Delft University of Technology
Master of Science Thesis in Embedded Systems

Efficient Memory Architecture for Next Generation Low-Power Embedded Systems

Sourav Mohapatra

Embedded Networked Systems

TU Delft Delft University of Technology

Efficient Memory Architecture for Next Generation Low-Power Embedded Systems

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Sourav Mohapatra
29/08/2022

Author
Sourav Mohapatra
Title
Efficient Memory Architecture for Next Generation Low-Power Embedded Systems
MSc Presentation Date

29/08/2022

Graduation Committee

dr. Przemysław Pawełczak (chairman)	Delft University of Technology
dr. Zaid Al-Ars	Delft University of Technology
Vito Kortbeek	Delft University of Technology

The work presented in this thesis has lead to a paper which will be submitted to a conference for publication.

Abstract

In this thesis we propose a novel memory architecture design that is robust to frequent memory failures targeting next generation low power embedded system. We explore the how the architecture works and perform detailed evaluations to show that our system achieves better performance than the state-of-the-art.

“Premature optimization is the root of all evil’ – Donald Knuth

Preface

There have been many advances in the computing sector. A fingertip-sized processor now has more computing capabilities than a full-room-sized computer 30 years ago. But the same cannot be said for the power management sectors. To power that fingertip-sized processor, we still need a large block of battery or a constant power supply. This dependency on the power supply can be ignored for many devices, but when we venture into the IoT domain, it plays a significant role. If IoT devices must depend on on-device power sources, the usability decreases and the environmental impact increases a lot. Battery-based systems require regular maintenance and are more difficult to dispose of. The harvesting of energy from nature provides a huge steppingstone for such systems.

Intermittent computing refers to a system in which the power supply is not constant. Devices that are said to run intermittently depend on sources of energy that can be harvested from the environment. Such a system requires dedicated hardware and software support to be functional. The motivation behind this work is to design the software and hardware framework for an intermittent system that supports a cache-memory hierarchy.

Words cannot express my gratitude to my daily supervisor Vito and my professor dr. Przemysław and for their invaluable patience and wonderful feedback. I would also like to thank my defense committee, who generously provided knowledge and expertise.

It would be remiss not to mention my amazing friends who gave me a social circle where I could enjoy my time at TU Delft. I would especially like to thank the embedded boys for being there for everything. A special mention to all the Villagers of Bengland for the amazing trips, dinners, barbeques, and fun times. I could not have undertaken this journey without these people.

I would like to mention my parents, who are there to always be my support system. Their belief in me has kept my spirits and motivation high during this process. Lastly, I would express my greatest gratitude to Swati, who listened, guided, helped, and supported me in all aspects of my journey.

Sourav Mohapatra

Delft, The Netherlands
26th August 2022

Contents

Chapter 1

Introduction

With the current boom in the Internet of Things (IoT) sector, accelerated by the ramping up of 5G deployment [66], the number of embedded IoT devices is increasing rapidly [9]. As the number of IoT devices increases, the concern for the sustainability of their deployment also increases [62]. Batteries are one of the primary artifacts in reducing the sustainability of devices [72]. They must be replaced every fixed period [22], they are a fire hazard [25] in deployment scenarios such as forests and are a burden on the environment when not properly disposed of [26]. Batteries also increase their temperature as the load increases [35] preventing them from performing computationally intensive tasks in heat-sensitive environments (think of a smart contact lens [36]). Batteries have limitations in evolving space technologies also where nano-satellites are removing them for consuming extra volume and low stability [20]. A potential solution is to completely renounce batteries and use a system in which energy harvested [65] from ambient sources is the sole power supply.

Intermittently-powered devices [58] use energy harvested from natural sources such as the sun, electromagnetic radiation, geothermal energy, etc. However, since such energy sources are inherently unpredictable and result in random power failures [56], the system must recover and continue to make progress. Consider an example system [64, Section 3A] with an energy harvesting source of $20\,\mu W/cm^3$, a capacitor of $46\,\mu F$ and a consumption energy of $1\,\mu J$. It takes 22 seconds to charge and perform a set of calculations, and then the system must harvest energy again for 22 seconds before continuing. In this cycle, the system is completely devoid of power. To become a viable alternative to a battery-based system, intermittently-powered systems must be wrapped with a layer of software and hardware support which, by saving the program progress in a Non-Volatile Memory (NVM), provides an option to recover and resume execution from the saved state in case of power failure.

NVM—since persistent—can be used to intelligently save the state of programs with mitigating actions such as backups and rollbacks of the stack, registers, and other volatile components. But copying all the volatile states is inefficient and consumes precious limited energy. An alternative is to use NVMs to replace volatile components in memory hierarchies. Various works explore this approach, for example, [61] uses a Magnetoresistive Random Access Memory (MRAM) for ultra-low power IoT devices, [70] proposes a consistency-aware checkpointing scheme, [55] develops a scheduler for Non-Volatile Processor (NVP) (which is

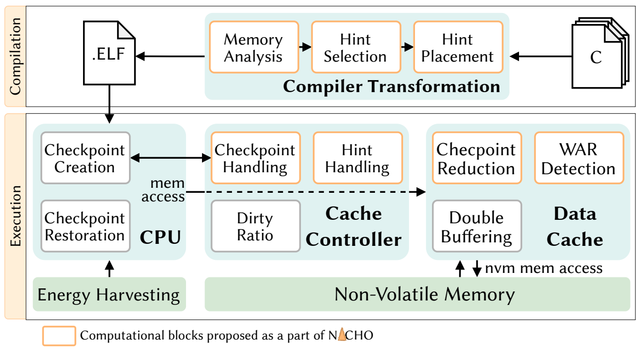


Figure 1.1: **An overview of NACHO architecture. The system is divided into two units - compilation and execution. The compilation unit creates the binary which acts as an input to the execution unit. The data cache and the cache controller implement NACHO's logic. Blocks with colored outlines are proposed as a part of NACHO**

when all the components in the system are non-volatile) and [63] proposes a custom memory hierarchy based on NVP.

However, using NVM as main memory is not straightforward; it introduces a set of consistency problems caused by Write After Read (WAR) memory accesses. Figure 1.2 explains how WARs occurs when there is read access to a memory location followed by write access to the same location. The system must ensure memory consistency by identifying and mitigating the occurrence of WAR.

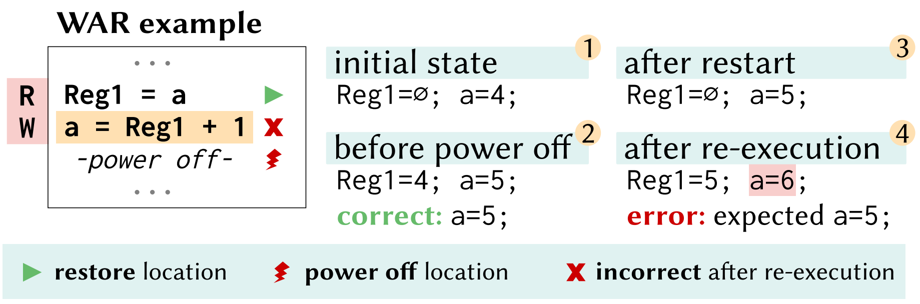


Figure 1.2: **A simple example showing how a memory sequence in which a read is followed by a write can result in an inconsistent state after a power failure. The register `Reg1` is volatile and `a` is a memory location in the non-volatile memory.**

An example of such mitigation adopted by some work [74, 47, 45, 17, 42] is the use of task-based programming constructs. The programmer divides the code into small logical blocks of computations called tasks, and each task is performed atomically whenever enough energy becomes available. Although highly efficient, these solutions require the intervention of the programmer [39, Section 5.4] and therefore cannot be used for legacy code without significant modification of the source code [68]. Also, the programmer must dimension the impotent code for a specific energy budget, which means that they need to rewrite the tasks when the target energy harvesting environment changes. Checkpoints [71] remove this dependency on the programmer. They automatically store the state of the program in non-volatile memory and restore them in the event of power failure. But, they are slower than tasks-based approach.

Some systems create checkpoints dynamically where a dedicated monitor checks the available energy for computation. If that drops below a certain threshold, a checkpoint is created. This method, termed Just In Time (JIT) checkpointing [67, 13, 12, 33] is, in essence, driven by the decreasing availability of energy. This makes it highly efficient as checkpoints are created only when needed, but they can lead to corruption. Any assurance of incorruptibility hinges on the prediction of energy consumption, which is inherently variable and can lead to system failure without proper backup [16, Section 2.2.2]. On the contrary, other systems incorporate a static method to create checkpoints. These are primarily compiler-based techniques in which the placement of checkpoints is determined during the compilation stage. Instead of energy availability, these are driven by different forms of code analysis [68, 70, 71, 76]. Unlike JIT, these systems ensure that the program does not get corrupted in any possible scenario. However, to do so, they often over-instrument the code with checkpoints. which makes them slower than JIT. Works like [30, 59], a dedicated hardware module has been introduced to perform memory tracking and control the placement of checkpoints during run-time.

To be as energy-efficient as possible, the goal is to maximize the computations for a given burst of energy. In addition to computations, the system must also create checkpoints to ensure intermittency. Few systems use different physical memory technologies such as FRAM [11], STT-RAM [40, 51], NV-SRAM [41], while others propose a form of resource-adaptive architecture, such as changes in execution pipelines to optimize and dynamically allocate power budget [46] and explore single-level cell (SLC) or multi-level cell (MLC) NVM [77]. This is especially energy-inefficient when the nature of placement is static, as the system creates checkpoints based on the code rather than on the energy availability. Both the size and the number of checkpoints contribute to this overhead.

