty lines reaches maxline (red dashed line), WLCache stalls the store instruction, bounding the total number of dirty lines in WLCache.

Moreover, the NVM part is unnecessarily large and mostly wasted because it is used only for checkpointing; therefore, these two versions are impractical. Table 6.1 list the high energy buffer requirement and the HW cost as their main downside.

NVSRAMCache (practical) [26], [27] integrates SRAM and NVM cache designs by maintaining SRAM cache lines and NV cache lines in the same cache set. At runtime, it migrates SRAM cache line to NV lines (if available). Upon power failure, NVSRAMCache (practical) uses JIT checkpointing to move the (remaining) dirty SRAM lines to NV lines, *i.e.,* it should ensure that there are enough available NV lines for JIT checkpointing at all times. Therefore, it writes-back dirty NV lines to NVM main memory at runtime, introducing additional traffic to NVM main memory. Besides, in NVSRAMCache (practical), a data may reside in NV lines, yet accessing NV cache lines are slower and consumes more energy than accessing SRAM lines (as in NVCache). Consequently, the performance improvement of NVSRAMCache (practical) is smaller than that the other designs (Table 6.1).

Though NVSRAMCache (practical) is claimed to be more practical, NVSRAMCache (ideal) can achieve better performance, so we later compare our proposed scheme with NVSRAMCache (ideal). We omit "(ideal)" from now on.

6.3 Design

6.3.1 Overview

The goal of WLCache is to achieve efficient and crash consistent cache design for energy harvesting systems. For performance, WLCache is designed on top of a traditional SRAM-based cache with a write-back policy that holds dirty cache lines and avoids NV main memory accesses on every store. However, unlike a traditional write-back cache, WLCache limits the possible number of dirty lines at a moment, so that it can failure-atomically flush the bounded number of dirty lines to NVM before power failure by leveraging the JIT checkpoiting with a small energy reservoir. For instance, Figure 6.1(e) shows the case in which WLCache allows up to two dirty lines in a cache. By bounding the maximum number of dirty lines, WLCache achieves crash consistency without requiring expensive hardware support such as a large energy buffer or an NVM cache counterpart.

WLCache tracks dirty cache lines with a small hardware component, called DirtyQueue. When a cache line becomes dirty, WLCache inserts its memory address in DirtyQueue. The data remains in the cache (as dirty) and does not become immediately persisted in NVM. When DirtyQueue is about to be full, WLCache selects one of the dirty lines and asynchronously writes it back to NVM—though it may not be in the LRU position[3]. WLCache does not evict the line but leave the data in the cache as clean. When the asynchronous write-back finishes, WLCache removes the entry from DirtyQueue to serve later stores.

To manage DirtyQueue, WLCache employs two configurable thresholds: maxline and waterline. The maxline threshold ($\leq$ DirtyQueue) defines the maximum number of dirty cache lines in WLCache. When the number of dirty cache lines reaches the maxline, WLCache stalls a store instruction until a free slot becomes available. In other words, the maxline determines and bounds the amount of energy that WLCache needs to secure for checkpointing dirty cache lines to NVM upon a power failure. WLCache is more energy-efficient than an alternative NVSRAMCache (ideal) that should reserve a larger amount of energy enough to flush all dirty cache states, *e.g.,* for the

[3]↑Such an eager write-back has originally been devised by Lee *et al* [169] for opportunistically flushing lines when memory bus is idle. However, the prior work was designed for performance so it did not consider either energy efficiency and crash consistency that are essential for energy harvesting systems.

worst case that every line is dirty. Initially, the maxline is set to be some reasonable number (*e.g.,* maxline $= 4$) while considering the energy availability of a given energy buffer.

The waterline threshold ($\leq$ maxline) determines when WLCache starts writing back a dirty cache line to NVM during a program execution. When the number of dirty lines exceeds waterline, WLCache picks a dirty cache line, based on the DirtyQueue replacement policy (Section 6.5.3), and asynchronously writes it back to NVM. Note that WLCache does not evict a dirty line from the cache (which is separately done by a conventional cache replacement policy). Instead, the persisted (written-back) cache line remains in the cache in a "clean" state for future references; the address of the clean cache line is just removed from DirtyQueue only. WLCache exploits instruction level parallelism (ILP) by overlapping the asynchronous write-back operations with the executions of the following instructions.

The gap between maxline and waterline defines the potential ILP opportunity. A high waterline would keep more dirty cachelines in a cache, and should allow WLCache to serve more subsequent write hits without traveling to NVM, saving energy and improving performance. However, at the same time, a higher waterline has a risk to stall the following store instructions in case where WLCache cannot effectively hide the write-back latency (*e.g.,* a code region with frequent/dense stores). By default, waterline is set to be maxline $- 1$. So WLCache cleans (persists) one cache line at a time. The default setting attempts to make DirtyQueue at least one slot available so that a new dirty line can be added in DirtyQueue with no stall.

Conceptually, one can view WLCache with a cache-size maxline (waterline) as a traditional write-back cache; WLCache with a zero maxline (waterline) as a write-through cache. WLCache interacts with the runtime system that reconfigures the maxline and waterline threshold and thus simultaneously behave as a write-back, write-through, and somewhere between them, depending on the quality/stability of power sources. WLCache in effect provides a tuning knob to make the best of two worlds; this will be discussed in Section 6.4.

Figure 6.2 illustrates a running example of WLCache. Supposedly, DirtyQueue's maxline is 2 and waterline is 1 in this example. Assumes that the system has enough energy to execute a given program on the top that consists of three store instructions and two arithmetic instructions in between. The first two store instructions introduces two new dirty cache lines and their addresses (0x10000 and 0x20000) are maintained in DirtyQueue as shown in Figure 6.2 (a) and (b). After

the second store, the number of dirty lines exceeds waterline. In response, WLCache picks one of dirty cache lines and asynchronously makes it persisted and clean (without eviction). If WL-Cache relies on FIFO-based DirtyQueue replacement policy, WLCache writes back the oldest line (0x10000) in DirtyQueue which is mapped to 0x1 tag address in a cache (Figure 6.2 (c)). In the meantime, a program can make a progress and execute the next ADD and SUB instructions, exploiting ILP. When the asynchronous write-back operation is completed (with the ACK message), WLCache removes the corresponding entry from DirtyQueue (Figure 6.2 (d)). With an empty slot in DirtyQueue, the third store instruction can be served without a stall. If the number of dirty lines reaches maxline (which did not happen in this example), WLCache stalls the store instruction and bounds the total number of dirty lines.

6.3.2 Crash Consistency with WLCache

WLCache ensures crash consistency using the following checkpointing and recovery protocols. When a voltage drops below the threshold V_{backup}, the voltage monitor signals the processor to checkpoint volatile registers (as in NVP) and volatile dirty cache lines (in DirtyQueue) to the NVM space. WLCache sets the JIT checkpointing voltage threshold (V_{backup}) high enough to persist the maxline number of cache lines. Compared to NVP (without a volatile cache), WLCache needs to reserve more energy for dirty cache line checkpointing, but WLCache's caching benefits are expected to be much higher (as will be shown in our evaluation (Section 6.6)). During JIT checkpointing, WLCache identifies the dirty cache lines based on the memory addresses stored in DirtyQueue using the existing cache lookup control/data path. Then, WLCache writes them back to the NVM using the existing cache-memory data path; note that the JIT checkpointing is always failure-atomic, thanks to the residue energy at V_{backup} set to be high enough for the completion of the checkpointing.

The recovery protocol after power becomes available again remains simple and the same as that of existing energy harvesting systems. When the capacitor becomes full, energy harvesting systems restore the register states (including instruction pointer) from NVFF for NVP (or from NVM for QuickRecall [3]). A program can safely resume from the exact program point where a power failure occurred.

6.3.3 Discussion

We found that a WTCache with a large write-back buffer can also behave like WLCache. However, the alternative design would be inferior to WL-Cache. for three issues First, the design increases HW cost since the large write-back buffer must be backed with content-addressable memory (CAM) search; one might think of a small buffer to reduce CAM search cost, but it would lead to a worse problem, i.e., frequent NVM writes and pipeline stalls. Second, the design is energy-inefficient since the large buffer requires a significant amount of energy to be secured for crash consistency (failure-atomic write-back before outages). Third, the proposal would suffer performance degradation by extending the critical path of memory access since the write-back buffer must be consulted before accessing memory, i.e., cache miss latency is lengthened.

The key architectural innovation of WLCache is that it decouples the metadata (which cachelines are dirty) from the actual cacheline data. That way WLCache does not increase the critical path of memory access thanks to the lack of metadata (DirtyQueue) lookup, e.g., load miss latency remains the same. This enables WLCache to achieve a lightweight cache along with its adaptation to varying energy harvesting conditions. Not only that, the decoupled design makes it possible for WLCache to realize the DirtyQueue as a volatile structure without compromising the crash consistency guarantee. Thus, WLCache outperforms the write-back buffer proposal in terms of all 3 (cost/energy/performance) aspects.

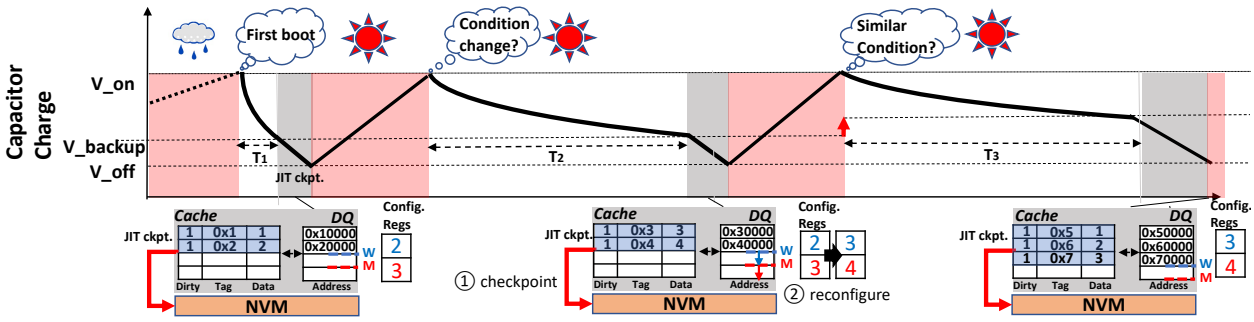


Figure 6.3. An example execution with adaptive maxline, waterline, and V_{backup}. The red and white intervals represents power-off and power-on periods, respectively. The system boots and runs when the charge reaches V_{on}, and starts JIT-checkpointing (gray interval) when it becomes below V_{backup}. T_n represents the power-on time of n-th interval.

6.4 Adaptive Management

To make the best use of write-through and write-back policies while considering the quality of the energy harvesting source, WLCache interacts with a runtime system (a system software) that reconfigures maxline and waterline in DirtyQueue. Energy harvesting systems stores energy in the capacitor so the recharging time (power-off time) directly depends on the quality of energy source. However, the power-off time is hard to measure. Instead, the WLCache runtime system measures a "power-on time" using a watch-dog timer to estimate the quality of energy source. Note that energy harvesting systems continuously collect energy as they execute (during power-on as well). Though every on-interval starts from the same level of energy (at the same V_{on} voltage level of the capacitor), when the harvesting condition is good, the system may run longer.

Based on the observation, WLCache runtime system estimates the energy source quality from the past power-on times, and adjusts the maxline and waterline thresholds at each boot time. Once set, they remain the same during execution (until energy drains). Changing the thresholds while running could be dangerous as energy harvesting systems may not be able to guarantee JIT checkpointing. Adjusting the thresholds before a power failure (for the next run) is also not a good idea as it requires reserving more energy to handle the adaptive logic (in addition to JIT checkpointing).

Algorithm 2 details the default adaptive algorithm that compares the power-on times of the last two intervals (T_{n-2} and T_{n-1}) to determine the maxline and waterline thresholds of the next interval (maxline_n and waterline_n). If the measured power-on time increases significantly (Line 3), it implies that the energy source quality is good, and thus the system adaptively raises maxline and waterline. With higher maxline, WLCache attempts to take more advantage of locality like a write-back cache. In contrast, if the power-on time decreases (Line 8), implying a poor energy source condition, the system lowers maxline and waterline because it is better to avoid a large voltage margin. Otherwise, the two thresholds remain the same. Once the maxline is determined, the runtime system also need to adjust the voltage margin V_{backup} large enough to JIT-checkpoint the maxline number of dirty cache lines at boot time.

Figure 6.3 demonstrates an example execution in which the maxline and waterline are reconfigured. Supposedly, at the beginning of a program execution, the maxline and waterline in DirtyQueue are 3 and 2, respectively. When the capacitor voltage level reaches V_{on}, the system

Algorithm 2 Adaptive maxline and waterline algorithm.

Input: T_{n-2} and T_{n-1}: The power-on times of the last two n-2-th and n-1-th intervals.
Output: maxline$_n$ and waterline$_n$: The maxline and waterline threshold for the n-th interval.
Function *Adaptive (T_{n-2}, T_{n-1}) :*

```
// if the power-on time increases significantly
if T_{n-2} * 2 < T_{n-1} then
    maxline_n = maxline_{n-1} + 1
    waterline_n = waterline_{n-1} + 1
end
// if the power-on time decreases significantly
else if T_{n-2} * 1/2 > T_{n-1} then
    maxline_n = maxline_{n-1} - 1
    waterline_n = waterline_{n-1} - 1
end
// otherwise, the thresholds remain the same
```

end

boots on (the first boot) and runs. When the voltage level becomes below V_{backup}, the system initiates JIT checkpointing of volatile registers and (maxline) dirty cache lines. The system also stores its power-on time for the first interval (T_1). Suppose now the energy source condition becomes much better. The system recharges energy quickly and it makes more forward progress. Upon the third boot, the system detects that the power-on time of the second interval (T_2) becomes much longer than that of the first interval (T_1). It increases the maxline and waterline thresholds (4 and 3, respectively) as well as the voltage margin V_{backup} accordingly. The reconfiguration allows the system to hold more dirty cache lines during the execution of the third interval, providing more opportunity to exploit locality. There is no big change in the power-on times of the second and third intervals (T_2 and T_3), so the thresholds remain the same. When the power-on time drops later, the system may adaptively decrease the thresholds accordingly.

6.5 Hardware and Protocols

Figure 6.4 illustrates WLCache hardware architecture, integrated in NVP. Based on a traditional SRAM-based (volatile) cache, WLCache introduces DirtyQueue and supplementary storages to hold maxline, waterline, and power-on times.

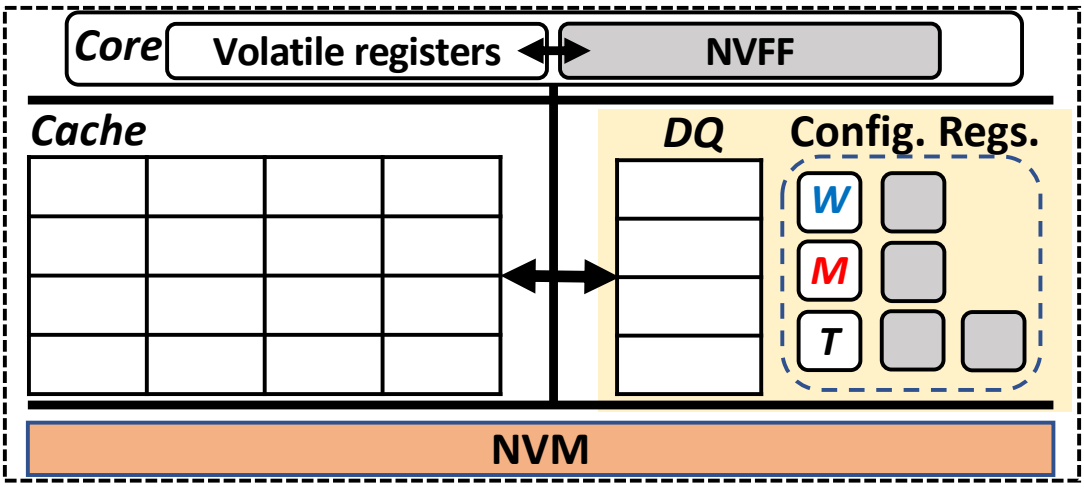


Figure 6.4. Non-Volatile Processor (NVP) with WLCache. Gray boxes represent non-volatile counterparts. Yellow boxes represent WLCache as newly introduced hardware support. Alongside DirtyQueue, WLCache has configuration registers such as waterline (W), maxline (M), and timer (T).