1.1 Problem statement

However, this act of creating checkpoints consumes a lot of energy and execution time [17]. Looking at the results of a recent work [38, Figure 4], even the most optimized solution has double the execution time compared to native unmodified binaries without checkpoints. The higher the number of volatile components (such as main memory, registers, peripheral states, etc.), the greater the energy requirement for a checkpoint. For example, a volatile system consumes 445 µJ, while the non-volatile system consumes only 23.1 nJ [64] for a backup operation. However, completely sacrificing volatile memory and using only NVM, while resulting in a decrease in checkpoint energy consumption, significantly increases the cost of normal read and write operations, both in terms of energy and time [44, 34]. There is a need for a balance between volatile and non-volatile components.

Some works try to achieve this balance by reducing the size of checkpoints while still using volatile components, i.e., a mixed-memory model [39, 48, 30]. Another direction is the use of a volatile data cache, which decreases the cost of writing to the NVM. However, integrating a cache with intermittent systems is not straightforward. The system still needs to address WAR hazards, which become even more complicated in the presence of cache [75] as the cache delays the actual write-back to NVM. The cache eviction policy that determines which cache block must be evicted to make space for a new block, needs to be aware of when the checkpoint is going to happen and how to proceed whilst maintaining consistency. Furthermore, since the cache is a volatile entity, it must be written to NVM before a checkpoint. These models can also be intelligently managed, for example, by a virtual memory manager that assigns variables to volatile or non-volatile memory [49].

Traditionally, low-power embedded systems, such as those based on STM [3, 4], MSP [2], ESP [1] based, have not had a cache. Since the SRAM memories onboard these systems are already fast, having a cache does not provide significant improvements in access speed. Furthermore, caches do not work well with real-time systems [27] without specialized cache management. However, this changes when we consider caches for intermittent systems where NVMs are used. Using a small data cache between the processor and NVM leads to a reduction in NVM accesses that can provide significant energy savings. Also, since intermittent systems themselves do not work well with real-time systems [19], having a cache is no longer a deterrent but rather advantageous.

PROWL [31] uses a different replacement policy (cuckoo hashing) and the associativity of the cache (skew associative) system to decrease the number of checkpoints in the system. But using such a computationally intensive cache architecture consumes more energy, is complicated to implement, and is highly inflexible for adapting to different systems. ReplayCache [75] discusses that the "stores" are not synchronous if a system uses a data cache. So, ReplayCache "replays" all "stores" in a fixed code region after an event of power failure, thus restoring the cache state to what it was before. In our interpretation, it is a modified version of a redo log [52] that incorporates the cache. Furthermore, they only support JIT checkpointing; therefore, they cannot provide any guarantee of program incorruptibility. In a very recent work [14] the authors propose a memory renaming approach in which dirty blocks that are evicted from the cache are renamed to avoid causing idempotency violations. Although they achieve performance gains, the sacrifice comes in terms of the use of additional and dedicated hardware components and the increase in the space complexity of NVM, which outweighs the gains in our opinion.

1.2 Research Question

Based on the above discussion, our opinion is that there is still a gap between current state-of-the-art and an efficient cache-based intermittent-system. To bridge that gap, there needs to be a system that closely integrates the hardware and software support system while providing an efficient solution to handle intermittency. This leads to the research question: **How to design an efficient intermittent system without complicated hardware support systems and without the need for programmer intervention?**

1.3 Fundamental insight

To address this research question, we take a different position. We argue that a mixed-memory model with a simple addition of a cache will not solve the problem. We rethink the fundamental way in which a cache can be integrated into an intermittent system. We propose a design in which a data cache is very closely integrated with the intermittent system. **We use the data cache to detect WAR during program execution**. Furthermore, we **use the cache to detect memory patterns that increase the accuracy in detecting WARs**. Using this as the fundamental building block, we improve both performance and energy consumption while ensuring forward progress with **guarantee of incorruptibility, compatibility, and not adding any additional hardware components**.

1.4 Contributions

In this paper, we present NACHO a compiler and hardware co-design in which we tightly integrate a data cache into an intermittent system. NACHO uses a volatile cache to reduce NVM accesses and also to detect possible WAR hazards that may arise from frequent power failures. Due to the close integration with the system, NACHO mitigates the need to use additional HW components, as well as the need to modify the program binary. It detects and prevents data inconsistency by using the inherent properties of how a cache accesses memory. On top of this, we implement optional efficiency improvements by closely incorporating the compiler into the system as well. NACHO is a complete intermittent framework without the need for programmer intervention. Our solution can work with both the JIT [67] and static [68] checkpoint-based paradigms. It can be used with any form of cache architecture with any choice of cache block replacement and placement policies. Furthermore, NACHO ensures that the system remains **incorruptible**. No sequence of power availability can cause the system to deviate from the correct execution.

We can conceptualize our solution through these three novel ideas.

1. We incorporate a *data cache* and use it to detect and handle WAR hazards by *dynamically initiating checkpoints*. We also group cache evictions in such a way that they can be handled by a single checkpoint, thus reducing the number of checkpoints required.

2. Among the WAR hazards, we detect and separate “safe” and “unsafe” writes using the cache. Thus, only “unsafe” writes require a checkpoint. By doing so, we further reduce the number of checkpoints and NVM accesses.

3. We introduce “compiler hints” in which, using a specific set of compiler analysis, the cache marks certain unsafe writes as safe. These compiler hints work in tandem with the cache controller to further reduce checkpoints and NVM accesses.

To the best of our knowledge, this is the first software system that uses a hardware software co-design with a data cache for an intermittent system.

Chapter 2

Related Work

In this section, we explore how caches work with an intermittent system and provide motivation towards our research.

2.1 Existing Intermittent Systems

Hibernus [13] and Hibernus++ [12] employ specialized hardware support to monitor the energy left. Whenever it falls below a threshold, both systems react by firing an interrupt that preempts the application and forces the system to take a checkpoint. Thus, checkpoints may take place at any arbitrary point in time. Both systems copy the entire memory area, including unused or empty portions, into NVM.

MementOS [58] and HarvOS [15] employ compile time strategies to insert specialized system calls to check the energy buffer. Checkpoints occur proactively and only when the execution reaches one of these calls. During a checkpoint, every segment used in main memory is copied to NVM regardless of changes since the last checkpoint.

There do not seem to be any recent hardware and software approaches that tackle the efficiency of checkpointing systems. The most recent is So Far So Good [73] which is a software-only approach. It proposes a software-only approach where the system uses an exploration-and-achieve kind of approach. The program runs until the power goes down and then places a checkpoint sometime before the power failure point in the next run. Feels too optimistic about the possibility of performance loss.

Software only	**Software + Hardware**	**Hardware only**
So Far So Good	Clank	A 3us wakeup timer
Alpaca	Hibernus++	A NV controller
Chain	Hibernus	TCCP
Incremental Checkpoint	Mementos	
Ratchet	QuickRecall	
DINO		

Table 2.1

There is another class of approach wherein the solution focuses on approximation to overcome the intermittency in power. In What's Next [23] an approximate calculation is used in the case of power failure, which is then improved if more power is available.This contrasts with a more traditional all-or-nothing computing approach.

DICE [8] aims to improve the amount of data that are written to NVM memory. It works with existing systems, and instead of copying the full data to the NVM, it copies only what is the differential between the current and the last checkpoint. iCheck [43] addresses systems that do not even have a capacitor. They use ambient energy analysis to predict power cycles with a certain probability and, accordingly, place checkpoints.

Table 2.1 shows the various different works categorized by the nature of the solution they provide. None of the software and hardware co-design solutions are recent.

2.2 RISC-V based Intermittent Systems

Most of the intermittent research is done on ARM or MSP430-based microcontrollers. Some distantly related research publications use RISC-V to evaluate their work. In Failure Sentinels [67] a low-power voltage monitor system is developed which is then run in a RISC-V system. Almost similar to the above, in Freezer [54] a non-volatile memory system is evaluated using a RISC-V-based microcontroller. In a somewhat tangential direction, [60] proposes a fault-tolerant RISC-V embedded system for use in space (intermittent power can be considered as 'fault').

Table 2.2: **State-of-the-art intermittent systems with cache support.**

system	compiler enhanced	legacy compatible	RISC-V support	supports JIT	No extra HW
Clank [30]	✗	✓	✗	✓	✗
PROWL [31]	✗	✓	✗	✗	✓
MEMIC [63]	✓	✗	✗	✓	✗
ReplayCache [75]	✓	✗	✗	✓	✓
COACH [32]	✓	✗	✗	✓	✗
NVMr [14]	✗	✓	✗	✓	✗
NACHO	✓	✓	✓	✓	✓

system	supports cache	cache arch agnostic	reduces NVM accesses	incorruptible
Clank [30]	✗	✗	✗	✓
PROWL [31]	✓	✗	✗	✓
MEMIC [63]	✓	✗	✓	✗
ReplayCache [75]	✓	✓	✗	✗
COACH [32]	✓	✓	✗	∼ [†]
NVMr [14]	✓	✓	✓	✓
NACHO	✓	✓	✓	✓

[†] The work relies on existing checkpointing strategies, thus it can be as incorruptible as the choice of strategy.
Yes (✓) No (✗) Partially (∼)

2.3 Caches in Intermittent Systems

The primary challenge of having a cache is addressing volatility. The state of the cache has to be saved to a non-volatile memory location, and various works have taken various different approaches in doing so. Table 2.2 shows the different features of all the related work compared to our system.

Changing the cache

Some works present a modified cache that supports or enables intermittency in some way. Modification can be in the cache architecture, placement/replacement policies, in the way the cache responds to intermittency, and in how closely knit the cache is to the system.

PROWL [31] uses a custom replacement policy and the associativity of the cache system to decrease the number of checkpoints in the system. The basis of their proposal is to mask the need for more checkpoints by delaying the eviction of a dirty cache block. As long as the dirty cache block has not been evicted, the main memory will not record the value, and there will be no consistency problems in the event of power failure. A detailed analysis is provided below in Section 2.3.1. What we do in NACHO is similar, but has key differences.

ReplayCache [75] discusses that "stores" are not synchronous if a system uses a data cache. They can happen independently of the code flow. For an intermittent system, this causes a problem because there is no way to know whether a given "store" has been evicted from the cache to memory. So, ReplayCache "replays" all stores in a fixed region of code after an event of power failure, thus restoring the cache state to what it was before. It is a case of a redo log [52] that incorporates the cache. A detailed analysis is provided below in Section 2.3.2.

Our solution NACHO incorporates cache very closely with the system, but retains the flexibility to configure without affecting intermittent capability.

Using hybrid memory

Other works use a hybrid memory architecture designed specifically to handle intermittency.

COACH [32] and MRAM-based [61] use parallel memories to effectively create live backup. Although elegant, these proposals lead to higher energy consumption and higher complexity of the design.

In [71, 69] the authors propose another approach in which the cache has a set of volatile ways and a set of non-volatile ways. In [57] a new SVN-RAM memory architecture is presented. Although these works claim to improve energy efficiency, they do not address any consistency approach.

Clank [30], which does not use a cache, but is interesting to explore, as it uses dedicated memory tracking hardware that detects data inconsistencies during execution time. In NACHO we do something similar, but we do not use such dedicated hardware. Instead, as we explain in Section 4.1, we intelligently use the cache to perform consistency detection.

In a very recent work [14] the authors propose a memory renaming approach in which dirty blocks that are evicted from the cache are renamed to avoid causing idempotency violations. This approach uses a section of the NVM as a map table, a corresponding map table cache, and adds filters to the data cache to facilitate renaming. Although they reduce NVM accesses, the sacrifice comes in terms of using extra, dedicated hardware components and consuming more space in the NVM. Our solution achieves this without the above-mentioned pitfalls.

2.3.1 Analysis - PROWL

PROWL uses a combination of a replacement policy and the associativity of the cache system to decrease the number of checkpoints in the system. Then it compares the proposal with three (four) different approaches and presents the results. The three (four) approaches are as follows.

- QuickRecall [33] - known as Unified NVM - has no cache but uses JIT
- Ratchet [68] - Offline approach of putting checkpoints
- Clank [30] - Online approach by detection of idempotent sections
- (Similar to what they propose but using a LRU replacement policy)

The basis of their proposal is to mask the need for more checkpoints by delaying the eviction of a dirty cache block. As long as the dirty cache block has not been evicted, the main memory will not record the value, and there will be no consistency problems in the event of a power failure. The way they achieve this is by proposing a cache replacement policy and a cache controller that integrates the checkpointing with the policy. Their primary idea is to continue searching on the 'ways' of the cache until a block is found to keep the dirty block. This idea of re-hashing is called cuckoo hashing. If there are no blocks available to store the dirty block, it is evicted to the main memory, and a checkpoint signal is raised. They also specify which family of hashing function they use to ensure maximum efficiency.

There is no need to restore the cache data, as there is always a checkpoint to ensure whenever a block is written to the main memory. This idea is similar to a delayed write-back, but applied to a cache system. Their result consists of the following takeaways.

- Increases energy efficiency by reducing the number of accesses to NVM
- Increases energy efficiency by reducing the number of checkpoints
- Improves the average response times
- Improves the cache load factor (utilization of cache)

2.3.2 Analysis - ReplayCache

ReplayCache enables intermittent systems to efficiently use data caches. To better understand the proposal in the research, a discussion of the problem of having caches needs to be highlighted. Having data caches means that the "stores" are not synchronous anymore. They can happen independently of the code flow. For an intermittent system, this causes a problem because there is no way of knowing whether a given store has been evicted from the cache to the memory or not. If this information is not known, there is a risk of memory inconsistency in the event of power failure. An obvious solution to this is to use NVM caches. But that results in high latency and high-power consumption - both of which are detrimental to intermittent computing. The paper in consideration provides a solution in which a SRAM (volatile) cache-based system is designed to be intermittent capable using a software-only technique. The proposed solution works with existing systems; QuickRecall [33] and NVP [61] are used.

The primary idea is to “replay” all the stores in a fixed code region after an event of power failure. Therefore, the “stores” in that region are saved during the compile time in a statically allocated memory region. The code region is divided so that the number of registers used for the stores is not overwritten and thus remains consistent.

In power failure recovery (after the existing checkpoint restoration), the saved stores are re-executed. Once that is done, the program state returns to a consistent state, and flow can continue. This solution is claimed to be the first software-based approach to enable volatile caches in intermittent systems. The analysis and results presented in the paper compare the performance with various other cache-based systems.

2.4 Chipyard

To evaluate the implementation of new ideas, it is important to use a suitable and capable platform. One of such a platform is Chipyard [10] which is an open source framework for the agile development of chisel-based systems on a chip. The framework takes into input the processor description written in Rocket Chip (Chisel) along with the description of peripherals and computing components. Then it produces the Verilog output, which is emulated through the Verilator emulator. The Chipyard and Rocket Chip Generator are both open source and can thus be used in future research.

We used Chipyard to evaluate our proposed designs in the early phase of implementation. But as the complexity of the design grew, it became difficult to incorporate the same into Chipyard. Eventually, we moved to a software emulation-based system.

With the related works discussed, we can now proceed to present the motivation behind this work.

Chapter 3

Motivation

The number of volatile and non-volatile components in a system have a big impact on the total energy consumed. By exploring this property of memory volatility, we can gain insight into determining an energy-efficient system architecture.

3.1 Trade-offs in memory volatility

To understand the motivation to use a volatile data cache, we need to look at the potential energy consumption of a system, as shown in Figure 3.1. In the case of completely non-volatile systems such as a NVP, everything is written back to persistent storage, and the need to create checkpoints and restore them in case of power failure disappears. However, the energy consumed to perform memory accesses becomes high, diminishing the gains resulting from non-volatility. As we move towards the opposite spectrum of fully volatile architectures, the energy consumed per cycle decreases, but the size and number of the checkpoints and the cost of re-execution increases. For a fully volatile system, everything needs to be saved and restored, again skewing the associated costs. **The desired solution is a balance between the two, an architecture in which the volatility offers fast access speeds without sacrificing too much on the associated energy costs for checkpoints**.

We argue that this sweet spot lies in having a volatile data cache between the processor and a non-volatile main memory. The volatile SRAM-based data cache provides high access speeds, while being small enough to ensure low checkpointing overheads. The backing non-volatile main memory acts as the persistent entity to ensure data retention across power failures.

There are works that have explored alliterative solutions such as Hybrid-Cache [69] which uses both volatile and non-volatile cache together in a hybrid architecture and COACH [32] which replicates the main memory to store both the backed-up and the modified data, resulting in the use of two parallel memories. Unlike these systems, NACHO is designed to consume lower energy by not using any extra hardware other than a modified data cache. We emphasize the specificity of using no extra hardware by noting that the functions performed by hybrid memory, parallel memories, and the memory-tracker are intelligently incorporated into the data cache of NACHO. Other works such as NVMr [14] proposes use of a cache and a renaming scheme for the NVM accesses while

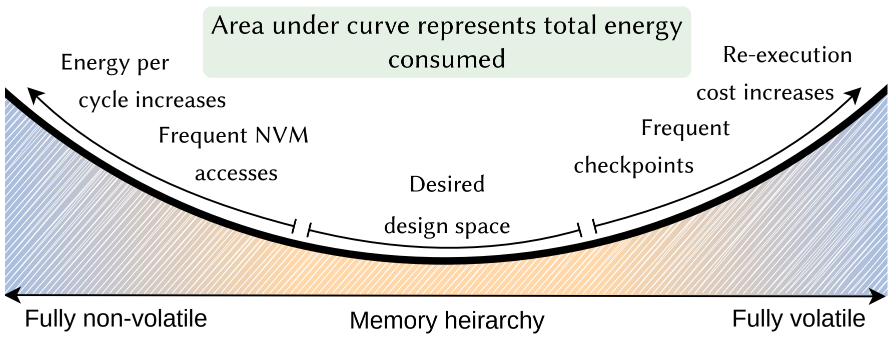
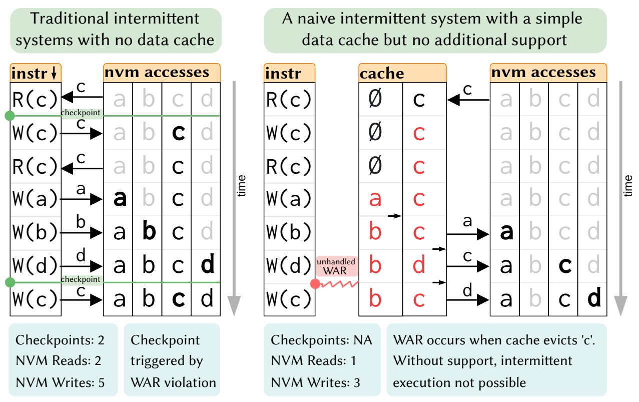


Figure 3.1: **An intermittent system needs to achieve a balance between the energy consumed in a backup operation and the energy consumed in performing regular memory accesses. And the key to this is controlling the volatility of the system.**



Traditional intermittent systems with no data cache
instr
nvm accesses
checkpoint
time
Checkpoints: 2
NVM Reads: 2
NVM Writes: 5
Checkpoint triggered by WAR violation
A naive intermittent system with a simple data cache but no additional support
instr
cache
nvm accesses
unhandled WAR
Checkpoints: NA
NVM Reads: 1
NVM Writes: 3
WAR occurs when cache evicts 'c'. Without support, intermittent execution not possible

Figure 3.2: **An example showing the memory accesses of a simple program performing R(x) (a read operation at the memory location "x") and W(x) (corresponding write operation) operations on four variables, as shown on the left-hand side. For the purpose of the illustration, we show only four memory blocks `a`, `b`, `c`, and `d` that correspond to some physical address in NVM memory. The cache shown is a simple direct associative write-back [29] cache of size two blocks.**

PROWL [31] makes changes to the cache policies to decrease checkpoints. But NACHO does not use use any new policies and memory access patterns which makes it compatible with any cache architecture.