6.5.1 DirtyQueue Insertion Protocol

When a cache line becomes dirty upon a store instruction, WLCache first checks if there is an empty space in DirtyQueue by comparing the maxline and the number of dirty lines (a tail of the circular queue). If available, the corresponding store address is appended in DirtyQueue. Otherwise, the store instruction stalls until an empty slot becomes available. The subsequent store instructions to the same dirty line (*i.e.,* no dirty state change) does not trigger an interaction between a cache and DirtyQueue. Note that, thanks to the waterline constraint, WLCache leaves an at least one space empty (Section 6.4); therefore, the stall rarely occurs.

6.5.2 DirtyQueue Replacement Policy

When the number of total dirty cache lines exceeds the waterline, WLCache selects a dirty line to write back in NVM. We refer to the decision process as a *"DirtyQueue replacement policy"* to differentiate it from a traditional cache replacement policy since WLCache does not evict it but cleans it. WLCache supports following two policies:

- FIFO: The FIFO replacement policy persists (writes back) the oldest dirty cache line. The FIFO policy causes no additional hardware cost except for DirtyQueue that is designed as a circular queue hardware.
- LRU: The LRU replacement policy persists the least recently used (LRU) dirty cache line first. This policy requires additional hardware logic to look up the LRU position of a cache line given a store address in DirtyQueue. If there are multiple same LRU-position cache lines (from different cache sets), then the policy chooses the oldest LRU cache line (as in FIFO).

6.5.3 DirtyQueue Replacement Protocol

After a dirty line is selected according to the above DirtyQueue replacement policy, WLCache asynchronously write-backs the dirty cache line using the following four steps.

1. WLCache marks the cache line clean without eviction.
2. WLCache sends an asynchronous write-back request.
3. WLCache waits for the delivery of an ACK message (acknowledging the completion of the write-back operation). In the meantime, a process executes the subsequent instructions.
4. Upon ACK, WLCache removes the associated slot from DirtyQueue. For FIFO policy, it will be always the head. The LRU-based scheme requires search.

Figure 6.2 (c) illustrates the first and the second steps in which the cache line of Tag 0x1 (Address 0x10000) becomes clean then persisted in NVM. At the moment, the processor executes ADD instruction (exploiting ILP). Figure 6.2 (d) shows the next two steps where the ACK message is delivered and the DQ entry (head, Address 0x10000) is removed.

Removing the associated entry in DirtyQueue last (Step 4) ensures that if a power fails any step before Step 4, WLCache can still safely find the entry in DirtyQueue and perform JIT-checkpointing of the dirty cache line, regardless of whether the write-back request (Step 2) has been fulfilled or not. If completed, WLCache may (redundantly) write back the cache line again, yet there is no correctness issue. If write-back is not done, WLCache will persist the cache line, making the NVM state consistent across a power failure.

Marking the cache line clean first (Step 1) ensures the correctness even when the store instruction to the same cache line is executed while the cache line is being written back asynchronously (Step 3). Suppose we have two stores to the same memory location X, say $W(X) = 1$ and $W(X) = 2$, and some other instructions in between. The first store $W(X) = 1$ makes the cache line $X = 1$ and dirty. Let's say DirtyQueue becomes full (by other store instructions) and the cache line X is selected to be cleaned. The protocol first marks the cache line clean (Step 1) and then start a write-back operation (Step 2). Suppose at the moment, the second store $W(X) = 2$ performs. Now we demonstrate what could go wrong if the cache line is not marked clean (not doing Step 1 first).

If the cache line remains dirty, then the second store will not find a state transition to dirty so it will not update DirtyQueue. The second store will make the cache line $X = 2$ and dirty. Suppose the write-back operation of the first store finishes (NVM now has $X = 1$) and the address X is removed from DirtyQueue (Step 3 and 4). Then, the power is out while the cache has $X = 2$. This is problematic as the cache has $X = 2$ but NVM has $X = 1$ (inconsistent) and DirtyQueue does not have the recent X, losing the cache state $X = 2$. To avoid the problem, WLCache marks the cache line clean first (Step 1), so that the second store will also add the address X in DirtyQueue. As this may happen very rarely, WLCache allows DirtyQueue to temporarily hold redundant X (wasting the slots in DirtyQueue), instead of actively searching the redundant entry in DirtyQueue with additional hardware logic. A redundant entry in DirtyQueue does not affect the correctness as it may only cause redundant write-back operations.

6.5.4 Cache Replacement Policy

WLCache can rely on a traditional LRU cache placement policy that determines which cache line to evict on a cache miss. Yet, the cache replacement policy may conflict with the DirtyQueue replacement policy. When the cache replacement policy decides to evict a dirty cache line whose address is in DirtyQueue, the cache line becomes invalid after a write-back operation (*i.e.,* after it becomes persisted). Thus, it would be ideal to remove the associated entry in DirtyQueue upon an eviction so that DirtyQueue can become more available to other stores. However, this eager cleanup requires searching DirtyQueue to find the entry on each eviction, increasing the latency

and hardware complexity. To avoid search, WLCache instead chooses not to remove it eagerly and allows an outdated slot to reside in DirtyQueue temporarily. When an DirtyQueue entry is selected for replacement or JIT-checkpoining, WLCache can find the cache line invalid (or does not exist) and safely ignore it. Moreover, we empirically found that the traditional LRU cache replacement policy is energy-inefficient for energy harvesting systems. Since it tracks LRU/MRU list at every memory access, it consumes more power and increases more latency than FIFO cache replacement policy; this will be discussed in Section 6.6.6.

6.5.5 DirtyQueue Threshold Management

WLCache introduces a small hardware space (1 byte each) to hold the maxline and waterline thresholds of DirtyQueue. As they have to be alive across a power failure, WLCache introduces NVFF-based backup and performs JIT checkpointing (similar to volatile registers). For adaptive threshold management (Section 6.4), WLCache adds a watchdog timer and two NVFF-based non-volatile storage (2 bytes each) to keep the last two past power-on times. When the power backs on, all these values are restored from NVFF. Figure 6.4 illustrates these additional storages and NVFF backups.

At boot time, given maxline, WLCache also needs to adjust the voltage margin V_{backup} to ensure the failure-atomic JIT checkpoining of (1) registers, (2) (up to maxline) dirty cache lines, and (3) maxline, waterline, and power-on timer values. To reconfigure V_{backup}, WLCache assumes existing hardware support in current commodity microcontroller such as TI-MSP430 [38] that already support different voltage selections. WLCache sets V_{backup} by choosing an associated voltage divider with a reference voltage.

6.6 Evaluation

6.6.1 Experimental Settings

We implemented WLCache on gem5 simulator [156] with ARM ISA, modeling a single core in-order processor. For power failure simulation, we used NVPsim [25] with the same core model. We compared WLCache to (1) non-volatile write-back cache (NVCache-WB) [22], [23], (2) volatile write-through SRAM-based cache (VCache-WT), (3) VCache-WB with ReplayCache

Table 6.2. Simulation configuration.

Processor (1.0GHz, 1 core)	
L1 I/D Cache	8kB, 2-way, 64B block
Cache Latencies (hit/miss)	NVRAM(1.6ns/1.5ns), SRAM(0.3ns/0.1ns)
NVM (ReRAM) Latency (ns)	0.94/7.5/18/15/7.5/150/30 (tCK/tBURST/tRCD/tCL/tWTR/tWR/tXAW)
Energy buffer (capacitor)	1uF
Vbackup/restore	NV(2.9/3.3), NVSRAM(3.2/3.5), WL(2.95∼3.1/3.3∼3.5)
Vmin/max	NV(2.8/3.5), NVSRAM(2.8/3.5), WL(2.8/3.5)

compiler (ReplayCache) [54], and (4) the state-of-the-art NVSRAM cache that uses ReRAM as the SRAM cache backup storage, with a write-back policy (NVSRAM-WB) [21]; we set this as an ideal design that can checkpoint only dirty lines from SRAM to ReRAM at power-off point. We used Mediabench [157] and MiBench [122] compiled with standard -O3 optimization.

As a default configuration, we use volatile L1 instruction and data caches, and their size is 8KB each as with the prior work [24], [54] (see Table 6.2 for details). For WLCache, we use the FIFO for DirtyQueue replacement policy and LRU for cache replacement policy as default—though we varied the policies for sensitivity analysis (Section 6.6.5). Also, we set the DirtyQueue size to 8 and the maxline to 6 as default (*i.e.,* the waterline is 5), then we enable the adaptive threshold management (Section 6.4) to reconfigure maxline and waterline.

To evaluate WLCache in realistic energy harvesting situations, we used the same two power traces, *i.e.,* Trace 1 and Trace 2, of the NVPsim which were collected from real RF sources [25], [54]; Trace 1 and 2 are from home and office, respectively. Trace 2 is relatively less stable than Trace 1.

6.6.2 Hardware Cost

We analyzed the hardware cost of WLCache by using CACTI [154] with 90 nm technology. WLCache requires at most 0.005 mm2 area and 0.0008 nJ (dynamic access). Furthermore, the total leakage power of WLCache (DirtyQueue with a logic) causes only 0.1mW in total, which is only 9% of NV cache leakage [25], [27], [54].

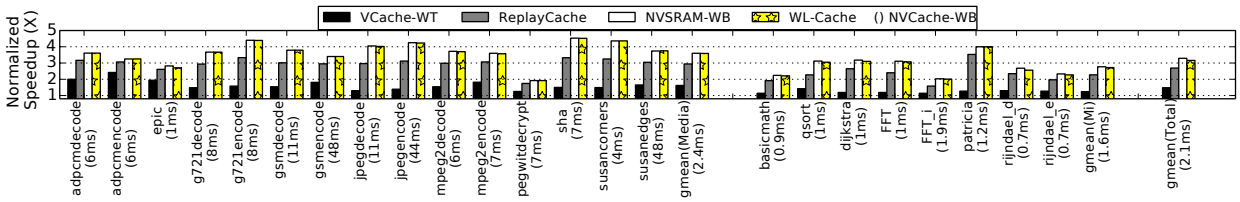


Figure 6.5. Normalized speedup of each cache design compared to NVCache with no power failure. WLCache is 3.1x, 2.18x, and 1.2x faster than NVCache-WB, VCache-WT, and ReplayCache, respectively; NVSRAM-WB is 4% faster than WL-Cache.

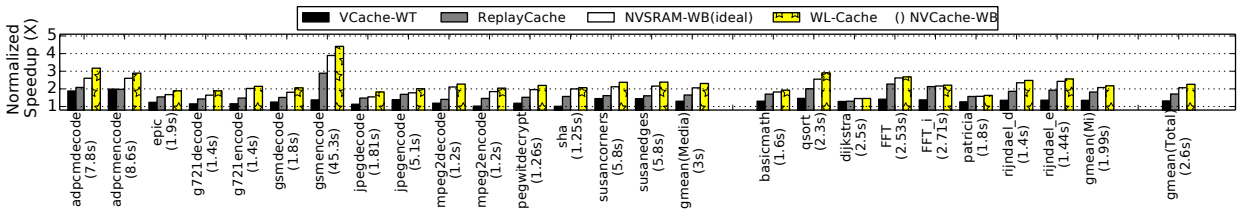


Figure 6.6. Normalized speedup of each cache design compared to NVCache in Power Trace 1. WLCache is 2.25x, 1.71x, 1.32x, and 1.09x faster than NVCache-WB, VCache-WT, ReplayCache, and NVSRAM-WB, respectively.

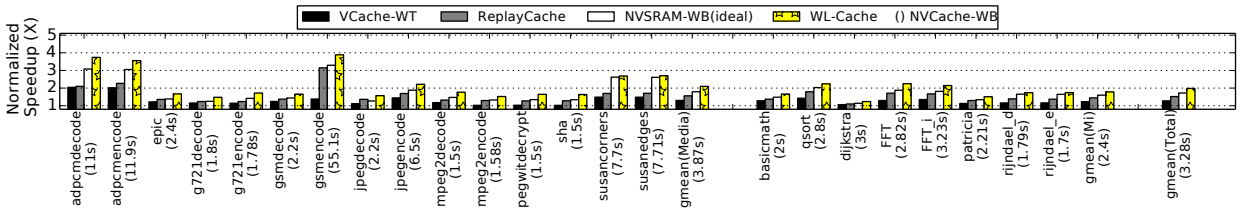


Figure 6.7. Normalized speedup of each cache design compared to NVCache in Power Trace 2. WLCache is 1.98x, 1.55x, 1.3x, and 1.12x faster than NVCache-WB, VCache-WT, ReplayCache, and NVSRAM-WB, respectively.

6.6.3 Performance Analysis

Performance Analysis without Power Outages: To analyze the performance impact of WL-Cache, we set the baseline to be NVCache-WB [22], [23] and compared with VCache-WT, ReplayCache with VCache-WB [54], NVSRAM-WB [21], and WLCache. Figure 6.5 shows the speedup over the baseline (NVCache-WB) when there is no power failure. Overall, WLCache

shows a similar speedup to NVSRAM-WB for all tested applications, achieving a 3.1x speedup on average.

The baseline NVCache-WB is the slowest as it has to pay long latency for every nonvolatile cache access. VCache-WT could take advantage of (SRAM-based) fast cache hits. ReplayCache improves the performance since it overlaps the next instruction execution with NVM stores of the same program region without waiting the corresponded ACK like ILP; it persists all stores at region-level granularity. Thanks to the region-level persistence with ILP execution, it can achieve almost 60% speedup compared to VCache-WT. On the other hand, NVSRAM-WB shows the best performance among all designs, demonstrating the benefit of write-back.

WLCache was slower than NVSRAM-WB, implying that WLCache could hold enough dirty cache lines. With waterline-based write-back, WLCache did not suffer much from potential stalls. WLCache also effectively hide the cost of asynchronous write-back operation, exploiting ILP.

Performance Analysis with Power Outages: Figures 6.6 and 6.7 show the speedup of each cache design using Power Trace 1 and 2, respectively. We took into account both power-on and power-off periods in this experiment.

For all applications, WLCache shows the best performance among all designs. WLCache achieves on average about 2.25x and 1.98x speedup compared to the baseline in Trace 1 and 2, respectively. WLCache is 71% and 55% faster than VCache-WT, and 32% and 30% faster than ReplayCache in Trace 1 and 2, respectively, demonstrating the benefits of holding dirty cache lines and exploiting cache locality. For the case with power failures, WLCache turns out to be faster than NVSRAM-WB by roughly 9% and 12% on average in Trace 1 and 2, respectively. Upon a power failure, WLCache needs to persist a bounded number of dirty cache lines, whereas NVCache-WB should reserve more energy to support cache backup. With less energy reserved for JIT checkpointing, WLCache could efficiently use the energy to compute and make a further forward progress.

Write Traffic with Power Outages: Figure 6.8 shows the write traffic overhead of WLCache compared to NVSRAM-WB cache using Trace 1. The result demonstrates that WLCache slightly increases the write traffic; however, it can be paid off by enabling asynchronous write back and adaptive execution as proven in Figure 6.6.