3.2 Challenges

However integrating a cache requires careful system design. We illustrate the challenges using a simple example in Figure 3.2. Firstly, we show a traditional intermittent system (left side of the figure), such as [68], that places a checkpoint whenever there is a WAR violation, decided during compilation time. This results in a functioning system with two checkpoints and seven NVM accesses. Let us now consider the scenario (right side of the figure) in which a data cache is present between the processor and NVM. Here, we see the advantage of having a cache which exploits memory locality and reduces the number of NVM accesses to four. However, this system *cannot be executed intermittently* because the checkpoint placement logic depends on when a “write” to NVM is performed. Having a cache that buffers the memory accesses delays this write. This is a run-time phenomenon that cannot be predicted by the compiler, thus rendering the checkpoint placement incorrect. Therefore, this system is not protected against WAR violations. A simple addition of a cache to a working intermittent system that uses compiler-based checkpointing is not feasible.

Other works have proposed different ways to mitigate this. Some use a separate hardware entity to detect WARs [14] while some [71, 69] propose having a hybrid cache with volatile and non-volatile components. COACH [32] and [61] use parallel memories to effectively create live backup. Other works such as PROWL [31] and ReplayCache [75] tweak the cache to control when a WAR can occur. But each of them has a set of problems. HybridCache needs to constantly move around data between the two cache components, COACH has to maintain two parallel memories, which means extra energy consumption in doing so, and PROWL and ReplayCache are software-only solutions that do not exploit the hardware to have better energy efficiency as we do.

Chapter 4

Cache Integration

To overcome the challenges mentioned above, we propose a fundamentally different approach. Instead of using the cache as a separate and standalone entity, we interweave its functionality into the intermittent system. We design the cache to become the detection entity for WAR, in addition to increasing data retrieval performance by reducing the need to access the underlying data NVM.

In the remainder of the section, we explain the proposed design in detail. We also use an example in Figure 4.1 to aid in understanding. It consists of three steps that we use to build our proposal. Each step is assigned to one of the subsections(4.1, 4.2 4.3) below.

① **Evicting all dirty bits on a cache-induced checkpoint**: The example program starts with an empty cache. Following the instructions in the column `instr`, we see that `(c)` is first stored in the cache and then becomes dirty upon execution of the next instruction. Similarly, `(a)` is also stored as a dirty block in the empty set. After the fourth instruction, we observe that the cache is full. The next instruction requires access to `(b)` which means that one of the cache blocks must be evicted. As the collision set contains a dirty block, that is, `(a)`, the cache instructs the processor to initiate a checkpoint. As part of the checkpoint, all dirty blocks are evicted to the NVM, depicted by the green arrow. After eviction, the instruction is executed and `(b)` is stored in the first set. Note that `(c)` in set 2 is now marked as valid (not dirty) as it was evicted to NVM during the checkpoint. Consequently, on execution of the next instruction, a write to `(b)`, `(c)` is overwritten without the need for eviction. For the final instruction, the cache again has to signal a checkpoint, as dirty `(d)` has to be evicted to accommodate `(c)`. Looking at the number of checkpoints and NVM writes, we arrive at 2 checkpoints and 4 NVM writes. We see the gains from exploiting the cache's data locality by comparing this with the first figure in 3.2. Both NVM reads and writes are reduced by one.

② **Using possible WAR flag to only checkpoint during eviction of unsafe writes**. For this optimization, we instrument a small modification to the cache by adding a flag per cache set, which we refer to as the `possibleWAR` flag. The program is the same as before, but the cache enables a flag if a set stores a block read from the NVM; if it is the start of

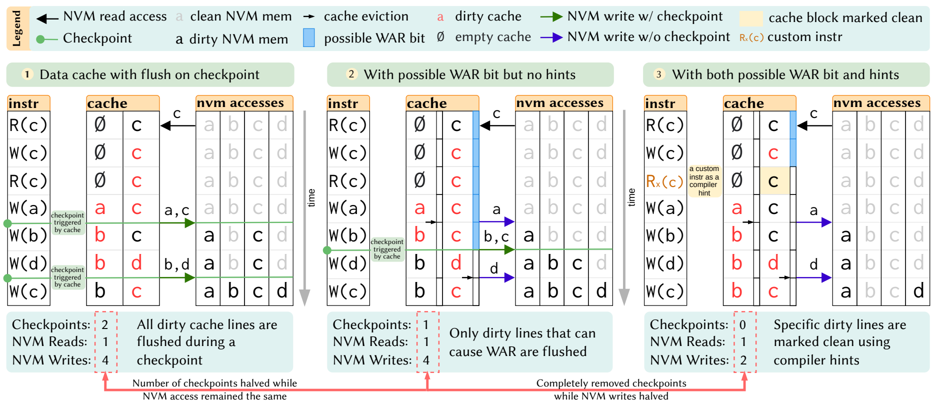

Legend
NVM read access
clean NVM mem
cache eviction
dirty cache
NVM write w/ checkpoint
cache block marked clean
Checkpoint
dirty NVM mem
possible WAR bit
empty cache
NVM write w/o checkpoint
custom instr
1 Data cache with flush on checkpoint
2 With possible WAR bit but no hints
3 With both possible WAR bit and hints
instr
cache
nvm accesses
time
checkpoint triggered by cache
a custom instr as a compiler hint
Checkpoints: 2 NVM Reads: 1 NVM Writes: 4
All dirty cache lines are flushed during a checkpoint
Checkpoints: 1 NVM Reads: 1 NVM Writes: 4
Only dirty lines that can cause WAR are flushed
Checkpoints: 0 NVM Reads: 1 NVM Writes: 2
Specific dirty lines are marked clean using compiler hints
Number of checkpoints halved while NVM access remained the same
Completely removed checkpoints while NVM writes halved

Figure 4.1: **This example shows the status of *cache* and *NVM accesses* for a simple abstract program. The column *instr* shows the memory instructions that are executed. A NVM access is a *read* from the NVM to the cache or an *eviction* of a dirty block from the cache to the NVM. There can be no access from the NVM without going through the cache. The cache used here is the same as that used in Figure 3.2. It is a direct associative cache with 2 sets. Each set can store one block, and the block size in the cache and in the NVM is the same. A cache block can be in one of the three states *valid*, *dirty*, and *empty*. A cache eviction is the writing of a dirty cache block back to the NVM. In some cases, this action can cause a checkpoint signal to be raised which is sent to the processor. In this case, all dirty cache blocks are evicted to the NVM. For all checkpoint-induced evictions, writes to the NVM are double-buffered and are not removed from the cache. In contrast, in a normal eviction, no checkpoints are created, and eviction implies that the data block is removed from the cache and written to the NVM. There are four arbitrary memory blocks *a*, *b*, *c*, *d* that are assigned to two sets of the cache in the following way *a*, *b* → *set 1 and c*, *d* → *set 2*.**

a read-dominated memory access. When the first instruction is executed, `(c)` is placed in set 2 of the cache as part of a NVM read. Thus, the `possibleWAR` flag is enabled (depicted by the blue box adjacent to the set). Now, the criteria for creating a checkpoint include the state of the flag `possibleWAR`. Only when the cache evicts a block that has the flag enabled can it raise a checkpoint signal. Therefore, when the fifth instruction is executed, `(a)` is evicted and replaced by `(b)` without creating a checkpoint. Tracking the history of the memory stored in the set allowed it to be safely evicted without the need to create a checkpoint. This normal eviction is depicted by a purple arrow in the example. The program continues, and a checkpoint is needed when `(c)` is evicted because the flag is enabled for the second set. The checkpoint also clears the flag `possibleWAR`. Comparing this with the approach in (1), we notice a reduction in the number of checkpoints.

③ **Using compiler analysis to mark certain dirty blocks as clean**. The compiler can provide valuable information about the structure of the program. For this step, we establish a bridge between the cache and the compiler, in which the compiler can signal the cache to perform intelligent evictions. Using custom assembly instructions, the cache can be instructed to mark certain dirty blocks as clean. Looking at the example, we focus on the memory block `(c)`. We can observe that after the fourth instruction that access `(C)`, the next access is a write in the last instruction. During this, it is not used by the program, and thus having it in the cache does not provide any gain. This can be detected by the compiler, and it can instruct the cache to mark the block as clean, even if it is dirty and has the flag `possibleWAR` set. Since it is now marked as clean, writing to `(d)` results in the eviction of a clean block that does not require any checkpoint. It also eliminates the need for a normal eviction. When the program needs `(c)` again, it is placed back in the cache. The important thing to note is that this optimization does not affect the correctness of the program in any possible way. The number of checkpoints is further reduced, and so is the number of NVM writes.