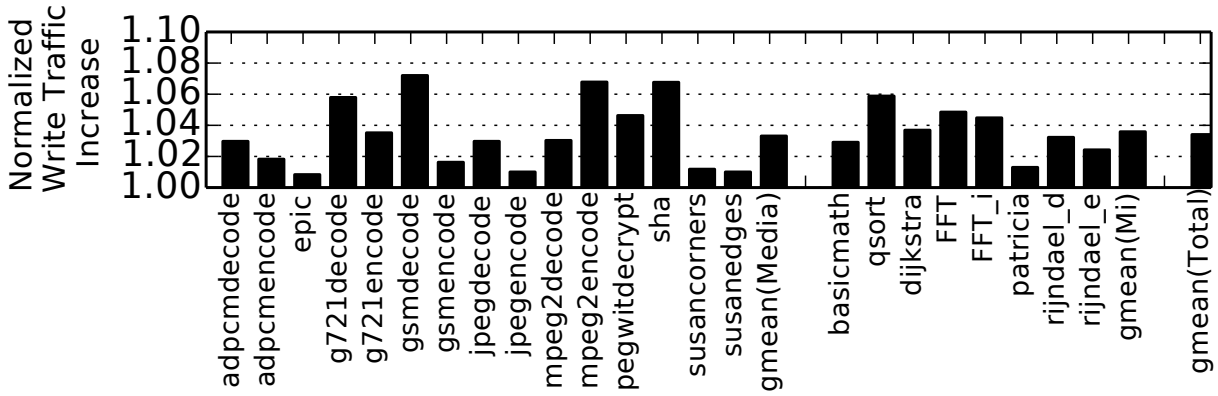
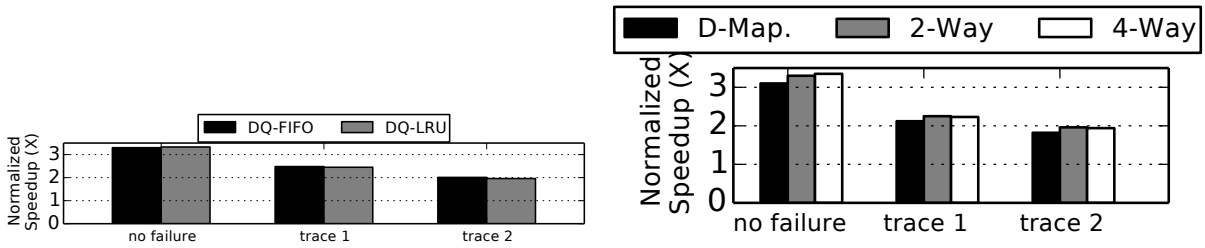


Figure 6.8. Normalized write traffic increase compared to NVSRAMCache in Power Trace 1.



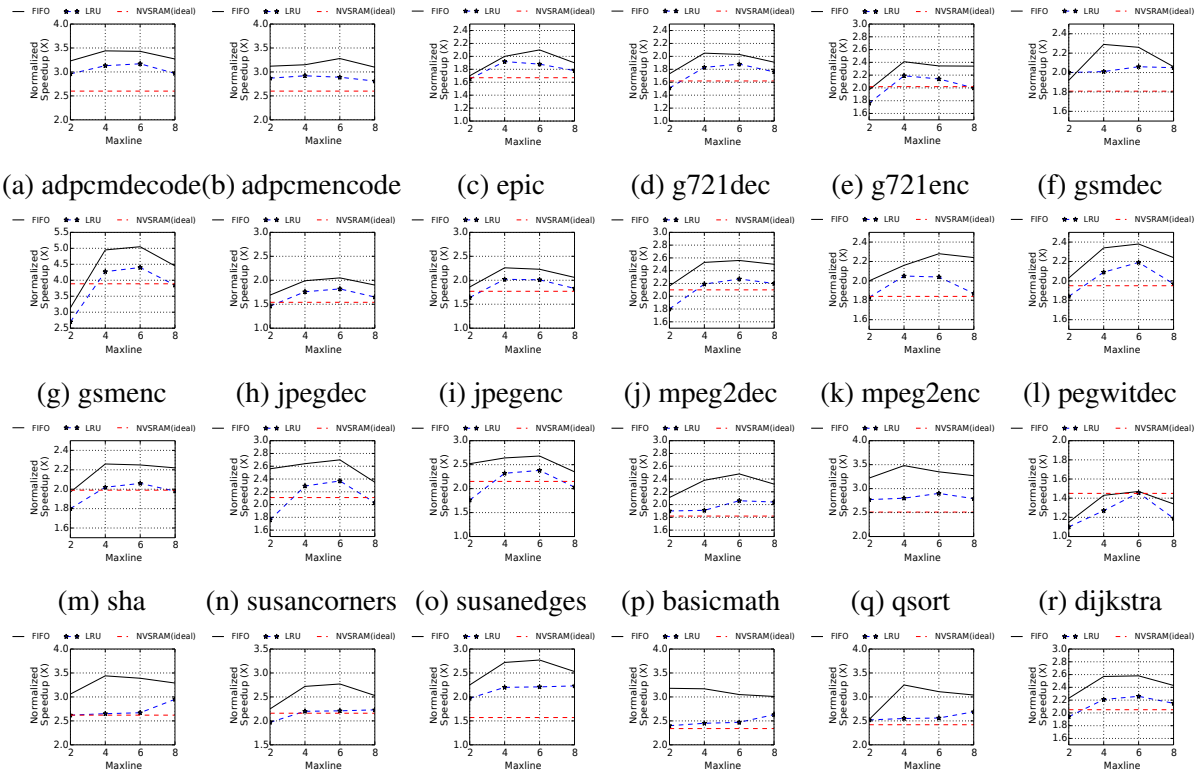
(a) DirtyQueue replacement

(b) Cache set associativity

Figure 6.9. Normalized speedup of WLCache with different DirtyQueue replacement and different cache set associativity compared to NVCache-WB on average.

6.6.4 DirtyQueue Replacement Policy

We measured performance of WLCache by varying the DirtyQueue replacement policies discussed in Section 6.5.3. WLCache with DirtyQueue-FIFO (DQ-FIFO) shows slightly higher performance than WLCache with DirtyQueue-LRU under power failures as shown in Fig. 6.9(a). Because WLCache does not evict the cache line and keeps it in the cache as clean regardless of DirtyQueue replacement policy, most subsequent memory references would show similar cache hit/miss trends. In general, LRU is known to be better than FIFO but it turns out that the additional power consumption for the LRU lookup logic often offsets the potential benefits of LRU, and the cache is often not warm enough to see the visible merits of LRU in energy harvesting systems. WLCache uses the FIFO-based DirtyQueue replacement policy by default.



(a) adpcmdecode (b) adpcmencode (c) epic (d) g721dec (e) g721enc (f) gsmdec

(g) gsmenc (h) jpegdec (i) jpegenc (j) mpeg2dec (k) mpeg2enc (l) pegwitdec

(m) sha (n) susancorners (o) susanedges (p) basicmath (q) qsort (r) dijkstra

(s) fft (t) ifft (u) patracia (v) rijndec (w) rijnenc (x) avg(gmean)

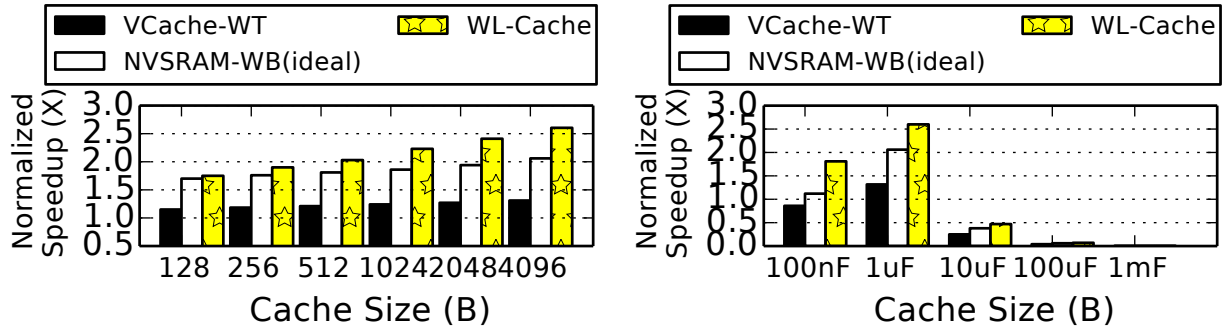
Figure 6.10. Sensitivity analysis on applications varying maxline sizes and cache replacement policies.

6.6.5 Sensitivity Analysis

We conducted sensitivity analysis on WLCache by varying the maxline size, cache replacement policy (LRU and FIFO), set associativity, cache size, and capacitor size. Notably, we explore cache replacement policy in this section (with FIFO-based DirtyQueue replacement policy). For analysis, we used NVCache-WB as baseline and measured the performance of each cache design using Trace 1. Fig. 6.10 shows the results.

Maxline Variation: We investigate how its performance varies when the maxline changes from 2 to 8 along with different replacement policies whose results will be discussed in the next section.

WLCache shows good performance with maxline 4 or 6. The performance differences between maxline 4 and 6 are not significant. With too large maxline (*e.g.,* 8), performance degrades as WLCache has to reserve more energy for JIT checkpointing. With too small maxline (*e.g.,* 2), performance also degrades as WLCache does not take advantage of dirty cache lines and locality.



(a) Cache size variation

(b) Capacitor size variation

Figure 6.11. Normalized speedup of each cache design compared to NVCache in Power Trace 1 varying (a) each cache size and (b) capacitor size on average.

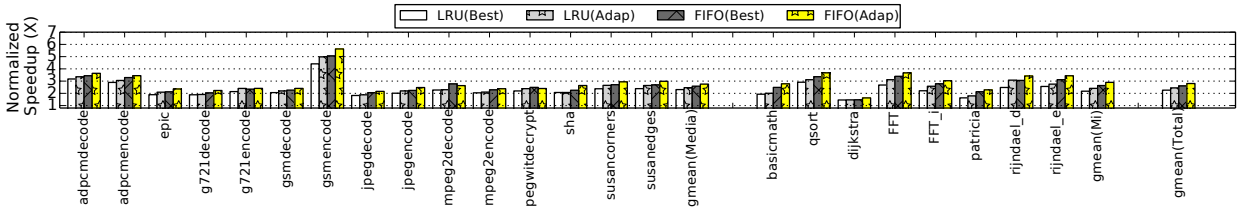


Figure 6.12. Normalized speedup of WLCache with adaptive management compared to NVCache in Power Trace 1. WLCache is 2.8x, 2.12x, 1.63x, and 1.35x faster than NVCache-WB, VCache-WT, ReplayCache, and NVSRAM-WB, respectively.

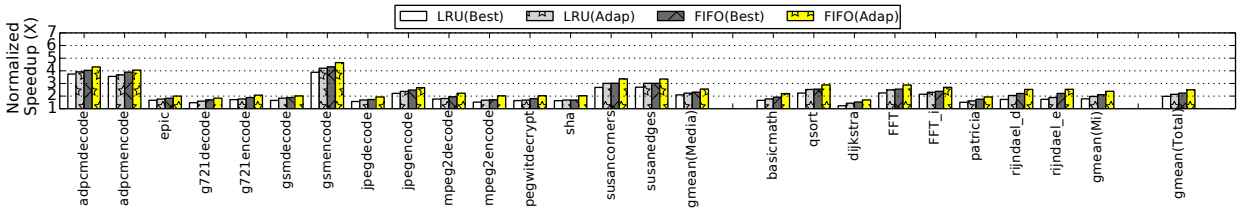


Figure 6.13. Normalized speedup of WLCache with adaptive management compared to NVCache in Power Trace 2. WLCache is 2.49x, 1.93x, 1.64x, and 1.48x faster than NVCache-WB, VCache-WT, ReplayCache, and NVSRAM-WB, respectively.

Cache Replacement Variation: Fig. 6.10 shows that FIFO-based WLCache (black line) outperforms LRU-based WLCache (blue dotted line) for cache replacement policies. When compared to NVSRAM-WB (red dotted line), FIFO-based WLCache performs better for almost all applications in all the maxline settings. For general purpose systems, LRU normally performs better compared

to FIFO. However, for energy harvesting systems with frequent power outages, we found FIFO is always faster than LRU. Further investigation reveals two main reasons for this surprising result.

First, the impact of cache replacement policy on cache miss rate is very limited in energy harvesting systems. We found that both policies cause almost the same cache miss rates. Energy harvesting systems experience frequent power failures, and they wake up with a "cold" cache—that has no data—across power failure. Therefore, WLCache is likely to cause compulsory (cold) misses across power failure regardless of the replacement policy. Furthermore, the systems run applications for a short amount of time. Within the limited time, the room for a smart cache replacement policy to address other conflict misses is simply small.

Second, the FIFO policy is energy-efficient without requiring additional cost to track LRU/MRU list at every memory access unlike LRU, that takes more latency and consumes more power possibly causing more power outages; we found LRU caused more power outages for most of applications.

Cache Size Variation: Fig. 6.11(a) illustrates that the normalized performance speedup of alternative cache schemes with a different cache size from 128B to 4KB using Power Trace 1. The results indicate that the performance gap between WLCache and NVSRAM-WB is decreased when the cache size is decreased, and vice versa; the speedup is also increased as the cache size increases.

Capacitor Size Variation: For capacitor size sensitivity analysis, we used NVCache-WB with 1uF as baseline, and measured the performance of alternative cache design with a different capacitor size from 100nF to 1mF using Power Trace 1. Fig. 6.11(b) describes the normalized performance speedup of alternative cache schemes. Overall, WLCache is always faster than other schemes. In particular, all schemes show their best performance when the capacitor size is 1uF. However, when the capacitor size is increased more than 1uF, their performance is decreased. This is mainly because their charging time is increased when the capacitor size is increased. On the other hand, the performance gap between WLCache and NVSRAM-WB is decreased when the capacitor size is increased, and NVSRAM-WB is eventually faster than WLCache when the capacitor size is 1mF.

6.6.6 Adaptive Maxline/Waterline Threshold Management

The above sensitivity studies motivate WLCache's adaptive maxline (and waterline) management optimization (Section 6.4). The performance of WLCache significantly varies by the maxline size, and thus the adaptive WLCache scheme (based on the estimated quality of power sources) has a potential to improve the performance further compared to the fixed size settings. For comparison, we measured the performance gain of "adaptive" WLCache with FIFO and LRU replacement policies and compared them with "static" WLCache. For the fixed settings, we picked the best performing maxline size for each application as we found in Fig. 6.10.

Figures 6.12 and 6.13 show the experimental results for Trace 1 and 2, respectively. We observe the similar performance trend for both power traces; Adaptive WLCache outperforms static-Best WLCache for both FIFO- and LRU- cache replacement schemes. Overall, for Trace 1, FIFO (Adap) and LRU (Adap) achieve 2.8x and 2.44x speedups, respectively, while FIFO (Best) and LRU (Best) get 2.6x and 2.25x speedups, respectively, For power trace 2, FIFO (Adap/Best) and LRU (Adap/Best) show speedups of 2.49x/2.24x and 2.14x/1.98x respectively. With FIFO(Adap), WLCache is 2.8x and 2.49x faster than NVSRAM-WB in Trace 1 and 2, respectively.

On average, WLCache reconfigures the maxline (and waterline) thresholds 11 and 12 times on trace 1 and 2, respectively. The minimum and maximum values of maxline are 2 and 6 for both traces. The energy source prediction accuracy is more than 98%, and thus the performance impact of mis-prediction is minimal in both traces. With adaptive threshold management, we also measured the number of dirty lines and the number of write-backs during each power-on period on average, which are 6/3 and 6/2 (dirty-lines/write-backs) in trace 1 and 2, respectively. In addition, the pipeline stall causes a trivial amount of time delay, less than 1% of total execution time on average in both traces.