4.1 Using cache as a WAR detector

This insight is based on the fact that a data cache delays when a “write” to the underlying memory actually takes place. A WAR violation can only occur when a “write” is performed after a “read” at a memory location. Therefore, the cache effectively determines when WAR can occur by tracking the presence of such access patterns in a given cache line. We take advantage of this insight that *a WAR violation can only be possible when a cache block is written back to the NVM.* We term this event of a cache line being written back to NVM as *Cache Write Back (CWB)* [1]. Upon detection of a CWB, the cache generates a checkpoint signal and demands the processor to back up safely. The processor then copies the registers to NVM. Since the cache is also volatile, *all modified memory blocks* are copied to NVM[2]. By doing this backup, *we ensure that the system remains consistent.* An added advantage comes in the form of clearing the cache of *all* dirty lines during the checkpoint since it decreases the possibility of a future WAR and thus reduces the number of created checkpoints.

Looking at the number of checkpoints and NVM writes in Figure 4.1 ①, we arrive at 2 checkpoints and 4 NVM writes. We see the gains from exploiting the cache’s data locality by comparing this with the first figure in 3.2. Both NVM reads and writes are reduced by one. But there is still room for improvement.

4.2 Optimizing WAR detection

As explained above, the core idea is that a CWB can lead to a WAR violation. However, some of these CWBs may not lead to a violation. This can be understood more formally as the memory being read-dominated or write-dominated

[1]Similar to Intel x86 instruction CLWB

[2]Since the cache and the registers are still volatile, this copy operation has to be a safe backup, for example by using double buffering.

as introduced in [30, Section 3.1.1]. They state that, for a sequence of memory instructions, if the first access to memory addresses is a write, then this location is write-dominated. Conversely, if the first access to a memory address (in a given sequence of instructions) is read, then this location is read-dominated. The idempotency violation can be defined as *a write to a read-dominated memory location*. Any other form of access is safe and will not cause a violation. Since any given memory sequence can be read-dominated or write-dominated, this condition bounds all possible idempotency violations. With this understanding, they used dedicated hardware to track whether memory accesses are read- or write-dominated. On the contrary, we use *the cache to perform the same tracking*, eliminating the need for an additional hardware component.

To understand how the cache performs this tracking, we take an example of a simple cache of total size 256 bytes. We can consider it to be a 2−way set-associative cache [29] with a block size of 4 B. This results in a total of 32 sets, each set having two lines with a data block of 4 B in each line. Program memory is then mapped to these 32 sets using a simple hash of the memory address. Since the mapping of addresses to the cache is static, we can deduce the memory access sequence during the execution of the program, which can then predict whether a CWB can cause a WAR or not by analyzing this memory sequence.

To facilitate a clearer understanding, we redefine read-dominated and write-dominated sequences to track a cache line instead of a memory address. A read-dominated cache line is when the accesses in that line contain a read access. Conversely, a write-dominated cache line is where there are only write accesses (no read accesses) in this way.

A CWB does not result in a violation if it comes from a write-dominated cache line. We term this write-back as a “safe-write”. A CWB can only result in a violation if the associated cache way is read-dominated, which we term as an “unsafe-write”. We track these memory sequences to all the cache lines during the course of program execution and create a checkpoint only if an unsafe-write is encountered.

This optimization is shown in Figure 4.1 ②. Compared to the approach in ①, we notice a reduction in the number of checkpoints, but the NVM access remains constant. An important thing to note is that the cache stores data based on a hash of the memory address *distinction between safe-write and unsafe-write is also based on the hashed address*. This implies that the WAR detection is not exact and can contain false positives. This is a trade-off in using the cache (and not a dedicated hardware module) as a memory tracker. However, to overcome this trade-off, we take the help of compiler analysis.

4.3 Using compiler analysis

Even though it is not possible to use compiler-based WAR detection and checkpoint in the presence of a cache, there are still various insights that static analysis can provide, leading to a gain in efficiency. The scope of improvement mentioned above can be reduced by using such compiler optimizations. In this work, we use the compiler to provide information on the memories accessed by the program.

4.3.1 Detecting safe memory sequences

Some memory sequences do not lead to WAR even though they contain a write to a read-dominated memory. For example, in the situation where the write leading to WAR is succeeded by another write. In this case, the write does not have any impact on the consistency, as it is always overwritten. In case of a power loss, the re-execution of that section ensures that whatever inconsistent data is written to the memory is again written over by another write. In such a scenario, the creation of an additional checkpoint is not needed.

The compiler can detect such an occurrence and report it to the cache controller. The cache controller can then mark the cache block containing that particular memory address as “safe”. This prevents a checkpoint creation and reduces the number of non-volatile accesses as well.

4.3.2 Detecting dead memory accesses

In another scenario, a write leading to WAR can be a write that ends the access to that memory location. This is common in scenarios where the program writes something to a variable before exiting a function and the memory location is not accessed anymore. This scenario, although detected as a WAR violation by the cache, does not lead to inconsistency. Thus, the compiler can detect and mark such cases as safe as well. Similarly to the above case, the cache can instruct the cache controller to mark the cache block mapped to the memory location as safe.

We term this communication by the compiler to the cache controller as “compiler hints”. In Figure 4.1 ③ shows the working of the compiler hints and how it prevents the creation of the extra checkpoint.

4.4 System Architecture

An overview of the architecture is shown in Figure 1.1. The system consists of two primary units: the compilation and execution units. As part of the compilation unit, the modified compiler generates a binary with embedded cache hints. The execution unit comprises Microcontroller Unit (MCU), a data cache, a cache controller, and an energy harvesting unit. In essence, the cache controller and the cache can be considered as one hardware unit. All three components work closely to ensure that the system works intermittently with a guarantee of incorruptibility.

4.4.1 System Requirements

Along with supporting intermittent computing, our system provides the following features that make it better than state-of-the-art.

1. **Incorruptibility:** Our system ensures that the state of the program is never corrupted. As shown in Table 2.2, incorruptibility is not always guaranteed. Using cache as a WAR detector, our solution does not rely on energy prediction to create checkpoints. Using an energy detector itself consumes energy and estimating the threshold for the system is not

accurate [58, Figure 1]. This removes the aspect of unpredictability and assures a consistent state for any possible power trace.

2. **Cache Architecture Agnostic:** Even though our system incorporates a custom cache, we do so carefully to be agnostic to the cache architecture. Our solution can be used with any form of cache with any placement/replacement policies as we make no modification to the fundamental way in which caches work. Our additions are three additional bits that can be seamlessly integrated with any cache architecture. As we show in our evaluation, we experiment our system with various forms of cache and provide the parameters that we consider to be the most efficient.

3. **Supports JIT Checkpointing:** Along with a guarantee of incorruptibility, we also provide the option to create checkpoints using the JIT paradigm if desired. If such a paradigm is used, it removes the need to perform double-buffering when there is enough energy available. When the energy level is detected to be below a certain threshold, the system goes back to the safe method of creating checkpoints.

Assumptions

Also, we do not address interaction with system peripherals such as communication modules or sensors. This is a problem that needs to be solved separately and can be used in combination with our system.

4.4.2 Compilation Unit

The compilation unit works by modifying the compiler to instrument the source code with custom instructions. As explained in Section 5.2, we use custom instructions to inform the processor and the cache controller of the hints. In this section, we present a brief overview of how the compiler computes when and where to provide the hints. A detailed explanation of the compilation unit is beyond the scope of this thesis.

We use the LLVM compiler infrastructure to make modifications to the default compiler. Since LLVM is open source and has the support to make granular changes, it becomes a better choice than GCC. Using the intermediate representation and analyzing the program dependency graph (PDG), we determine the optimum location where the hints can be placed. Once the location of the hints are determined, the binary is instrumented with a custom instruction that can be understood by the processor and transferred to the cache controller.

The execution units fully supports the compilation unit in performing the above optimizations. The cache controller takes over and handles the WAR violations during runtime while taking into account the hints instrumented in the modified binary. This is performed with very low overhead.

4.4.3 Execution Unit

The execution unit executes the program and manages the checkpoints to ensure that intermittent computing is possible. The following major steps are performed when a checkpoint must be created.

1. The cache determines whether there is a need to create a checkpoint as described in Section 4.1. If so, then it signals the processor to perform a checkpoint.

2. As part of the checkpoint, the processor copies the registers to the NVM. Since the cache is volatile, all memory blocks that can cause WAR as determined by the logic described in Sections 4.2 and 4.3 are copied to the NVM.

3. The write-back to the memory during a checkpoint goes through a double-buffering mechanism. Once successfully backed up, the cache blocks that are copied back to NVM are marked as clean.

4. In the event a power failure occurs, the last checkpoint is restored. As part of this, the registers, the program counter, and the instruction pointer are restored. The program then continues to run. The cache, being a volatile entity, starts again from an empty state.

Chapter 5

Implementation

We now proceed to present the implementation details of the architecture described above.

5.1 Cache Controller

The cache in our system is managed by a cache controller through which we perform WAR detection and checkpoint placement as described in Section 4. An overview of the algorithm that governs this cache controller is presented in Algorithm 1 and is explained in the following sections.

5.1.1 Handling Cache Accesses:

The processor calls the procedure `MemoryAccess` (Line 1, Algorithm 1) for every memory instruction. When the processor requests access to a memory address, the required data can be in the cache (cache hit) or need to be retrieved from NVM (cache miss). In the event of a cache miss, the system can create a checkpoint if it has the possibility of leading to a WAR violation (see Section 5.1.2). As part of this checkpoint the cache controller also performs the eviction of the ditry cache lines in procedure `Checkpoint` (Line 25, Algorithm 1). Procedures `CacheLine`, which fetches a cache line using the memory address, and `ReplacementPolicy`, which selects a cache line to be evicted in case of a cache miss, are standard implementations based on the cache architecture.