6.6.7 Energy Consumption Analysis

We measured energy consumption of WLCache and compared it to other cache designs with power outages for each part of the system: core (computation), cache, and main memory (NVM). For analysis, we set WLCache with FIFO replacement policy and adaptive maxline management optimization. Figure 6.14 shows the breakdown, normalized to the same NVCache-WB baseline

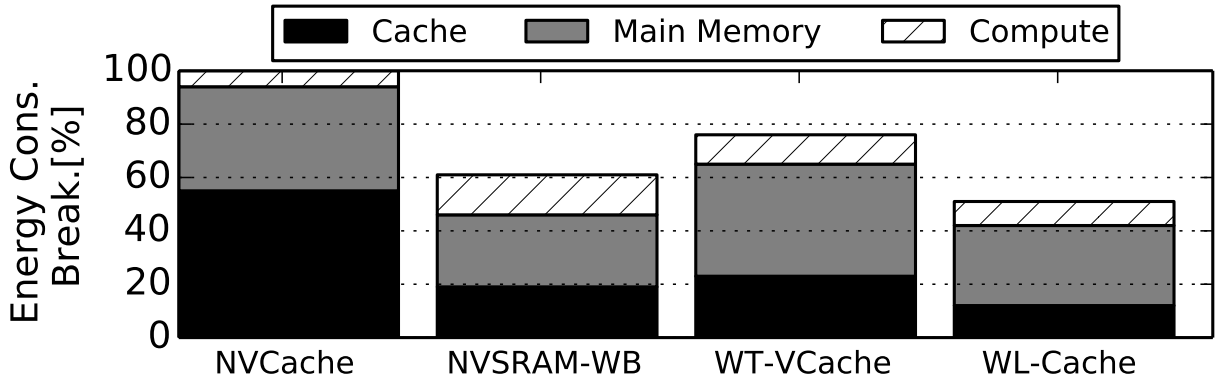


Figure 6.14. Normalized energy consumption breakdown in different cache designs.

on average, using Trace 1. Overall, WLCache reduces the total energy consumption by almost 50% compared to the baseline. In particular, WLCache significantly reduces the energy consumption from a cache part, which is less than the other designs. Though WLCache may access the NVM more frequently than NVSRAM-WB due to the maxline constraint, WLCache runs faster.

6.7 Summary

We introduce a new cache organization with a new write policy for energy harvesting systems, called WLCache. WLCache combines the benefits of write-back caches (efficiency) and write-through caches (persistence) without causing their downsides. Our evaluation demonstrates that WLCache significantly improves the performance and performs the best among all existing cache designs.

7. CAPOS: CAPACITOR ERROR RESILIENCE FOR ENERGY HARVESTING SYSTEMS

So far we have discussed how to improve performance for power failure recovery solutions. In this chapter, we focus on a reliability problem (i.e., capacitor error) in EHS devices and introduce CapOS, an OS-driven capacitor error resilience solution. However, it is particularly challenging to address the capacitor error problem. If we detect the problem proactively, it may cause false-positive errors causing performance degradation. If we detect the problem reactively, it may cause false-negative errors leading to a wrong recovery problem.

7.1 Introduction

Energy harvesting systems are an attractive alternative to battery-operated IoT. Thanks to the batteryless nature, energy harvesting systems cover a wide range of batteryless IoT applications [7], [33], [72], [78]–[83], [170]–[174]. However, due to the unreliable nature of ambient energy sources, energy harvesting systems suffer frequent and unpredictable power failure on which all volatile data are lost breaking program correctness. To address the problem, they use NVM as main memory with no cache—due to its power demand—and have some form of recovery support to backup and restore volatile data, i.e., registers, across the power failure [4], [8], [14].

For the recovery, energy harvesting systems leverage a capacitor-backed just-in-time (JIT) checkpoint mechanism [3], [6], [14], [171], [174]. At a high level, they keep gauging the level of the buffered energy in the capacitor with a voltage monitor and checkpoint volatile data (registers) into NVM when a power outage is about to occur. Then, in the wake of the outage, they restore the checkpointed registers from NVM and resume the program from the power-interrupted point with no rollback—thus being called roll-forward recovery. In this way, the JIT checkpointing ensures correct recovery, giving energy harvesting systems an illusion that the volatile data are persistent across power failure.

Unfortunately, we discover that the capacitor-backed JIT checkpointing can be inoperative due to capacitor degradation in real energy harvesting settings. Since ambient energy sources are unstable causing frequent power failure, the capacitor is repeatedly charged and discharged, which

is a stressful condition expediting the capacitor degradation [19], [28]–[30]. Furthermore, since energy harvesting systems are used as IoT devices in various environments, their capacitors are often exposed to other stressful conditions, e.g., high humidity/temperature [28], [31]. Under the circumstances, the capacitor is seriously degraded over time and finally unable to buffer enough energy for JIT checkpointing. Due to the insufficient amount of buffered energy in the degraded capacitor, the systems fail the checkpointing, thereby corrupting/losing volatile data (without providing any further service) across power failure; we refer to the problem as a *capacitor error*.

To deal with the capacitor error, we introduce CapOS, a lightweight operating system (OS)-driven capacitor error resilience solution. CapOS runs in either (1) normal mode or (2) safe mode depending on the capacitor condition. When the capacitor is not degraded, CapOS runs in the normal mode where JIT checkpointing is used as a roll-forward recovery mechanism. Upon capacitor error detection, CapOS enters the safe mode that conducts a rollback recovery with the JIT checkpointing disabled; while the degraded capacitor is idle, it restores the original capacitance on its own thanks to capacitor's resilient nature [29], [175]. Later, when the capacitor is fully recovered, CapOS gets back to the normal mode. Thus, CapOS switches between the two modes to recover from any capacitor error detected.

In particular, CapOS dynamically detects the capacitor error in a reactive yet safe manner. It opts for reactive error detection mainly due to accuracy and performance benefits over proactive detection that suffers false positives leading to unnecessary mode switches and the resulting performance penalty. However, since reactive error detection condones the JIT checkpointg failure corrupting the data, care must be taken to keep all the registers being checkpointed safe. For this purpose, CapOS leverages an acknowledgment (ACK) as a barrier of the JIT checkpointing, i.e., CapOS persists the ACK after checkpointing all registers in NVM. The key insight here is that a capacitor does not become faulty all of a sudden, but instead it is gradually degraded over time [28]–[31], [36]. That is, when the capacitor is first degraded, the JIT checkpointing only fails to write the last data, i.e., ACK, to NVM yet succeeds in persisting all registers. Consequently, if the capacitor error occurs, CapOS can detect it by checking the ACK in the wake of power failure.

Once the ACK corruption is detected, CapOS switches its execution mode from normal to safe. In safe mode, CapOS disables the JIT checkpoint mechanism since the registers can eventually be corrupted as the faulty capacitor is further degraded. Then, CapOS electrically isolates the faulty

capacitor to restore its original capacitance via the self-recovery nature of a supercapacitor which can restore its original capacitance while it rests [29] [1]. To recover from power failure without JIT checkpointing (roll-forward recovery), the safe mode needs another type of power failure recovery solution, i.e., rollback recovery [12], [32], [33].

However, it is daunting challenging to dynamically switch from the roll-forward to the rollback and vice versa. This is because, unlike the roll-forward recovery, the program should be partitioned to a series of recoverable regions, the boundary of which serves as a rollback recovery point in case the following region is interrupted by power failure [12], [32], [33]. The problem is the mode change can happen at arbitrary time. When the rollback recovery solution is enabled not necessarily at the region boundary, it may not restore required data (e.g., memory logs or checkpoints) across power failure, thereby causing the wrong recovery problem.

To correctly enter the safe mode, CapOS leverages a timer-based checkpointing and copy-on-write (CoW) with a memory protection unit (MPU) as a boundary-free recovery scheme. Instead of using the boundaries, CapOS sets a recovery point at each timer expiration. Within the timer interval, CapOS tracks every memory update by leveraging the MPU, and logs the original pages as the CoW. When a new interval starts, CapOS clears the page information logged in the previous interval to track its own memory updates and checkpoints all registers as a recovery point. Thanks to the boundary-free nature, CapOS can seamlessly switch the execution mode and ensure correct recovery.

Once the capacitor becomes reliable, CapOS returns to the normal mode. To figure out when to return, CapOS leverages a capacitor recovery model and dummy JIT checkpointing. CapOS dynamically measures the capacitor isolation period and compares it to the capacitor recovery model [28], [29]. If the total isolation time exceeds the model, CapOS examines the capacitor by enabling the JIT checkpointing with dummy data. Once the dummy JIT checkpointing succeeds, CapOS assumes the capacitor becomes reliable and switches back to the normal mode. Consequently, CapOS switches back and forth between the rollback and roll-forward recovery solutions.

Our contributions can be summarized as follows:

[1]↑ A supercapacitor is used as energy buffer for many prior works such as ECO, Capybara, and so on [7], [33], [72], [78]–[83].

- We discover a capacitor error where energy harvesting systems corrupt/lose their data or fail to provide any service when their energy buffer (i.e., capacitor) is degraded.
- We introduce an operating system driven capacitor error resilience solution called CapOS, that can preserve volatile data against capacitor errors and recover the degraded capacitor by leveraging its self-recovery nature.
- Our experiments demonstrate that CapOS causes less than 1% performance overhead on average compared to a (unprotected) roll-forward recovery when there is no capacitor error. In the presence of capacitor errors, CapOS incurs only 15% performance overhead on average.

7.2 Background and Challenges

To address the capacitor errors, one might suggests rollback recovery solutions since they always preserve volatile data in safe across power outages (Section 7.2.1). Also, a prior work can periodically isolates a capacitor to restore its capacitance by leveraging the self-recovery nature (Section 7.2.2), which might prevent from the capacitor errors. Furthermore, another prior works can ensure a voltage margin to achieve safe JIT checkpointing with a degraded capacitor [3], [6], [7], [14](Section 7.2.3).

7.2.1 Rollback Solutions

The rollback recovery solutions form a series of recoverable program tasks in case of power failure [4], [12], [13], [68]. In detail, to achieve correct recovery across power failure, they insert logs and checkpoint stores within each program task. If power failure occurred, they roll back to the beginning of the interrupted task by restoring the checkpointed data and logs in the wake of power failure without causing the memory inconsistency problem.

However, if there is a task whose execution time is greater than the power failure period, the program ends up rolling back to the beginning of the same task indefinitely (i.e., stagnation problem as discussed in Section 2.2.2). To address the problem, a prior work called Chinchilla statically form the task by considering the capacitor size [12] so that it can complete at least one task across any power outage case including the worst case scenario.

Not a Solution: Unfortunately, Chinchilla ends up generating short tasks and causing additional checkpoint stores; hence, it causes 50∼400% run-time overhead [6]. Since the considerable overhead is impractical for power-hungry energy harvesting systems, it leads to JIT checkpointing roll-forward recovery solutions (Section 2.3). More importantly, since the rollback solution statically forms a series of program tasks by considering the capacitor size, its task can be stagnated when the capacitor is degraded as shown in Table 2.1. Due to the stagnation vulnerability, the rollback recovery solutions cannot address the capacitor errors.

7.2.2 Periodic Capacitor Isolation

With the self-recovery nature of capacitors in mind (Section 2.4.2), a prior work [29] suggests to periodically leave the degraded capacitor in idle for capacitor aging deceleration. For example, in every thousands of charge/discharge cycles, they give the capacitor a rest period, e.g., the systems stop their operation disconnecting the capacitor at an interval of thousands charge cycles. Thanks to the isolation, they found that the capacitor degradation rate could be decreased as shown in Fig. 7.1; the lifespan is exponentially increased when it is isolated at lower stop intervals.

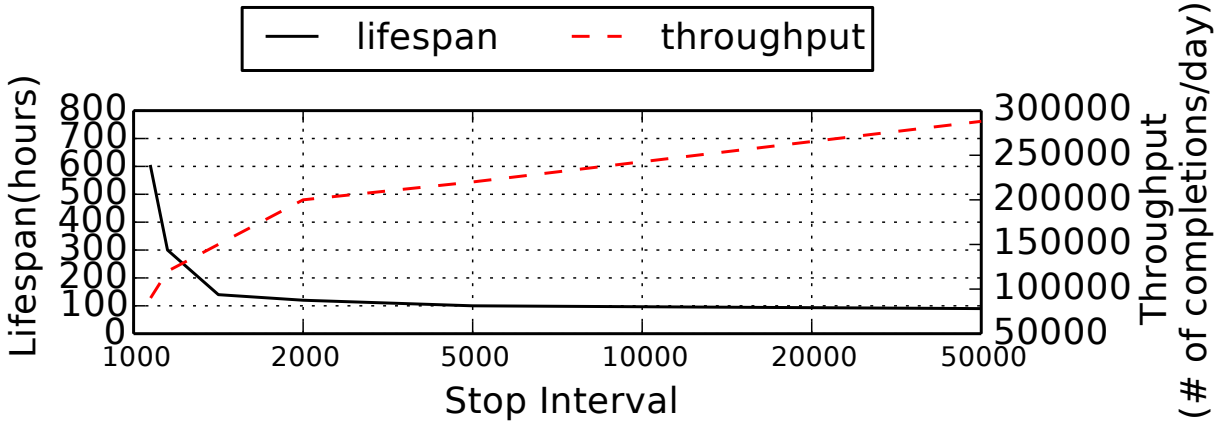


Figure 7.1. Capacitor degradation deceleration and its side-effect analysis in energy harvesting systems

However, the periodic capacitor isolation is impractical for energy harvesting systems. In fact, since energy harvesting situation is unstable and unpredictable, the capacitor may or may not suffer from the stressful charge/discharge condition. In other words, the solution can unnecessarily

isolate a capacitor although it has not been degraded. In addition, while the capacitor is being isolated, energy harvesting systems provide poor performance as a side-effect. This is because the systems cannot enable the JIT checkpoint mechanism without using a capacitor, and thus inevitably stop their operation; otherwise, they can cause the wrong recovery problem overwriting or losing important data in NVM.

Poor Performance: To analyze the performance overhead of the prior work, we measured the quality of service running benchmark applications; we estimated throughput by counting the number of task completion times during a day with various the capacitor isolation interval, i.e., we stopped its program execution for capacitor isolation at every certain number of power failures.

From our experiments, we found out that the more frequently the capacitor is isolated (less stop cycles), the more performance overhead it causes. On the other hand, the capacitor lifespan is inversely proportional to the performance overhead. Fig. 7.1 shows the trends as an example; it covers one of benchmark applications, *bitcount*, that is used for performance analysis in prior works [6]–[8]. As shown in the figure, while the throughput is increased as the stop interval is increased, the capacitor lifespan is decreased. The takeaway is that, to ensure long lifespan, the prior work must frequently stop its operation; however, it results in low throughput, i.e., poor quality of service.

Not a Solution: More importantly, although the periodic capacitor isolation can lengthen the lifespan of capacitors decelerating the capacitor degradation rate, it is still susceptible to the capacitor error. This is because the capacitor can be degraded between adjacent rest intervals, and thus the systems can eventually fail to make a checkpoint when the capacitor is being used for JIT checkpoint.

7.2.3 Voltage Margin Solution

To ensure safe JIT checkpointing, prior works ensure 10% checkpoint voltage margin [3], [6], [7], [14]; they reserve a small amount of energy to achieve safe checkpointing in case of unexpected errors as $V_{backup} > V_{off}$. In this way, they can increase a checkpoint voltage threshold (V_{backup}) and a start-up voltage level (V_{on}) to buffer more energy for safe checkpointing when the capacitor is degraded.

Poor Performance: However, the large safe margin causes performance overhead. This is mainly because the increased voltage/energy can only be used for checkpointing purposes, not for computation/progress. Also, such a large safe margin can further increase the start-up voltage level, which can take a lot longer reboot/recharge time than the lower start-up voltage to secure the high energy enough across power failure. That is why none of the prior works uses such a large safe margin.

Not a Solution: Although they have such a large safe margin, the prior works are still not free from the capacitor error problem. This is because the capacitor is eventually degraded as discussed in Section 2.4.1, i.e., a capacitor can be degraded by 50% within a year in real energy harvesting situation, thereby causing the checkpoint failure and losing data in NVM.

7.3 Design

CapOS is a lightweight capacitor error resilience solution for energy harvesting systems, which can detect the capacitor error and ensure safe recovery. At a high level, CapOS runs in either normal mode (Section 7.3.1) or safe mode (Section 7.3.2). In the normal mode, CapOS runs with a JIT checkpoint mechanism as a roll-forward recovery solution and diagnoses a capacitor in a reactive manner at each reboot time. In the safe mode, CapOS runs with a rollback recovery mechanism to prevent from data loss (or checkpoint failure) and stagnation.

However, it is challenging to switch the recovery solution. This is because typical rollback recovery solutions [12], [32], [33] partition program into a series of recoverable code regions—whose boundary serves as a rollback recovery point—by leveraging new programming model, compiler, or runtime system support. Since the mode change can happen at arbitrary locations, the rollback recovery solution can be enabled in the middle of program region without having required data (e.g., memory logs or checkpoint stores) for correct recovery, thereby causing wrong outputs. As an alternative solution, CapOS may hold multiple versions of program and dynamically keep them all in sync in case of mode change; however, it is technically complicated and impractical. To address the issue, CapOS leverages OS modules such as a timer-based checkpointing and a CoW mechanism with a MPU as a boundary-free recovery scheme.

Figure 7.2. CapOS workflow: CapOS diagnoses a capacitor at each reboot time. If the capacitor error is detected, CapOS disables the JIT checkpointing and switches its execution mode from normal (blue shaded box) to safe (yellow shaded box). If a capacitor can be recovered in the safe mode, CapOS would switch its execution mode back to normal.

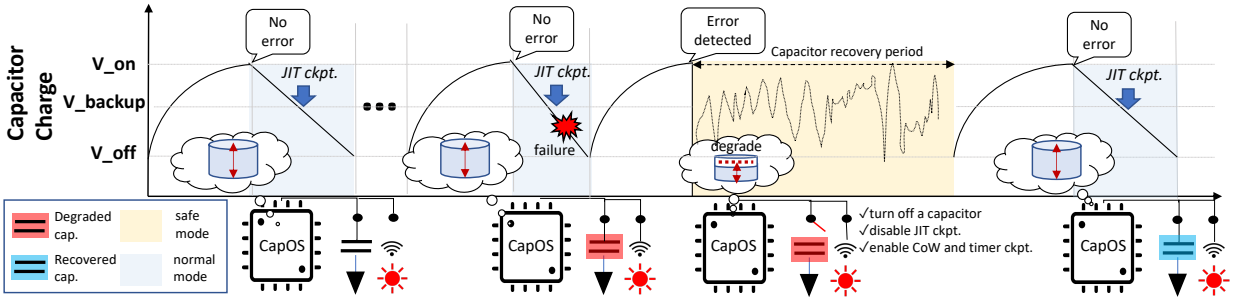
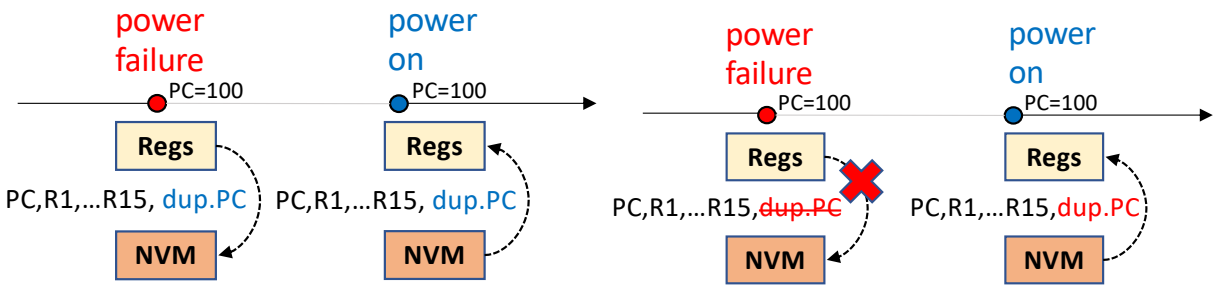


Fig. 7.2 describes the overall workflow of CapOS. For the first and second capacitor charge cycles in the figure, CapOS runs in the normal mode. It diagnoses a capacitor at reboot time, and runs with the JIT checkpoint mechanism as long as the capacitor is not degraded. When the capacitor is degraded as shown in the second capacitor charge cycle, CapOS fails to checkpoint data in NVM when a power outage occurs. Across the second outage, CapOS detects the checkpoint failure at the third reboot time and starts to run in safe mode. In this mode, CapOS enables the CoW with a MPU and switches off the power line connected to the capacitor for its self-recovery. Then, when the capacitor is recovered, CapOS turns back to the normal mode, disabling the CoW and reconnecting the capacitor; this will be discussed in Section 7.3.3.

7.3.1 CapOS in Normal Mode

A capacitor does not become faulty all of sudden, but it is gradually degraded over time as discussed in Section 2.4.1 [28]–[31], [36]. Due to the gradual degradation, the JIT checkpointing fails to persist the last 2-byte of volatile data when a capacitor is degraded; the JIT checkpointing sequentially stores volatile data (e.g., registers) to a designated space in NVM at power-off time by spending the rest of buffered energy in a capacitor. Note that, since the current energy harvesting system platform guarantees failure-atomic NVM writes at word size granularity (i.e., 2-byte) [32], the last volatile register (e.g., R15) is lost for the first when the capacitor error occurs.



(a) Successful JIT checkpoint (b) Unsuccessful JIT checkpoint

Figure 7.3. JIT checkpoint failure (capacitor error) detection mechanism with a duplicated PC

With this in mind, CapOS stores a duplicated PC register as an ACK in NVM for the last part of the JIT checkpointing protocol. Fig. 7.3 describes the checkpoint failure detection mechanism. CapOS stores a duplicated PC at the end of the JIT checkpointing when a power outage occurs. In (a), CapOS stores the duplicated PC value, i.e., 100, at power failure point. Then, in the wake of power failure, CapOS restores the duplicated PC and the original PC from NVM. In this case, since the duplicated PC is the same as the original PC, CapOS considers the capacitor is reliable. On the other hand, in (b), CapOS fails to store the duplicated PC to NVM. In this case, CapOS detects the capacitor error by comparing the original PC and the corrupted PC across power failure.

Notably, the ACK-based checkpoint failure detection mechanism protects registers from being lost. As shown in the figure, the registers can be persisted while losing the ACK, and thus CapOS can achieve correct recovery across a power outage; the registers can be lost when the capacitor continues to degrade more than 2 hours after the ACK failure. With the help of correct recovery, CapOS can disable the JIT checkpointing and switch the execution mode at the next reboot time (i.e., when it detects the capacitor error).

7.3.2 CapOS in Safe Mode

Once CapOS detects the JIT checkpoint failure, CapOS switches its execution mode from the normal to the safe mode in a seamless manner by leveraging OS modules such as timer interrupt and CoW with a MPU.

Timer-based checkpoint: The timer-based checkpoint mechanism flexibly sets a recovery point by considering the energy source condition without requiring statically-defined task boundaries [8].

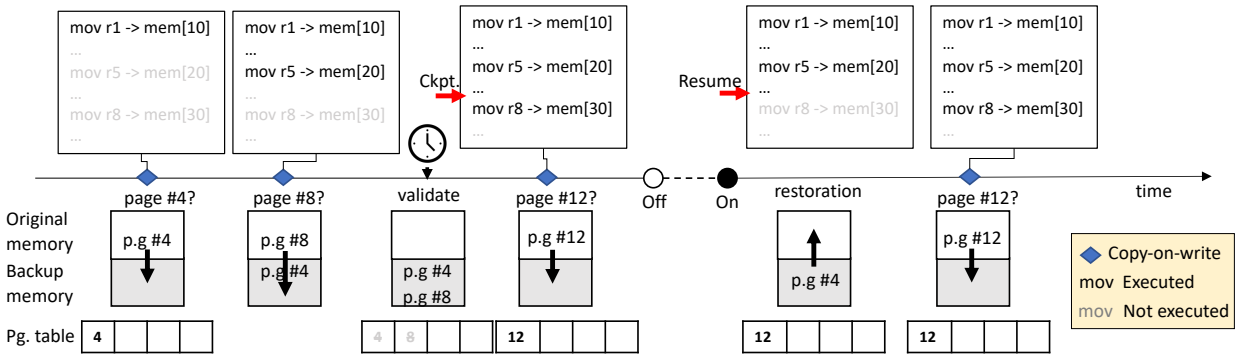


Figure 7.4. CapOS marks the populated pages in a page table during a checkpoint interval. Across power failure, CapOS restores checkpointed registers and the marked page(s) to resume the interrupted program.

For example, when the energy source condition is good, it can increase the timer interval to avoid frequent checkpointing. On the other hand, when the energy source condition is weak, it can decrease the timer interval to make a checkpoint at least once during the power-on period by considering the power failure frequency. In particular, when the capacitor is degraded holding a less amount of energy, CapOS can further decrease the timer interval to checkpoint at least once, thereby achieving the forward progress execution with a degraded capacitor.[2] However, since the timer-based checkpoint persists only register data, it should also preserve other data in NVM across power failures for crash consistency. To address this issue, CapOS leverages the CoW with a MPU.

Copy-on-Write: CapOS checks each store operation if it has a write permission of the target page by looking over a page table. Fig. 7.4 describes that CapOS can track every memory access on a per-page. For example, at each store operation (e.g., mov r1→mem[10]), CapOS checks if it has a write permission of the target page by looking over a page table, which is managed in NVM. If it does not have a write permission, CapOS makes a copy of the target page in a designated backup memory space in NVM. Once the page copy is completed, CapOS marks the write permission for the page in the page table to grant it a write permission. Finally, when CapOS finishes the memory backup, it can execute the store instruction.

On the other hand, if CapOS has a write permission of the target page, both the page copy and the table mark steps are unnecessary. That is, for a store instruction that accesses to writable

[2]↑It keeps copies of checkpointed data as double buffering [4], [8] in case of power failure during checkpointing.

pages, CapOS checks only the page table. Then, CapOS can execute the memory write. By following these steps, no matter when power failure occurs, CapOS does not cause a crash consistency problem.

Memory Protection Unit: CapOS runs an uninstrumented binary and dynamically tracks memory updates by leveraging the MPU—that has been already equipped with the low-power MCU such as MSP430 series [176], [177]. In detail, CapOS partitions a memory into three segments, code, original data, and designated backup memory space (shadow memory) as shown in Fig. 7.5. The code section contains OS code, binaries, interrupt, and so on. The original data space contains heap, stack, and data. And, the shadow memory space is for backup, i.e., one-to-one mapped with original data space. Second, CapOS sets read, write, and execute permissions to each segment. In particular, CapOS sets the original data space to be not writeable. This is because CapOS needs to track memory updates in the data segment for CoW.

In this design, when the system tries to write data into the memory, the MPU ① checks the target memory address. If the address is in the original data section, the MPU ② sends a violation signal to CapOS with the address. Then, CapOS translates the address to a page ID and looks over a page table [8] then figures whether the target memory page has been already populated. If it has not, CapOS triggers the CoW backing up an original page to the shadow memory space. Note that CapOS manages the page table in the shadow memory segment; therefore, it is free to write/delete the entry without requiring the write permission. Note that, thanks to the MPU, CapOS can enable the CoW in a seamless manner without requiring compiler-assisted program [8]; the MPU permission checking takes only 1∼2 cycles.

Correct Power Failure Recovery: For correct recovery, CapOS sets a recovery point when the timer expires by checkpointing registers and clearing the page table. In other words, when the timer restarts, the page table is clear, i.e., no page has write permission. In the new timer interval, CapOS can newly track memory writes for crash consistency. If power failure occurs during the program execution, CapOS restores all the backed-up pages by looking over the page table in the wake of power failure. Then, it goes back to the recovery point where the checkpoint was made. In this way, CapOS can achieve crash consistency across power failure. Note that when the timer expires, CapOS must checkpoint register first then clear the page table; otherwise, CapOS can cause wrong

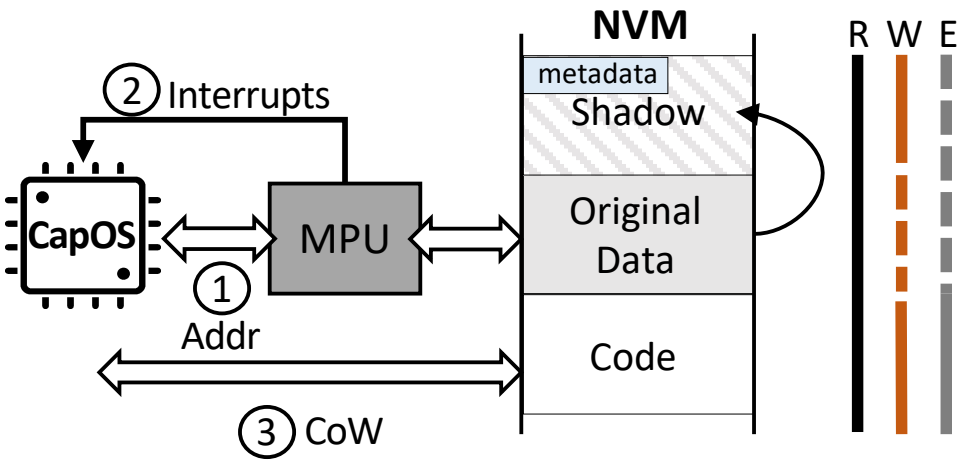


Figure 7.5. Copy-on-Write with memory protection unit (MPU). Dashed lines represent no permission while solid lines mean the system has a permission.

recovery. For example, if CapOS cleared the page table first, and power failure occurred during register checkpointing, the system would not be able to restore required pages across power failure.

Capacitor Recovery: CapOS restores a degraded capacitor in the safe mode by (electrically) isolating it; when the capacitor is isolated, the degraded capacitance can be recovered as its ESR decreases [29]. To isolate the capacitor, CapOS disconnects it from systems by using a programmable power gate [33]. In detail, CapOS leverages a controllable state-retaining switch by connecting it to MCU through GPIO pins. In this way, CapOS turns on/off the capacitor at a software level. Then, the input power from the ambient energy source can bypass the capacitor as storage-less systems.

7.3.3 CapOS Mode Change

CapOS switches back from the safe to the normal mode when the degraded capacitor is recovered. To ensure the capacitor is recovered enough, CapOS checks the capacitor condition by leveraging a two-step mechanism. The first step is to leverage a recovery model of a capacitor [29] defined as: $C_{recovery}(t,T,V_{end}) = a * exp(-\frac{t}{\tau_1}) + b * exp(-\frac{t}{-\tau_2})$, where a and b characterize the capacitor state, and τ_1 and τ_2 are the time constants governing the recovery rate of the capacitor; CapOS must be aware of these parameters from the device manual [29]. With this model, CapOS stati-

cally defines a recovery period. Then, CapOS dynamically measures the time taken for a degraded capacitor to get recovered and compares the time to the recovery period.

In detail, CapOS measures the accumulated power-on time across power outages by using the timer. CapOS checks whether the accumulated time is greater than the recovery period—defined by the recovery model—along the way. Once the measured time is greater than the recovery period, CapOS moves on the second step connecting the isolated capacitor back to the system.

The second step is to examine the JIT checkpointing. While it runs with the rollback recovery solution in the safe mode, CapOS also enables the JIT checkpointing, not for registers, but for dummy data, i.e., CapOS checkpoints dummy data at the power failure point. Then, at the next reboot time, CapOS checks whether the dummy JIT checkpointing has been safely finished or not. If the JIT checkpointing succeeded with the re-connected capacitor, CapOS switches back to the normal mode disabling the rollback recovery solution.

Discussion: It is possible that CapOS can switch back to the normal mode with a *not-fully-recovered* capacitor; however, the two-step mechanism ensures that the capacitor is at least recovered enough to buffer a sufficient amount of energy for safe JIT checkpointing. This is because CapOS conservatively measures the accumulated power-on time excluding the power-off time, i.e., the capacitor must be recovered longer than the recovery time. Also, CapOS ensures the safe JIT checkpointing with the dummy data.

Generality: CapOS can also be applied for energy harvesting systems that have another type of capacitors such as ceramic or electrolytic. Since those capacitors do not have such a self-recovery nature, CapOS does not have to isolate them for capacitance restoration when they are degraded. Instead, CapOS triggers the safe mode when the JIT checkpoint failure occurs, and continues to run in the safe mode without switching back to the normal mode. In this way, CapOS can always ensure correct recovery regardless of the capacitor type.