5.1.2 Detecting WAR:

The cornerstone of our system is to detect when a WAR violation occurs; see Section 4.1. In NACHO WAR detection is performed with the help of four flags present for each cache line.

d (dirty): This standard flag indicates that the cache line has data that are out of sync with the corresponding data in the non-volatile memory.

rd (read-dominated): This custom flag is set by the cache controller to true when a memory access sequence in a given cache line is read-dominated as explained in Section 4.2.

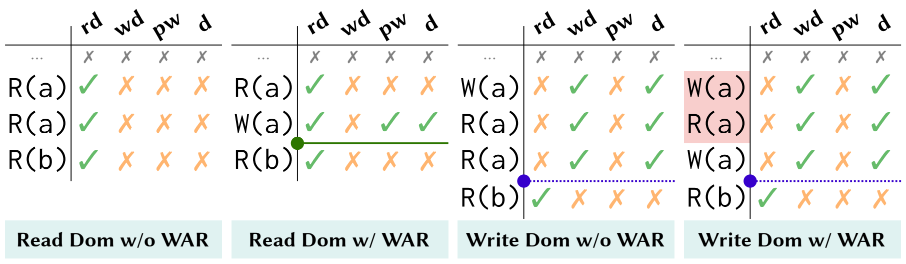


Figure 5.1: **The function of the various flags that are implemented in the cache as bits. The four flags are `rd`: read-dominated, `wd`: write-dominated, `pw`: possible-war and `d`:dirty. The function of the `rd` and `wd` flags corresponds to memory tracking as described in Section 4.2. These flags control when a checkpoint signal is triggered. The four different scenarios show all the possible memory accesses that can occur. `R(x)` represents a read-memory access to an address location `x`. The sequence marked in red causes WAR.**

wd (write-dominated): Corresponding custom flag for when a sequence is write-dominated.

pw (possible-war): This custom flag controls when a checkpoint is created as shown in Line 9, Algorithm 1. The value of this flag is set or reset in Procedure 15, Algorithm 1. In the event an unsafe write (see Section 4.2) is detected, the cache controller raises a checkpoint signal.

Figure 5.1 shows the different scenarios in which the flags can detect safe-writes and unsafe-writes (see Section 4.2).

5.2 Compiler

The detailed implementation of the custom compiler is beyond the scope of this thesis work. The execution unit is designed keeping in mind the functionality of the compiler. When receiving a compiler hint for a given address, the cache controller clears the appropriate cache block's `pw`, `rd` and `wd` bits.

This is supported as an add-on and can be seamlessly integrated with the system design. As described in Section 4.3, this additional compiler hint improves on the existing, already functional system.

Algorithm 1: Memory access handling

```
Algorithm MemoryAccess(address, type, value) :
    line, miss ← CacheLine(address)
    if miss is true then
        line = CacheMiss(address, type)
    UpdateLine(line, type)
    UpdateData(line, value)

Procedure CacheMiss(address, type):
    line = ReplacementPolicy(address)                    // Line to be evicted
    if line_pw is true then
        Checkpoint()
    else if line_dirty is true then
        Evict(line)                                       // Write the line to memory
        ResetLine(line)                                   // Clear all the bits in the line
    return line

Procedure UpdateLine(line, type) :
    if type is Read then
        if line_wd is false then
            line_rd ← true
    else if type is Write then
        if line_rd is false then
            line_wd ← true
        if line_rd is true then
            line_pw ← true
        line_dirty ← true

Procedure Checkpoint() :
    for line in Cache do                                 // For each line in the cache
        if line_dirty then
            SafeEvict(line)                              // Double buffered evict
        ResetLine(line)                                  // Clear all the bits in the line
    DoubleBufferWriteback()                              // Write back double buffered cache
    CheckpointInterrupt()                                // Trigger CPU checkpoint interrupt
```

Chapter 6

Evaluation

We now proceed with the evaluation of NACHO.

6.1 Evaluation Setup

We begin by presenting an overview of our evaluation setup. We present and justify our choice of target and platform.

6.1.1 Target Architecture

We implemented NACHO on top of the RISC-V architecture. The primary motivation behind this decision is the open-source nature of the RISC-V ISA [53], which allows modification of its ISA. We based our RISC-V processor on the SiFive E21 standard core [6], a basic 32−bit embedded processor that targets microcontroller applications. In addition to non-volatile main memory, we added our NACHO specific cache and cache controller, which are governed by the logic described in Section 5.

6.1.2 Evaluation Platform

The performance of NACHO was measured using an emulator designed for the evaluation of intermittent computing. Emulation enables us to track vital performance metrics, such as the number of clock cycles executed, the frequency of memory accesses, and the occurrence of checkpoints. These metrics are collected without adding to the execution time of the actual program, which is crucial for our evaluation. Furthermore, emulation allows us to introduce two verification steps, in which we verify the correct implementation of NACHO and related works (see Section 6.3. Firstly, for every memory access generated by the processor, we duplicate the same to a shadow memory. In this way, a memory access request handled by NACHO would return the same value as contained in the shadow memory. For the second safety measure, the emulator performs WAR detection to verify the absence of any WAR violation, as done in [50, Section 5.2] and [38, Section 5.1.1] by using read and write specific address lists and observing access patterns. These steps guarantee that, while using NACHO, the evaluated benchmarks are incorruptible on power failures.

We selected ICEmu [21] as it supports the 32-bit RISC-V CPU architecture and allows us to extend its behavior to implement NACHO and the related work. ICEmu internally uses the QEMU-based [5] Unicorn CPU emulator [7].

6.2 Memory Cost Model

It should be noted that the exact performance of NACHO depends on the implementation of the memory technology and components used. For the purpose of evaluation, we have assumed a conservative set of metrics in which an access to the NVM consumes twice the number of cycles than an access to the cache. Usually, the cost is much higher, around 4-5× [31, Table 1].

6.3 Evaluated Systems

To assess NACHO we need to compare it with state-of-the-art systems for intermittent computing. For this, we have selected PROWL [31]: a state-of-the-art intermittent system supporting cache, and Clank [30], a dedicated memory tracker-based intermittent system that performs dynamic WAR detection similar to NACHO but without cache support. Additionally, we have considered two versions of NACHO, enabling us to see how cache operations introduced in Section 4 affect its performance.

① **Oracle NACHO**: This is a theoretical version of our system that performs exact memory tracking on top of a cache. As we describe in Section 4.1, detection of WARs in our system is performed based on the granularity of a cache line—we check accesses to read/write-dominated cache lines—which can lead to false WARs being detected. Oracle NACHO makes this detection with exact addresses, similar to how Clank uses memory buffers to detect WARs [30, Section 3.1]. In other words, Oracle NACHO is a Clank with added cache. Although this makes it a perfect WAR detector, this approach will incur a significant hardware cost and complexity to the system that is not practical to implement, as it consists of a cache as well as a dedicated per-memory address tracker. In contrast, NACHO achieves WAR detection using just three bits per cache line (see Section 5.1. Thus, we treat Oracle NACHO a theoretical lower bound to NACHO and use it only for comparison purposes.

② **Naive NACHO**: To see the improvements of each set of cache optimizations (as listed in Section 4) that we implement, we also consider a naive version of NACHO that has just the WAR detection logic without any of the optimizations implemented. Similar to Oracle NACHO, we use Naive NACHO just for comparison purposes.

③ **PROWL**: Then we have also re-implemented PROWL [31][1]. We choose to include PROWL so as to provide a reference frame as it performs cache modifications for intermittent systems. PROWL incorporates Cuckoo hashing into its cache replacement policy and uses the skew-associativity of the cache

[1]Our implementation of PROWL might not be the exact same as the authors as at the time of writing PROWL implementation was not released as open source. Most specifically, the authors use a family of hashing functions, but do not provide the exact functions they use for the evaluation. However, the overall idea remains the same and we consider that the numbers presented are accurate enough for comparison purposes.

system to decrease the number of checkpoints in the system. The basis of PROWL is to mask the need for more checkpoints by delaying the eviction of a dirty cache block. To achieve this, PROWL performs a lot of in-cache data movement, which increases the cost of execution on every cache miss.

In addition, PROWL does not perform any WAR detection, instead it *delays* the occurrence of WAR. This makes it a system that can work in tandem with NACHO which can detect WARs, and PROWL can implement its custom replacement policy on top of NACHO's WAR detection logic.

② **Clank**: Lastly, we also implement Clank [30], which does not use a cache, but uses a dedicated memory tracking hardware module which detects data inconsistencies during execution time. Since NACHO performs similar detection of WAR using only the data cache, we include Clank as a baseline to compare performance metrics.

6.4 Benchmarks Used

The benchmarks used in the evaluation are CoreMark [18], CRC, SHA, and Dijkstra from MiBench suite [28], picojpeg [24] and Tiny AES [37]. Coremark is an industry-grade benchmark for measuring embedded system's CPU performance. TinyAES and picojpeg represent real-life application examples. All selected benchmarks have been widely used in previous works on intermittent computing, such as [38, 39, 68].

6.5 Results

With the setup explained, we can now present the results of our evaluation.

6.5.1 Analysis

Figures 6.1, 6.2, 6.3 and 6.4 show various evaluations of our system using different parameters.

Execution Time

The execution time for a program gives insight into the actual computations performed. In Figure 6.1 we see that NACHO consumes **24.3% and 28% less execution cycles than Clank** for a 2-way 256 B and 512 B cache configuration respectively. It also **consumes less cycles compared to Prowl by 17% and 9%**. We would like to emphasize that this performance improvement over Clank—which has a dedicated hardware memory tracker—and PROWL—which uses complicated cache associativity and replacement policies—is achieved by adding just three extra bits (see Section 5.1.2) to a standard data cache (see Section 3.1). However, the execution time does not provide the full picture. We need to bisect and understand how individual aspects of the systems contribute to the variation of performance.