7.4 Implementation

7.4.1 JIT checkpoint with Duplicated PC

CapOS runs an uninstrumented binary and stores a duplicated PC as an ACK along with the JIT checkpointing at the power failure point (Section 7.3.1). Then, in the wake of power failure,

CapOS compares the original PC to the duplicated PC whether they are equal to each other. This causes only one more checkpoint store at power failure point, which requires about at most 6% more energy than the naive JIT checkpointing; we assume the systems at least 10% energy margin for JIT checkpointing as discussed in Section 7.2.3.

7.4.2 Timer handler

CapOS uses the timer not in normal mode, but in safe mode. In safe mode, CapOS enables the timer interrupt to (1) checkpoint (2) and measure recovery time. First, CapOS checkpoints when the timer expires and reconfigures the timer interval underline power failure behavior (Section 7.3.2).

Second, CapOS measures the total recovery time with the timer interrupt (Section 7.3.2) to switch back from safe to normal mode when a capacitor is recovered. CapOS conservatively accumulates power-on time across power outages as following: $N_{expiration} \times T_{period}$, where $N_{expiration}$ and T_{period} represent the number of timer expiration and the timer interval, respectively.[3] Then, CapOS compares the accumulated time to the predefined recovery period (Section 7.3.3). When the time is greater than the threshold, CapOS turns on the capacitor switch and enables the JIT checkpointing for dummy data to make sure whether the capacitor is fully recovered; this can be done by one store instruction in a failure-atomic manner. Note that, after enabling the JIT checkpointing, CapOS shuts down the system to run with a fully charged capacitor across a power outage–though the system can immediately suffer a power outage when the isolated capacitor is re-coupled because the harvested power must be buffered first then supply the sufficient power to the end-device. In other words, CapOS intentionally reboots the system for full capacitor diagnosis.

Once the capacitor turns out to be fully recovered, CapOS switches back from safe to normal mode; CapOS disables the rollback recovery but enables the roll-forward recovery. However, if the capacitor turns out to be not recovered failing to checkpoint the dummy data, CapOS disables the dummy JIT checkpointing and continues to run in safe mode waiting for another recovery period.

Persistent Timer: To measure not only the power-on period but also the power-off period, CapOS can leverage a persistent timer [141], [178], that can be achieved by using Static Random-Access

[3]↑CapOS must be able to reconfigure the T_{period} accordingly when the clock frequency changes.

Table 7.1. Lifespan analysis varying applications

Scheme	Capacitor	Application	Error Type	Location	Lifespan
Chinchilla [12]	100uF Elec.	dhrystone	Stagnation	Pro0() task	7 days
		fft	Stagnation	msp_cmplx_fft_fixed_q15() task	10 days
		basicmath, crc32, dijkstra, stringsearch, blinker, bitcnt, crc16, fir, qsort	Stagnation	main() task	≈ 1 year
Samoyed [6]	1mF Super.	basicmath, blinker, bitcnt, crc16, crc32, dijkstra, fft, fir, dhrystone, stringsearch, qsort	Register loss	Random	10 days
CapOS	1mF Super.	basicmath, blinker, bitcnt, crc16, crc32, dijkstra, fft, fir, dhrystone, stringsearch, qsort	No damage	-	72 years [42]

Memory (SRAM) scratchpad and capacitor-backed timer. First, the SRAM-based solution [178] can estimate the power-off period by measuring the percentage of decayed SRAM cells across power failure. However, when the power-off period is too long, all SRAM data disappear, i.e., the timer cannot measure the power-off period accurately. Second, the capacitor-backed persistent timer [141] measures the power-off time by using separated buffered energy in an additional capacitor. However, since the capacitor is vulnerable to the reliability issue, the timer cannot measure the power-off time eventually. Overall, since the persistent timer requires architectural support, e.g., SRAM scratchpad or additional capacitor, we leave it as our future work.

7.4.3 Reboot process

CapOS reboot process runs in either the normal mode or the safe mode. In the normal mode, CapOS diagnoses a capacitor by examining the checkpointed ACK (Section 7.3.1). If the ACK is not corrupted, i.e., JIT checkpointing succeeded at the recent power failure point, CapOS continues to run in the normal mode. On the other hand, if the ACK is corrupted, CapOS switches its execution mode from the normal to the safe mode disabling the JIT checkpointing and disconnecting the degraded capacitor (i.e., disabling the roll-forward recovery); these can be done with a 2-byte store operation in a failure-atomic manner. After that, CapOS directly runs in the safe mode.

In the safe mode, CapOS enables CoW with a MPU. Then, CapOS looks over the page table and restores the required data. CapOS further calculates the accumulated safe mode period. If the accumulated period in the safe mode is greater the predefined threshold, CapOS turns on the

capacitor switch and enables the dummy JIT checkpointing (Section 7.3.3); this can also be done with a 2-byte store operation in a failure-atomic manner. If the capacitor is fully recovered, CapOS switches back to the normal mode. Here, to run with a fully charged capacitor, CapOS also shuts down the system (Section 7.4.2).

7.5 Evaluation

7.5.1 Experimental Setting

In the same environment for Section 2.4.1, we measured the performance and lifespan of prior works [6], [12] and CapOS running 11 benchmark applications [3], [8], [25] with a real power trace (trace 12 in Table 7.2) that is collected from a real RF energy harvester [25]; we will vary the power trace and conduct the sensitivity analysis in Section 7.5.4.[4] For lifespan analysis, we repeatedly ran the applications until they suffered the stagnation or data loss problem; we consider the systems as dead when the problem occurs.

We consider Chinchilla [12] and Samoyed [6] as a basis/representative of rollback and roll-forward recovery scheme, respectively. In particular, the roll-forward recovery scheme called Samoyed forms the I/O operations to be safely recoverable across power failure since the JIT checkpointing is unavailable during the I/O operations [6].

7.5.2 Energy Harvesting System Lifespan Analysis

As shown in Table 7.1, we found that the lifespan of prior works are about 10 days. On the other hand, CapOS continues to run safely without causing the stagnation or data loss even when the capacitor is degraded; hence, we assume the lifespan of CapOS is 72 years as the same as the lifespan of MCU [42].

In particular, we found that Chinchilla [12], the state-of-the-art rollback recovery solution (Section 7.2.1), is more reliable than other prior works for some applications; it continues to make a forward progress with a degraded capacitor. This is possible because Chinchilla statically inserts a region boundary at each basic block, the entry of which checkpoints all registers, rendering the

[4] ↑We used the power traces to scale to the various experimental settings for our benchmark suite with multiple sensitivity analysis. If the real devices are used, the transmitter and harvester should move back and forth very fast with a large distance; otherwise, the power failure frequency is fixed like prior works [6], [32].

regions very short. Since the short regions require a small amount of energy to be completed, Chinchilla can finish them across power failure even if the capacitor is degraded—though 50% capacitance degradation eventually causes the stagnation problem; we assume the lifespan of Chinchilla is about a year since a capacitor is degraded by 50% within $\approx$ 1 year. However, Chinchilla can also generate long regions if applications have long basic blocks (e.g., dhrystone and fft). In this case, the long regions can cause the stagnation problem easily when a capacitor is degraded by more than 10% (Table 7.1).

7.5.3 Performance Analysis

To analyze the performance overhead of CapOS, we conducted experiments in the same setting for Section 7.5.2. We measured the throughput of Chinchilla, Samoyed, and CapOS for 11 benchmark applications (1) when the capacitor is not degraded and (2) when the capacitor is degraded. For throughput, we counted the number of application completion time during one day, i.e., $\frac{\#.of.comp}{24hours}$. We set the naive roll-forward recovery solution without capacitor protection, i.e., Samoyed, as a baseline. For peripheral (I/O) tasks in applications such as *blinker and fft*, we design them to be power-failure-atomic assuming that is required by system administrators [6].

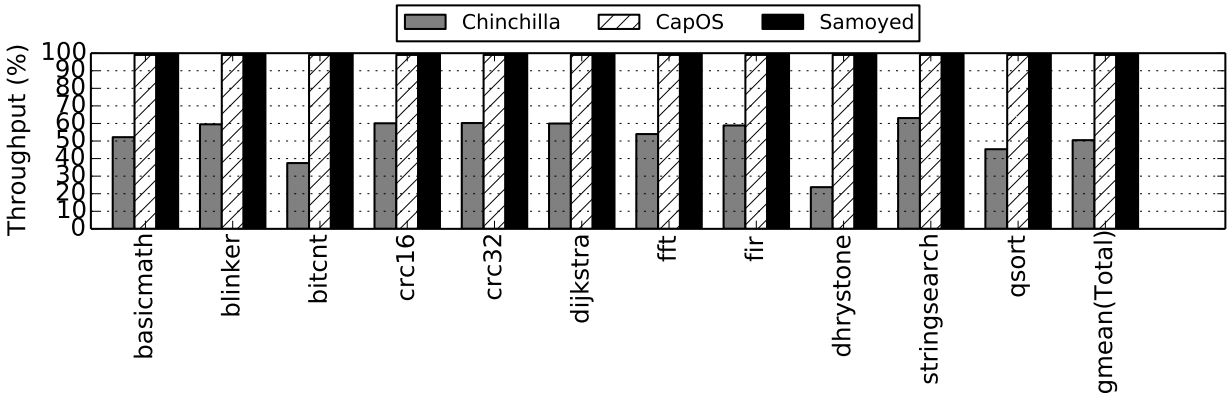


Figure 7.6. Normalized throughput of each recovery scheme compared to Samoyed without a degraded capacitor.

Performance Analysis Without a Degraded Capacitor: Fig. 7.6 describes the performance overhead of each recovery scheme. Without a degraded capacitor, Chinchilla causes 51% throughput on average compared to the baseline. Unlike the prior work, CapOS achieves almost the same

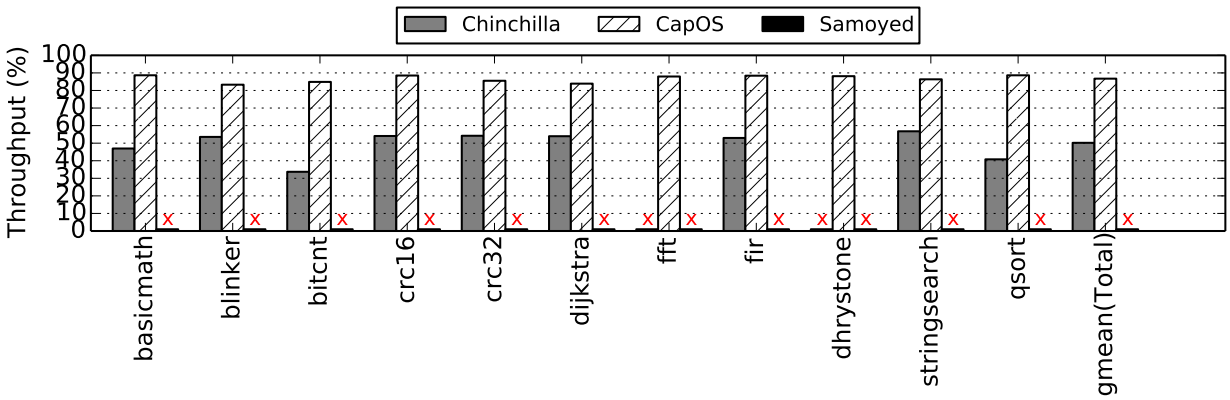


Figure 7.7. Normalized throughput of each recovery scheme compared to Samoyed with a degraded capacitor.

throughput, 99%. This is because, when CapOS runs in normal mode, it does not track memory updates with a MPU but just checkpoint the duplicated PC as an ACK (Section 7.3.1).

Performance Analysis With a Degraded Capacitor: We continued to conduct our experiments (Section 7.5.3) until the capacitor error occurs. We found that CapOS could make a forward progress while Samoyed failed due to the JIT checkpoint failure within 10 days. Chinchilla also failed to run some applications (i.e., *dhrystone and fft*) due to the capacitor error as shown in Fig. 7.7, since they consisted of long regions—that are vulnerable to the stagnation problem—as we discussed in Section 7.5.2; Chinchilla could run other applications without causing the problem.

Once we detected the capacitor error, we measured the throughput of each scheme in the same way of Section 7.5.3, i.e., $\frac{\#.of.comp}{24hours}$. Overall, CapOS and Chinchilla shows 88% and 51% throughput on average compared to the same baseline (Section 7.5.3). In particular, CapOS shows the best performance. This is because CapOS switches back to normal mode once the degraded capacitor is recovered. To analyze the performance benefit of mode change, we also measured the throughput of CapOS in safe mode. It turned out that CapOS caused about 38% throughput on average in safe mode, i.e., CapOS may cause poor performance when the capacitor is not used in the first place. However, CapOS ran in safe mode for about 2∼3 hours of a day, i.e., less than 16% of a day, and turned back to normal mode. Furthermore, we found that the capacitor is not degraded again quickly (this will be discussed in Section 7.5.4 more detail). Thanks to the short safe mode, CapOS could achieve a high throughput on average.

Two-step Capacitor Diagnosis: For mode change, CapOS first waits for the recovery period and then checks the capacitor condition with the dummy JIT checkpointing (i.e., two-step mechanism as discussed in Section 7.3.3). We measured the accuracy of the mechanism by counting the number of dummy JIT checkpointing and the number of mode change from safe to normal. We found that CapOS could always switch back from safe to normal whenever it examined the capacitor with the dummy JIT checkpointing. This is because CapOS isolated the capacitor for a longer time than the recovery period thanks to the conservative recovery time measurement (Section 7.4.2).

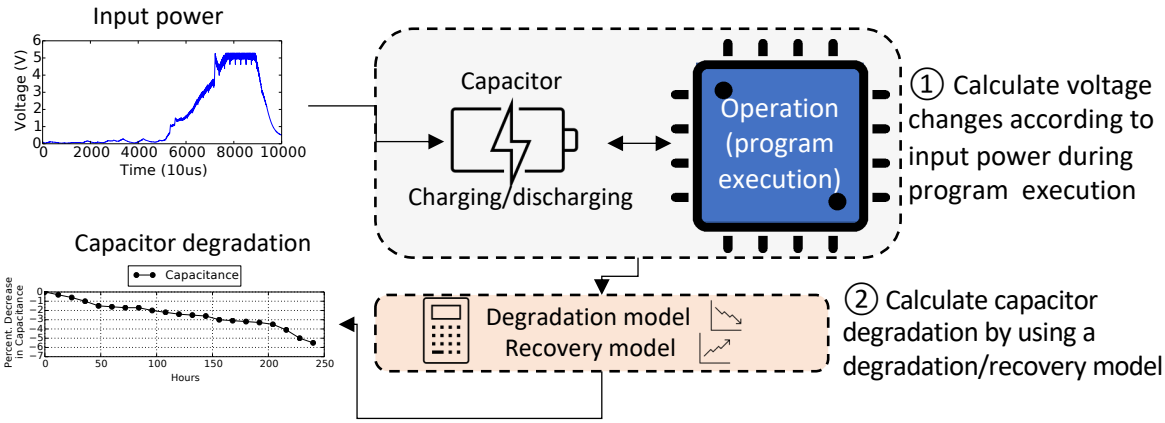


Figure 7.8. Overview of Capacitor Aging Simulator (CapSim)

7.5.4 Sensitivity Analysis

For sensitivity analysis, we varied the power failure patterns by using various power traces and estimated the performance of CapOS comparing to possible solutions such as (1) Samoyed with a proactive capacitor isolation (Section 7.2.2) and (2) Samoyed with Chinchilla. First, to analyze the impact of proactive capacitor isolation, we enabled the periodic isolation on top of Samoyed, i.e., Samoyed periodically isolates a capacitor for capacitor restoration. Second, to explore another alternative solution, we implemented Samoyed with Chinchilla; the solution runs the Chinchilla-assisted binaries but enables JIT checkpointing as long as the capacitor is not degraded. If the capacitor is degraded, the solution disables the JIT checkpointing in the same way of CapOS.

To conduct the sensitivity analysis, we implemented a capacitor aging simulator (called CapSim) on top of energy harvesting system simulator, NVPsim [25], modeling a single core in-order

processor with ARM ISA. For evaluation, we equipped the CapSim with a 1mF capacitor as energy buffer and disabled a cache in its memory hierarchy (Section 2.1). Then, we ran each scheme (i.e., Samoyed, Samoyed with a proactive capacitor isolation, Samoyed with Chinchilla, and CapOS) with the simulator to analyze its lifespan and throughput.

Capacitor Aging Simulator (CapSim): CapSim measures the lifespan of the system in a two step manner as shown in Fig. 7.8. First, CapSim takes a power trace as an input power source and runs a program charging/discharging a capacitor in NVPsim (gray dashed line box). In this step, CapSim dynamically updates capacitor condition according to input power during program execution. Second, as the condition of capacitor changes, CapSim measures the capacitor degradation by considering both the capacitor degradation model and recovery model.

Realistic Environment: For realistic evaluation, we used 12 different power traces (as shown in Table 7.2) collected from a real RF energy harvester; 10 traces from Mementos [95] and 2 traces from NVPSim [25]. Each power trace causes a different power failure pattern that affects the capacitor condition and degradation rate. In different power failure traces, CapSim measures the capacitor degradation. And, when the capacitor is 20% degraded, CapSim reports that the system is unavailable and considered as dead.

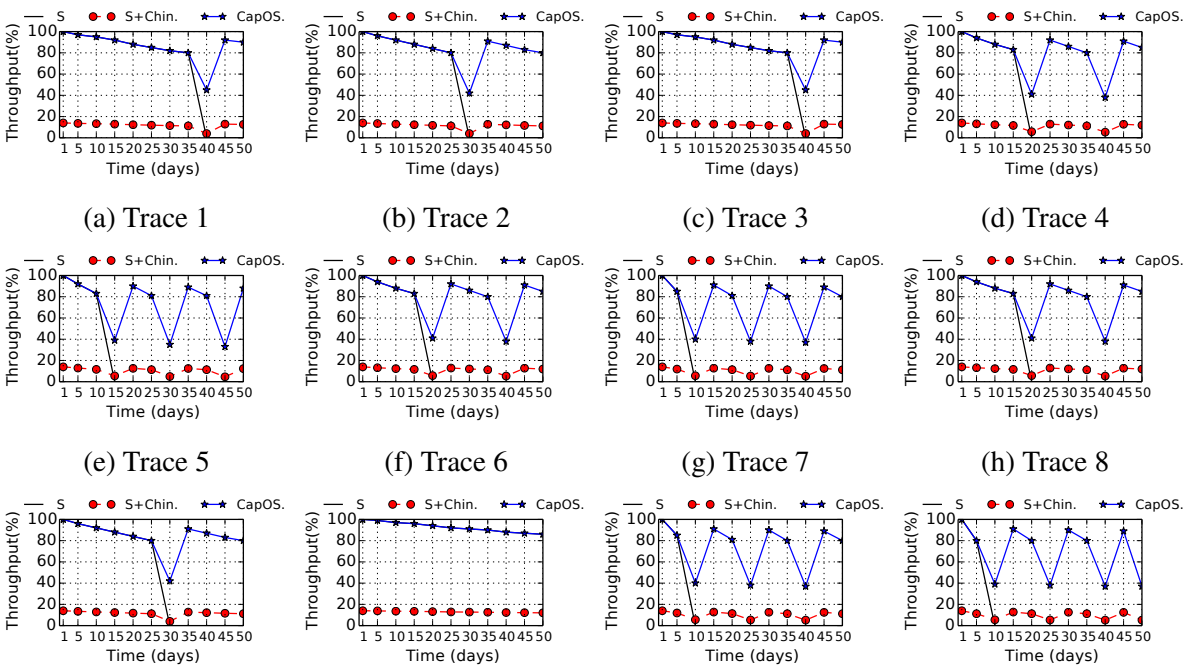
Table 7.2. The number of power failures per second in traces

Trace	1	2	3	4	5	6	7	8	9	10	11	12
# of P.F (s)	2	3	2	4	5	4	8	4	3	1	9	12

Table 7.3. Lifespan analysis (days) for energy harvesting systems varying power traces

scheme/trace	1	2	3	4	5	6	7	8	9	10	11	12
S	34	23	34	17	13	17	8	17	23	69	7	5
S+P(5000)	104	69	104	52	41	52	26	52	69	208	23	17
S+P(1000)	530	352	530	265	209	265	133	265	352	1061	117	87
CapOS	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Proactive Capacitor Recovery Solution Analysis: We measured the lifespan of the original Samoyed (S), Samoyed with proactive capacitor isolation (S+P), and CapOS; we set the original Samoyed as the baseline. For the proactive isolation, we also varied the isolation cycle as 1000



(a) Trace 1 (b) Trace 2 (c) Trace 3 (d) Trace 4

(e) Trace 5 (f) Trace 6 (g) Trace 7 (h) Trace 8

(i) Trace 9 (j) Trace 10 (k) Trace 11 (l) Trace 12

Figure 7.9. Sensitivity analysis varying power failure patterns and recovery solutions

and 5000 cycles, e.g., Samoyed stops program execution for capacitor recovery by considering the recovery model at every 1000 or 5000 running cycles.

Table 7.3 summarizes results of the evaluation. We discover that original Samoyed becomes unavailable within at least 35 days in all power traces except for trace 10. Also, we observe that the proactive capacitor isolation mechanism with Samoyed can increase the lifespan by more than 10x compared to the baseline. However, although the proactive mechanism could increase the lifespan by having a short isolation interval, e.g., 1000 capacitor charge cycles, the capacitor is eventually degraded, implying that the system can be corrupted causing a wrong recovery problem within 3 years.

Reactive Capacitor Recovery Solution Analysis: Unlike the proactive mechanism, the reactive mechanism can ultimately achieve an infinite lifespan without causing the data loss or stagnation problem regardless of the capacitor condition. To analyze the performance of alternative solutions, we measured the throughput of Samoyed (S), Samoyed with Chinchilla (S+Chin.), and CapOS in the same environment; we set the original Samoyed throughput without a degraded capacitor as the baseline.

Fig. 7.9 shows the overall performance overhead of each solution in different power traces on average. In most of power traces, while Samoyed (S) is unavailable after the capacitor is degraded, the reactive solutions such as S+Chin., and CapOS can continue to make a forward progress. However, we found that Samoyed could continue to provide a service for a while without any problem in trace 10—though Samoyed with Chinchilla unnecessarily cause significant performance overhead in trace 10.

Although the reactive solutions can make a forward progress protecting the capacitor, their performance can be varied depending on which solution is selected. As shown in the figure, the performance gap between CapOS and Chinchilla is significant. In particular, when the capacitor is in idle for recovery (that is the moment where the naive Samoyed can be corrupted), Samoyed with Chinchilla causes less than 10% throughput on average. Unlike the prior works, CapOS shows about 40% throughput on average, i.e., 4x improvement.

7.6 Summary

We observe that current energy harvesting systems can lose their volatile data due to capacitor errors. To address the problem, we introduce an OS-driven solution called CapOS. By leveraging the unique properties and operating characteristics of a capacitor in energy harvesting systems, CapOS can detect the capacitor errors and safely restore it. Our experiments demonstrate that CapOS can successfully address the capacitor fault causing almost 0% performance overhead.

8. CONCLUSION

Energy harvesting systems have emerged as an alternative to battery-operated embedded devices. Due to the intermittent nature of energy harvesting, researchers propose power failure recovery solutions that can ensure correct output across frequent power failure. However, many solutions are impractical, limiting the wide adoption. In this thesis, we study and address common problems of power failure recovery solutions, helping designers improve service quality.

First, we study software recovery solutions and address their run-time overhead problem (PS1). To reduce run-time overhead, we leverage compiler support with hardware available in commodity hardware (TS1). We study the limitation of software recovery solutions, such as statically-defined region boundaries that entails energy-consuming checkpoint stores. To address the limitation, we design Elastin, a boundary-free recovery solution with the full potential of checkpoint adaptation. The experimental results show that Elastin improves the performance by about 3.5X compared to the state-of-the-art software recovery solution. We also study another limitation of software recovery solutions: they must roll back to a recovery point and re-execute the exact instructions across power failure. Considering the limitation, we design RockClimb, a rollback-free and memory-log-free intermittent computation scheme, by leveraging compiler, run-time system, and hardware support. Our evaluation shows that RockClimb can outperform the state-of-the-art by 8X on average without requiring hardware modification costs.

Second, we study hardware-based recovery solutions and address their energy efficiency and expensive modification cost problem (PS2). To address the problem, we propose to reuse existing hardware components such as store buffer (TS2), reducing hardware cost yet improving performance. We present an architecture/compiler co-design recovery solution called CoSpec, enabling new speculative execution (power failure speculation) and ILP execution on top of the in-order processor for performance improvement. The evaluation shows that CoSpec achieves 2$\sim$3X speed-up compared to the state-of-the-art hardware-based recovery solution without requiring hardware modification. Furthermore, we present a new cache architecture called WLCache. With the help of run-time system support, WLCache can benefit from write-through and write-back policies while avoiding their downsides. Our evaluation tells that WLCache can improve performance by 4X on average while reducing hardware cost by 90% compared to the state-of-the-art design.

Finally, we study a reliability problem in EHS devices (PS3), i.e., the capacitor error. We discover the capacitor error where energy harvesting systems corrupt/lose their data or fail to provide any service when their energy buffer (i.e., capacitor) is degraded. To address the error, we propose an OS-driven capacitor error resilience solution called CapOS that can preserve volatile data against capacitor errors and recover the degraded capacitor by leveraging its self-recovery nature. Our experiments demonstrate that CapOS causes less than 1% performance overhead on average compared to an (unprotected) roll-forward recovery when there is no capacitor error. In the presence of capacitor errors, CapOS incurs only 15% performance overhead on average.

REFERENCES

[1] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 514–530, 2016.

[2] K. MAENG, A. COLIN, and B. LUCIA, "Alpaca: Intermittent execution without checkpoints," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2017.

[3] 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 *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, IEEE, 2014, pp. 330–335.

[4] J. V. D. Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, 2016, pp. 17–32, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude.

[5] M. Hicks, "Clank: Architectural support for intermittent computation," in *In Proceedings of ISCA '17*, ACM, 2017.

[6] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2019, pp. 1101–1116.

[7] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1005–1021.

[8] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2019, pp. 331–344.

[9] J. Choi, Q. Liu, and C. Jung, "Cospec: Compiler directed speculative intermittent computation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2019, pp. 399–412.

[10] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: ACM, 2012, pp. 475–486, ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254120. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254120.

[11] S. S. Baghsorkhi and C. Margiolas, "Automating efficient variable-grained resiliency for low-power iot systems," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ACM, 2018, pp. 38–49.

[12] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, 2018, pp. 129–144, ISBN: 978-1-931971-47-8. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maeng.

[13] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *Proceedings of the 27th International Conference on Compiler Construction*, ACM, 2018.

[14] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, IEEE, 2012, pp. 149–152.

[15] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 150.

[16] Y. Lui, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, J. Sampson, Y. Xie, J. Shu, and H. Yang, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15, San Francisco, CA: ACM, 2015, ISBN: 978-1-4503-3520-1.

[17] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. T. Kandemir, and V. Narayanan, "Incidental computing on iot nonvolatile processors," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2017, pp. 204–218.

[18] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, 2014.

[19] A. Teverovsky, "Insulation resistance and leakage currents in low-voltage ceramic capacitors with cracks," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 2014.

[20] R. Shigeta, T. Sasaki, D. M. Quan, Y. Kawahara, R. J. Vyas, M. M. Tentzeris, and T. Asami, "Ambient rf energy harvesting sensor device with capacitor-leakage-aware duty cycle control," 8, vol. 13, IEEE, 2013, pp. 2973–2983.

[21] F. Su, Y. Liu, Y. Wang, and H. Yang, "A ferroelectric nonvolatile processor with 46μ s system-level wake-up time and 14μ s sleep time for energy harvesting applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 3, pp. 596–607, 2017.

[22] C. E. Herdt and C. P. de Araujo, "Analysis, measurement, and simulation of dynamic write inhibit in an nvsram cell," *IEEE transactions on electron devices*, vol. 39, no. 5, pp. 1191–1196, 1992.

[23] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 10–22, 1992.

[24] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[25] Y. Gu, Y. Liu, Y. Wang, H. Li, and H. Yang, "Nvpsim: A simulator for architecture explorations of nonvolatile processors," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 147–152.

[26] M. Xie, C. Pan, Y. Zhang, J. Hu, Y. Liu, and C. J. Xue, "A novel stt-ram-based hybrid cache for intermittently powered processors in iot devices," *IEEE Micro*, vol. 39, no. 1, pp. 24–32, 2018.

[27] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, and J. Hu, "Checkpoint aware hybrid cache architecture for nv processor in energy harvesting powered systems," in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, IEEE, 2016, pp. 1–10.

[28] A. Gupta, O. P. Yadav, D. DeVoto, and J. Major, "A review of degradation behavior and modeling of capacitors," in *ASME 2018 International Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Microsystems*, American Society of Mechanical Engineers Digital Collection, 2018.

[29] R. Chaari, O. Briat, and J.-M. Vinassa, "Capacitance recovery analysis and modelling of supercapacitors during cycling ageing tests," *Energy conversion and management*, vol. 82, pp. 37–45, 2014.

[30] C. Kulkarni, G. Biswas, J. Celaya, K. Goebel, and N. SGT, "Prognostic techniques for capacitor degradation and health monitoring," in *The Maintenance& Reliability Conference, MARCON*, 2011.

[31] C. S. Kulkarni, *A physics-based degradation modeling framework for diagnostic and prognostic studies in electrolytic capacitors*. Vanderbilt University, 2013.

[32] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1085–1100.

[33] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2018.

[34] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, IEEE, 2013, pp. 1–12.

[35] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 140–151.

[36] A. M. Imam, D. M. Divan, R. G. Harley, and T. G. Habetler, "Electrolytic capacitor failure mechanism due to inrush current," in *2007 IEEE Industry Applications Annual Meeting*, IEEE, 2007, pp. 730–736.

[37] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2022, pp. 40–54.

[38] *Msp430fr5994launchpad development kit (mspexp430fr5994)*, Mar. 2016. [Online]. Available: http://www.ti.com/lit/ug/slau678a/slau678a.pdf.

[39] K. Ma, X. Li, S. R. Srinivasa, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, IEEE, 2017, pp. 678–683.

[40] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, IEEE, 2015, pp. 526–537.

[41] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *LIPIcs-Leibniz International Proceedings in Informatics*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 71, 2017.

[42] P. Thanigai and W. Goh, "Msp430 fram quality and reliability," *Texas Instruments, SLAA526A*, 2014.

[43] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. Xue, "Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor," in *Proceedings of The 52nd IEEE/ACM Design Automation Conference (DAC 2015)*, ser. DAC '15, San Francisco, CA: ACM, 2015.

[44] W. Zhang, S. Liu, M. Lv, Q. Chen, and N. Guan, "Intermittent computing with efficient state backup by asynchronous dma," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 543–548.

[45] J. Jeong and C. Jung, "Pmem-spec: Persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 517–529.

[46] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 525–538.

[47] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight soft error resilience for in-order cores," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 654–666.

[48] S. Liu, W. Zhang, M. Lv, Q. Chen, and N. Guan, "Latics: A low-overhead adaptive task-based intermittent computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3711–3723, 2020.

[49] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," 2, vol. 16, ACM, 2016, p. 32.

[50] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, ser. LCTES'15, Portland, OR, USA: ACM, 2015, 2:1–2:10, ISBN: 978-1-4503-3257-6. DOI: 10.1145/2670529.2754959. [Online]. Available: http://doi.acm.org/10.1145/2670529.2754959.

[51] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, IEEE, 2016, pp. 228–239.

[52] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[53] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, IEEE, 2016, pp. 1–6.

[54] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "Replaycache: Enabling volatile cachesfor energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 170–182.