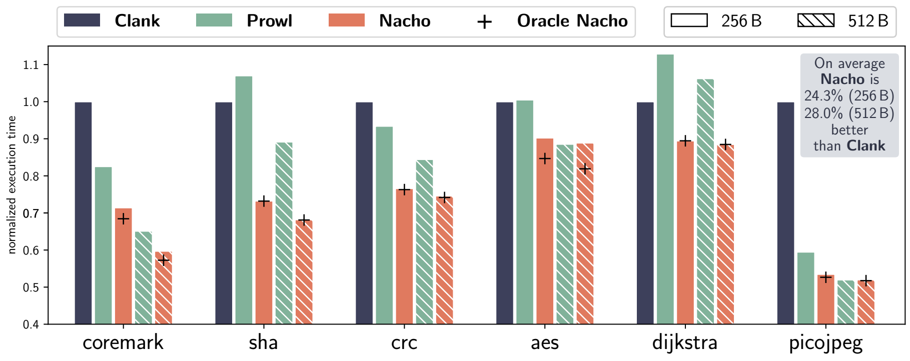


Figure 6.1: **Execution time for all benchmarks for Clank [30], Prowl [31], NACHO and Oracle NACHO (see Section 6.5). All results are normalized to Clank as a reference system. The Oracle NACHO is shown as the lower bound. The cache configuration used is 2-way set-associative for two cache sizes 256 B and 512 B. Note that Clank is a non-cache system and is thus not affected by cache configuration. NACHO uses up to 28% less execution cycles compared to Clank and up to 17% less than PROWL**

Checkpoints

We now consider the number of checkpoints created by the systems during the execution of the benchmarks. Checkpoints are usually correlated with a longer execution time and higher memory accesses. Therefore, a reduction in the number of checkpoints would indicate lower execution times. Results are presented in Figure 6.2 that contradict this observation. We observe that NACHO creates fewer checkpoints than Clank but that number is higher compared to PROWL, which creates on average three times less checkpoints than NACHO.

Let us discuss PROWL's results further. PROWL uses Cuckoo-hashing [31, Section 3.C] combined with a skew associative cache that stores dirty blocks in the cache longer than a traditional cache. Skew associativity provides multiple hashing functions, which are then used when handling evictions to find an empty position for an evicted block. With this modification to the replacement policy, PROWL delays the need to create a checkpoint until the cuckoo hashing fails to relocate a dirty block within a certain number of iterations. This results in a significant reduction in the number of checkpoints, as we can see in Figure 6.2. However, such a reduction in the number of checkpoints does not translate into a reduction in the execution time shown in Figure 6.1. This disparity in numbers leads us to explore the memory accesses for the systems.

Memory Accesses

The number of volatile memory accesses by each system is presented in Figure 6.3. We observe that the accesses by PROWL are significantly higher than the rest. On average, NACHO has 38.3% fewer volatile memory accesses compared to PROWL. This higher number of volatile accesses is related to the cache replacement policy use by PROWL. On every cache miss, the cache performs cuckoo hashing,

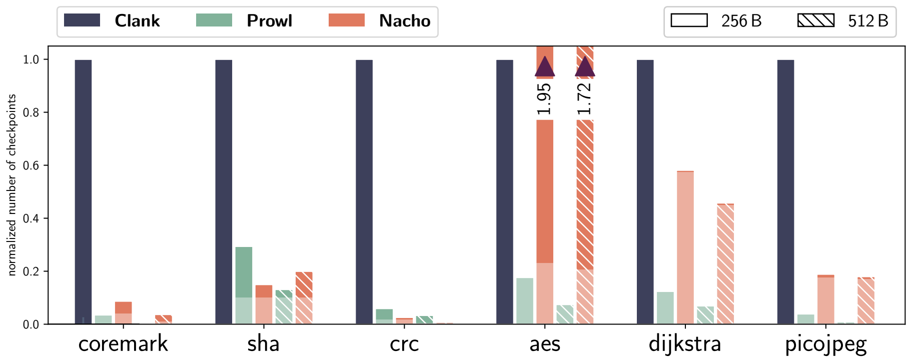


Figure 6.2: **Number of checkpoints created during all benchmarks for Clank [30], Prowl [31] and NACHO. All results are normalized to Clank as the reference system. Bars with outlier results are marked with the maximum value they reach. The same cache configuration as presented in Figure 6.1 is used. NACHO creates fewer checkpoints than Clank, but that number is higher compared to PROWL**

leading to a much higher number of cache accesses (see the corresponding bar stack in Figure 6.3) than our system. This rise in volatile-memory accesses compensates for the gains obtained via the reduction in checkpoints for PROWL. Effectively, NACHO creates more checkpoints for a lower number of memory accesses and simultaneously maintains low execution cost. Frequent checkpoints also mean that the regions between checkpoints are small. This leads to the size of checkpoints being smaller compared to PROWL, thus reducing the re-execution cost as well.

Discussion

Having a lower number of checkpoints makes PROWL closer to a volatile-heavy system, which then has to be supplemented with periodically induced checkpoints. Although NACHO sacrifices the number of checkpoints, it keeps the overall execution cost lower. Furthermore, PROWL achieves fewer checkpoints by modifying the cache architecture. This makes it inflexible and complicated, as seen with the extra volatile-memory accesses that are carried out during the cuckoo hashing on cache misses. NACHO on the other hand, can be used with any form of cache architecture with minimal overhead. NACHO can also be implemented on top of PROWL to combine the improvements of both systems and future cache mechanisms.

The stepwise improvement from Clank to the initial naive NACHO and then to our system shows how each implementation influences the performance. A further comparison with Oracle NACHO also shows that our system is on average within 2% of its performance. Recall that Oracle NACHO is an ideal lower bound with a perfect memory tracker on top of a data cache, and our system remains so close to this bound using very low hardware overhead and software complexity.

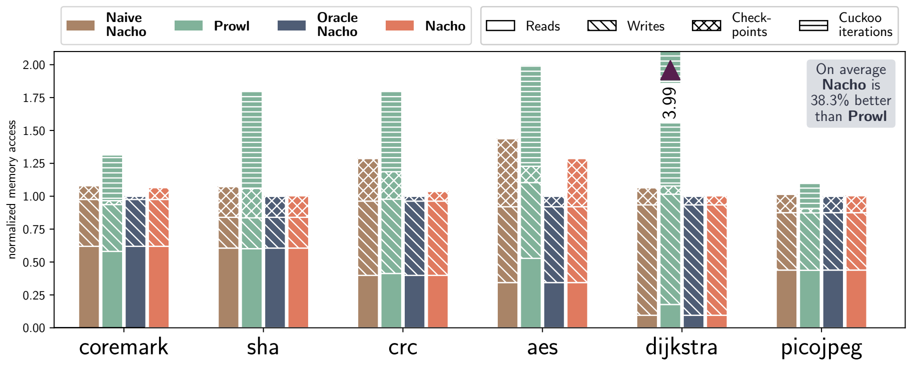


Figure 6.3: **The number of volatile memory accesses during all benchmarks for Prowl [31] and the three variations of NACHO (see Section 6.5). All results are normalized to Oracle NACHO as a reference system. Bars with outlier results are marked with the maximum value they reach. The different forms of volatile accesses are shown in the stacked bars for each system. The "Cuckoo iterations" is present only for Prowl due to their nature of cache replacement policy.**

6.6 Cache Configurations

We now explore the effects of cache configurations on NACHO. In the evaluations, we used 2-way set associative caches. The associativity of a cache implies the number of cache blocks that are assigned to each hashed entry. A higher associativity implies that the cache can store a larger number of blocks for a given mapping before it needs to evict to make space. In Figure 6.4 we present the execution cost of our system in different cache configurations with varying sizes and associativity.

6.6.1 Cache Sizes

We observe that an increase in cache sizes improves performance. This is expected, as with a larger cache, more dirty blocks can stay in the cache, effectively increasing the region between WARs and therefore checkpoints. Furthermore, a larger cache creates smaller mappings between cache lines and program memory, which gives higher accuracy to per-line WAR detection used in our system, thus bringing our system closer to the performance of Oracle NACHO.

6.6.2 Cache Associativity

Similar to cache sizes, the increasing number of cache associativity further improves the performance of NACHO. It is interesting that this varies slightly with the benchmark used, with the effect being more pronounced with picojpeg and AES. Furthermore, in these benchmarks, the improvement of performance in doubling the number of ways is more than that of doubling the size. This can be attributed to the fact that increasing associativity decreases the probability of a cache collision on cache miss, thus reducing the probability of a checkpoint.