[55] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight gpu register file protection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 989–1004.

[56] H. Song, S. Kim, J. H. Kim, E. J. Park, and S. H. Noh, "First responder: Persistent memory simultaneously as high performance buffer cache and storage," in *2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021, pp. 839–853.

[57] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "Slm-db: Single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies FAST'19)*, 2019, pp. 191–205.

[58] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "Ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 258–270.

[59] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *11th USENIX Conference on File and Storage Technologies (FAST13)*, 2013, pp. 73–80.

[60] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems.," in *FAST*, vol. 12, 2012.

[61] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 91–104, 2017.

[62] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvm," *ACM Transactions on Storage (TOS)*, vol. 13, no. 2, pp. 1–13, 2017.

[63] H. R. Mendis, C.-K. Kang, and P.-c. Hsiu, "Intermittent-aware neural architecture search," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–27, 2021.

[64] N. A. Bhatti and L. Mottola, "Harvos: Efficient code instrumentation for transiently-powered embedded sensing," in *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, IEEE, 2017, pp. 209–220.

[65] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[66] A. Colin, G. Harvey, A. P. Sample, and B. Lucia, "An energy-aware debugger for intermittently powered systems," *IEEE Micro*, vol. 37, no. 3, pp. 116–125, 2017.

[67] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 577–589, 2016.

[68] B. Islam and S. Nirjon, "Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2020, pp. 95–109.

[69] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu, "Enabling failure-resilient intermittent systems without runtime checkpointing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4399–4412, 2020.

[70] W.-M. Chen, P.-C. Hsiu, and T.-W. Kuo, "Enabling failure-resilient intermittently-powered systems without runtime checkpointing," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2019, pp. 1–6.

[71] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, and T.-W. Kuo, "Multiversion concurrency control on intermittent systems.," in *ICCAD*, 2019, pp. 1–8.

[72] S. Lee, B. Islam, Y. Luo, and S. Nirjon, "Intermittent learning: On-device machine learning on intermittently powered system," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 4, pp. 1–30, 2019.

[73] P. Cronin, C. Yang, and Y. Liu, "Reliability and security in non-volatile processors, two sides of the same coin," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2018, pp. 112–117.

[74] C. S. Kulkarni, J. R. Celaya, G. Biswas, and K. Goebel, "Accelerated aging experiments for capacitor health monitoring and prognostics," in *2012 IEEE AUTOTESTCON Proceedings*, IEEE, 2012, pp. 356–361.

[75] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A lightweight kernel for nvram-based transiently-powered systems," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1390–1403, 2018.

[76] L. Zhang, X. Hu, Z. Wang, F. Sun, and D. G. Dorrell, "A review of supercapacitor modeling, estimation, and applications: A control/management perspective," *Renewable and Sustainable Energy Reviews*, vol. 81, pp. 1868–1878, 2018.

[77] W. Chen, Y. Wang, J. Dai, S. Lu, X. Wang, P. Lee, H. Chan, and C. Choy, "Spontaneous recovery of hydrogen-degraded tio 2 ceramic capacitors," *Applied physics letters*, vol. 84, no. 1, pp. 103–105, 2004.

[78] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Sense your power: The eco approach to energy awareness for iot devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 3, pp. 1–25, 2021.

[79] J. De Winkel, V. Kortbeek, J. Hester, and P. Pawełczak, "Battery-free game boy," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 4, no. 3, pp. 1–34, 2020.

[80] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 939–954.

[81] H. R. Mendis and P.-C. Hsiu, "Accumulative display updating for intermittent systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[82] F. Yang, A. S. Thangarajan, S. Michiels, W. Joosen, and D. Hughes, "Morphy: Software defined charge storage for the iot," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 248–260.

[83] M. Katanbaf, A. Saffari, and J. R. Smith, "Multiscatter: Multistatic backscatter networking for battery-free sensors," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021.

[84] E. C. Lab, *Wearable technology and the internet of things*, 2016.

[85] C. Perera, C. H. Liu, and S. Jayawardena, "The emerging internet of things marketplace from an industrial perspective: A survey," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 4, pp. 585–598, 2015.

[86] S. Mauilk, *Wearables and internet of things 2015*, 2015.

[87] M. S. Reserach, *Wearable devices: The internet of things becomes personal*, 2014.

[88] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2015, pp. 575–585.

[89] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs.," in *In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2015, pp. 514–530.

[90] A. C. Kiwan Maeng and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. ACM Program. Lang.1, OOPSLA, Article 96*, Oct. 2017.

[91] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: Low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 398–409.

[92] Q. Liu, C. Jung, D. Lee, and D. Tiwarit, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–12.

[93] M. Xie, C. Pan, J. Hu, C. Yang, and Y. Chen, "Checkpoint-aware instruction scheduling for nonvolatile processor with multiple functional units," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, IEEE, 2015, pp. 316–321.

[94] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. J. Xue, "Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 184.

[95] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *Acm Sigplan Notices*, vol. 47, no. 4, pp. 159–170, 2012.

[96] Q. Li, M. Zhao, J. Hu, Y. Liu, Y. He, and C. J. Xue, "Compiler directed automatic stack trimming for efficient non-volatile processors," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 183.

[97] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *2012 Proceedings of the ESSCIRC*, ser. ESSCIRC '12, Bourdeaux, France: IEEE Press, 2012, pp. 149–152.

[98] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Porland, OR: ACM, 2015, pp. 575–585.

[99] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective Plus MasteringEngineering with Pearson eText – Access Card Package*, 3rd. Pearson, 2015, ISBN: 0134123832, 9780134123837.

[100] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 165–176, 2004.

[101] C. Jung, D. Lim, J. Lee, and S. Han, "Adaptive execution techniques for smt multiprocessor architectures," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, 2005, pp. 236–246.

[102] J. Lee, J.-H. Park, H. Kim, C. Jung, D. Lim, and S. Han, "Adaptive execution techniques of parallel programs for multiprocessors," *J. Parallel Distrib. Comput.*, vol. 70, no. 5, pp. 467–480, May 2010, ISSN: 0743-7315.

[103] M. Chen, K. K. Afridi, and D. J. Perreault, "Stacked switched capacitor energy buffer architecture," *IEEE Transactions on Power Electronics*, vol. 28, no. 11, pp. 5183–5195, 2013.

[104] X. Wang, D. M. Vilathgamuwa, and S. S. Choi, "Determination of battery storage capacity in energy buffer for wind farm," *IEEE Transactions on Energy Conversion*, vol. 23, no. 3, pp. 868–878, 2008.

[105] *Wisp5 wiki*, https://wisp5.wikispaces.com/WISP+Home, 2017.

[106] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "Ido: Compiler-directed failure atomicity for nonvolatile memory," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 258–270. DOI: 10.1109/MICRO.2018.00029. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00029.

[107] C. Pan, M. Xie, J. Hu, Y. Chen, and C. Yang, "3m-pcm: Exploiting multiple write modes mlc phase change main memory in embedded systems," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, ACM, 2014, p. 33.

[108] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging pcm lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *Proceedings of the 2014 international symposium on Low power electronics and design*, ACM, 2014, pp. 327–330.

[109] T. Wang, D. Liu, Z. Shao, and C. Yang, "Write-activity-aware page table management for pcm-based embedded systems," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, IEEE, 2012, pp. 317–322.

[110] H. A. Khouzani, C. Yang, and J. Hu, "Improving performance and lifetime of dram-pcm hybrid main memory through a proactive page allocation strategy," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, IEEE, 2015, pp. 508–513.

[111] M. Zhao, Y. Xue, C. Yang, and C. J. Xue, "Minimizing mlc pcm write energy for free through profiling-based state remapping," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, IEEE, 2015, pp. 502–507.

[112] C. Liu and C. Yang, "Improving multilevel pcm reliability through age-aware reading and writing strategies," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, IEEE, 2014, pp. 264–269.

[113] M. Zhao, L. Shi, C. Yang, and C. J. Xue, "Leveling to the last mile: Near-zero-cost bit level wear leveling for pcm-based main memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, IEEE, 2014, pp. 16–21.

[114] C. Wang and S. Chattopadhyay, "Lawn: Boosting the performance of nvmm file system through reducing write amplification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[115] C.-C. Ho, Y.-M. Chang, Y.-H. Chang, H.-C. Chen, and T.-W. Kuo, "Write-aware memory management for hybrid slc-mlc pcm memory systems," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 2, pp. 16–26, 2017.

[116] T.-Y. Chen, Y.-H. Chang, S.-H. Chen, C.-C. Kuo, M.-C. Yang, H.-W. Wei, and W.-K. Shih, "Wrjfs: A write-reduction journaling file system for byte-addressable nvram," *IEEE Transactions on Computers*, 2018.

[117] *Maximizing write speed on the msp430™ fram*, Accessed: 2018-10-14, Feb. 2015. [Online]. Available: http://www.ti.com/mcu/docs/.

[118] A. Sinha and A. P. Chandrakasan, "Jouletrack-a web based tool for software energy profiling," in *In Proceedings of the 38nd Annual Design Automation Conference*, ser. DAC '01, 2001.

[119] *Msp430fr59xx mixed-signal microcontrollers (rev. f)*, Mar. 2017. [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[120] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[121] M. Hicks, *Thumbulator: Cycle accurate armv6-m instruction set simulator*, 2016.

[122] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, IEEE, 2001, pp. 3–14.

[123] Q. Liu, X. Wu, L. Kittinger, M. Levy, and C. Jung, "Benchprime: Effective building of a hybrid benchmark suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 179, 2017.

[124] P. Cronin, C. Yang, D. Zhou, K. Qiu, X. Shi, and Y. Liu, "'the danger of sleeping', an exploration of security in non-volatile processors," in *Hardware Oriented Security and Trust Symposium (AsianHOST), 2017 Asian*, IEEE, 2017, pp. 121–126.

[125] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '15, Burlingame, CA: IEEE Press, 2015, pp. 526–537.

[126] X. Zhang, C. Patterson, Y. Liu, C. Yang, C. J. Xue, and J. Hu, "Low overhead online checkpoint for intermittently powered non-volatile fpgas," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2018, pp. 238–244.

[127] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Fast and energy-efficient state checkpointing for intermittent computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–27, 2020.

[128] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.

[129] J. R. Smith, *Wirelessly Powered Sensor Networks and Computational RFID*. New York, NY, USA: Springer, 2013.

[130] M. Karimi, H. Choi, Y. Wang, Y. Xiang, and H. Kim, "Real-time task scheduling on intermittently-powered batteryless devices," *IEEE Internet of Things Journal*, 2021.

[131] J. San Miguel, K. Ganesan, M. Badr, and N. E. Jerger, "The eh model: Analytical exploration of energy-harvesting architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 76–79, 2018.

[132] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multicore platforms," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 124, 2014.

[133] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Precise micro-architectural modeling for wcet analysis via ai+ sat," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2013, pp. 87–96.

[134] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *2011 IEEE 32nd Real-Time Systems Symposium*, IEEE, 2011, pp. 339–348.

[135] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[136] T. Instruments, *Msp430 family instruction set summary*, 2006.

[137] J.-M. Chang and M. Pedram, "Register allocation and binding for low power," in *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, ACM, 1995, pp. 29–35.

[138] T. Instruments, *Msp430fr family of ultra low-power microcontrollers*, 2015.

[139] *Powercast hardware*. [Online]. Available: http://www.powercastco.com..

[140] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, 2018, pp. 41–53.

[141] J. de Winkel, C. Delle Donne, K. S. Yildirim, P. Pawełczak, and J. Hester, "Reliable timekeeping for intermittent computing," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 53–67.

[142] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 85–99.

[143] H. G. Lee and N. Chang, "Powering the iot: Storage-less and converter-less energy harvesting," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, IEEE, 2015, pp. 124–129.

[144] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[145] Y. Wang, Y. Liu, C. Wang, Z. Li, X. Sheng, H. G. Lee, N. Chang, and H. Yang, "Storage-less and converter-less photovoltaic energy harvesting with maximum power point tracking for internet of things," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 2, pp. 173–186, 2016.

[146] Y. Xie, *EMERGING MEMORY TECHNOLOGIES*. Springer, 2016.

[147] S. Mehta and J. Torrellas, "Wearcore: A core for wearable workloads?" In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, IEEE, 2016, pp. 153–164.

[148] *Arm research starter kit: System modeling using gem5*, Accessed: 2018-11-18, Jul. 2017.

[149] C. Wang and Y. Wu, "Tso_atomicity: Efficient hardware primitive for tso-preserving region optimizations," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 509–520, 2013.

[150] E. Gunadi and M. H. Lipasti, "A position-insensitive finished store buffer," in *Computer Design, 2007. ICCD 2007. 25th International Conference on*, IEEE, 2007, pp. 105–112.

[151] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[152] N. Vedula, A. Shriraman, S. Kumar, and W. N. Sumner, "Nachos: Software-driven hardware-assisted memory disambiguation for accelerators," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 710–723.

[153] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ACM, 2016, pp. 265–266.

[154] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," 2001.

[155] T. Instruments, *Msp430fr59xx mixed-signal microcontrollers*, 2017.

[156] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[157] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30, Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 330–335, ISBN: 0-8186-7977-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=266800.266832.

[158] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015, pp. 476–488.

[159] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, IEEE, 2016, pp. 1–6.

[160] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2017, pp. 318–329.

[161] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015, pp. 309–319.

[162] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie, "Persistence parallelism optimization: A holistic approach from memory bus to rdma network," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[163] S. Priya and D. J. Inman, *Energy harvesting technologies*. Springer, 2009, vol. 21.

[164] J. Hester, K. Storer, L. Sitanayah, and J. Sorber, "Towards a language and runtime for intermittently-powered devices," *sleep*, vol. 9, p. 10, 2016.

[165] E. Nwafor, A. Campbell, D. Hill, and G. Bloom, "Towards a provenance collection framework for internet of things devices," in *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, IEEE, 2017, pp. 1–6.

[166] S. Nirjon, "Lifelong learning on harvested energy," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ACM, 2018, pp. 500–501.

[167] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial internet of things: Challenges, opportunities, and directions," *IEEE Transactions on Industrial Informatics*, 2018.

[168] M. A. De Kruijf, "Compiler construction of idempotent regions and applications in architecture design," Ph.D. dissertation, Madison, WI, USA, 2012.

[169] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback-a technique for improving bandwidth utilization," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 11–21.

[170] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, *et al.*, "Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 368–381.

[171] C.-K. Kang, C.-H. Lin, P.-C. Hsiu, and M.-S. Chen, "Homerun: Hw/sw co-design for program atomicity on self-powered intermittent systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.

[172] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Eco: A hardware-software co-design for in situ power measurement on low-end iot systems," in *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, 2019, pp. 22–28.

[173] L. Sigrist, A. Gomez, R. Lim, S. Lippuner, M. Leubin, and L. Thiele, "Measurement and validation of energy harvesting iot devices," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 1159–1164.

[174] W.-M. Chen, T.-S. Cheng, P.-C. Hsiu, and T.-W. Kuo, "Value-based task scheduling for non-volatile processor-based embedded devices," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2016, pp. 247–256.

[175] D. B. Murray and J. G. Hayes, "Cycle testing of supercapacitors for long-life robust applications," *IEEE Transactions on Power Electronics*, vol. 30, no. 5, pp. 2505–2516, 2014.

[176] T. Hardin, R. Scott, P. Proctor, J. Hester, J. Sorber, and D. Kotz, "Application memory isolation on {ultra-low-power}{mcus}," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[177] G. Berthou, K. Marquet, T. Risset, and G. Salagnac, "Mpu-based incremental checkpointing for transiently-powered systems," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, IEEE, 2020, pp. 89–96.

[178] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu, "{tardis}: Time and remanence decay in {sram} to implement secure protocols on embedded devices without clocks," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 221–236.

VITA

Jongouk Choi received his Master of Science (MS) and Bachelor of Science (BS) in Computer Science from Kentucky State University. His primary interests are computer architecture and compiler.