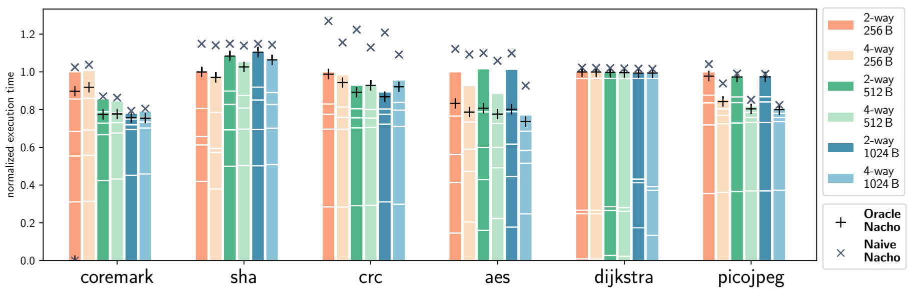


Figure 6.4: **A design space exploration of cache configurations with NACHO. All results are normalized to 2-way 256 B cache. Oracle NACHO and Naive NACHO metrics are marked as references to provide a lower and upper bound, respectively.**

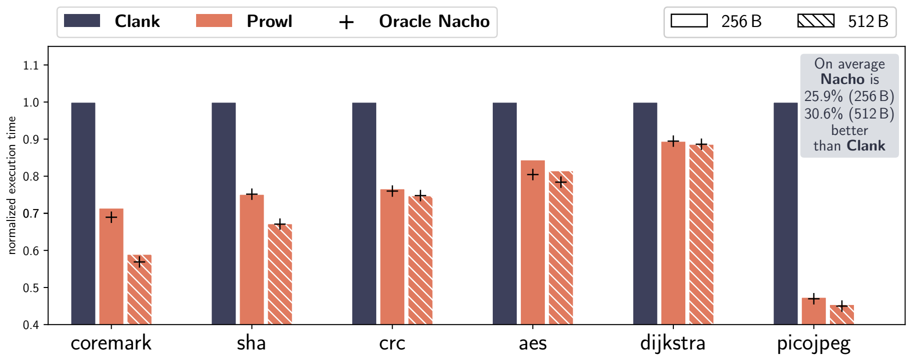


Figure 6.5: **Execution time for all the benchmarks for Clank and NACHO with a 4-way set associative cache. Similar to Figure 6.1 but with associativity doubled. The performance of NACHO increases with an increase in the associativity of the cache.**

An increase in cache sizes and associativity leads to an increase in the checkpoint size, as the volatile data cache needs to be saved to the non-volatile memory. However, the results show the opposite, and this can be attributed to the higher accuracy of the detection of WAR and the fewer cache collisions.

On the basis of the above discussion, we can state that a 4-way set-associative cache is more effective compared to a 2-way cache. Our prior evaluations used 2-way due to limitation on the number of hashing functions available to us for PROWL's implementation. An evaluation of 4-way cache is shown in Figure 6.5 where the performance improvement is better than that shown in Figure 6.1.

Chapter 7

Conclusions and Future Works

7.1 Conclusion

We presented NACHO an intermittent system with full cache support without additional hardware components. NACHO uses the cache as a WAR detection entity and builds on this idea to provide a complete, energy-efficient and yet incorruptible system. By reducing the number of checkpoints and memory accesses, it achieves better performance than the state-of-the-art, while offering support for any form of cache architecture. We evaluated the system presented providing comparisons to similar works and noted our observations. We ended the work by stating the cache configuration that works best for our system.

7.2 Future Work

The use of caches in intermittent systems is still being developed and needs further dedicated research. In this work, we have explored how to use a cache efficiently, but there remains scope for improvement.

Currently NACHO decides on the occurrence of WARs only by tracking the memory in the cache. Compiler support mentioned in Section 4.3 still needs to be implemented. This will provide a much higher efficiency gain in both the reduction of the number of checkpoints and the number of non-volatile accesses.

Furthermore, our implementation is done in a highly configurable emulator. This enabled us to implement all the different variations of NACHO and provide a proper evaluation. The next step would be to implement the same in the actual hardware. We initially aimed to implement the same in a SoC generator and synthesize it in an FPGA but the workload proved to be higher than expected, and thus that remains for future work. A brief introduction to the explored SoC generator is given in Section 2.

Another possible direction of work is to implement NACHO on top of PROWL. Our idea can work in collaboration with that of PROWL and thus can lead to a better and efficient system.

Lastly, the next logical step after integrating a data cache is to add support for a similar instruction cache. This would inch the system towards a complete intermittent framework.

Bibliography

[1] ESP8266 Wi-Fi MCU. https://www.espressif.com/en/products/socs/esp8266.

[2] MSP430G2452 data sheet, product information and support. https://www.ti.com/product/MSP430G2452.

[3] STM32F103C8 - Mainstream Performance line, Arm Cortex-M3 MCU with 64 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN - STMicroelectronics. https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html.

[4] STM8S103F3 - Mainstream Access line 8-bit MCU with 8 Kbytes Flash, 16 MHz CPU, integrated EEPROM - STMicroelectronics. https://www.st.com/en/microcontrollers-microprocessors/stm8s103f3.html.

[5] QEMU / QEMU · GitLab. *GitLab*, 2021.

[6] SiFive E21 Core Complex Manual. page 155, 2021.

[7] Unicorn Engine. Unicorn Engine, August 2022. original-date: 2015-08-20T16:35:45Z.

[8] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. ACM Press, 2019.

[9] Naved Alam, Prashant Vats, and Neha Kashyap. Internet of Things: A literature review. In *2017 Recent Developments in Control, Automation & Power Engineering (RDCAPE)*, pages 192–197, October 2017.

[10] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, July 2020. Conference Name: IEEE Micro.

[11] Domenico Balsamo, Anup Das, Alex S. Weddell, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Graceful Performance Modulation for Power-Neutral Transient Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):738–749, May 2016. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[12] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[13] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters*, 7(1):15–18, March 2015. Conference Name: IEEE Embedded Systems Letters.

[14] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. NvMR: non-volatile memory renaming for intermittent computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1–13, New York New York, June 2022. ACM.

[15] Naveed Anwar Bhatti and Luca Mottola. HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 209–220, April 2017.

[16] Jongouk Choi, Qingrui Liu, and Changhee Jung. CoSpec: Compiler Directed Speculative Intermittent Computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 399–412, New York, NY, USA, October 2019. Association for Computing Machinery.

[17] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 514–530, New York, NY, USA, October 2016. Association for Computing Machinery.

[18] Embedded Microprocessor Benchmark Consortium. Coremark. August 2022. original-date: 2018-05-23T00:53:13Z.

[19] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable Timekeeping for Intermittent Computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 53–67, New York, NY, USA, March 2020. Association for Computing Machinery.

[20] Brad Denby, Emily Ruppel, Vaibhav Singh, Shize Che, Chad Taylor, Fayyaz Zaidi, Swarun Kumar, Zac Manchester, and Brandon Lucia. Tartan Artibeus: A Batteryless, Computational Satellite Research Platform. *Small Satellite Conference*, August 2022.

[21] Virgil Dupras. ICemu - Emulate Integrated Circuits. February 2022. original-date: 2017-09-06T12:18:44Z.

[22] Laura Marie Feeney, Christian Rohner, Per Gunningberg, Anders Lindgren, and Lars Andersson. How do the dynamics of battery discharge affect sensor lifetime? In *2014 11th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 49–56, April 2014.

[23] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. The What's Next Intermittent Computing Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 211–223, February 2019. ISSN: 2378-203X.

[24] Rich Geldreich. Picojpeg. July 2022. original-date: 2015-05-21T03:53:59Z.

[25] Mohammadmahdi Ghiji, Vasily Novozhilov, Khalid Moinuddin, Paul Joseph, Ian Burch, Brigitta Suendermann, and Grant Gamble. A Review of Lithium-Ion Battery Fire Suppression. *Energies*, 13(19):5117, January 2020. Number: 19 Publisher: Multidisciplinary Digital Publishing Institute.

[26] Rabeeh Golmohammadzadeh, Fariborz Faraji, Brian Jong, Cristina Pozo-Gonzalo, and Parama Chakraborty Banerjee. Current challenges and future opportunities toward recycling of spent lithium-ion batteries. *Renewable and Sustainable Energy Reviews*, 159:112202, May 2022.

[27] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, 48(2):32:1–32:36, November 2015.

[28] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, December 2001.

[29] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Elsevier, October 2011.

[30] Matthew Hicks. Clank: Architectural Support for Intermittent Computation. *ACM SIGARCH Computer Architecture News*, 45(2):228–240, June 2017.

[31] Ali Hoseinghorban, Mohammad Abbasinia, and Alireza Ejlali. PROWL: A Cache Replacement Policy for Consistency Aware Renewable Powered Devices. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2020. Conference Name: IEEE Transactions on Emerging Topics in Computing.

[32] Ali Hoseinghorban, Amir Mahdi Hosseini Monazzah, Mostafa Bazzaz, Bardia Safaei, and Alireza Ejlali. COACH: Consistency Aware Check-Pointing for Nonvolatile Processor in Energy Harvesting Systems. *IEEE Transactions on Emerging Topics in Computing*, 9(4):2076–2088, October 2021. Conference Name: IEEE Transactions on Emerging Topics in Computing.

[33] Hrishikesh Jayakumar, Arnab Raha, and Vijay 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, pages 330–335, January 2014. ISSN: 2380-6923.

[34] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Transactions on Embedded Computing Systems*, 16(3):65:1–65:23, April 2017.

[35] Andreas Jossen. Fundamentals of battery dynamics. *Journal of Power Sources*, 154(2):530–538, March 2006.

[36] Do Hee Keum, Su-Kyoung Kim, Jahyun Koo, Geon-Hui Lee, Cheonhoo Jeon, Jee Won Mok, Beom Ho Mun, Keon Jae Lee, Ehsan Kamrani, Choun-Ki Joo, Sangbaie Shin, Jae-Yoon Sim, David Myung, Seok Hyun Yun, Zhenan Bao, and Sei Kwang Hahn. Wireless smart contact lens for diabetic diagnosis and therapy. *Science Advances*, 6(17):eaba3252, April 2020. Publisher: American Association for the Advancement of Science.

[37] kokke. Tiny AES (C). August 2022. original-date: 2012-05-24T15:27:24Z.

[38] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. WARio: efficient code generation for intermittent computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 777–791, New York, NY, USA, June 2022. Association for Computing Machinery.

[39] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive Intermittent Computing Meets Legacy Software. ACM, 2020.

[40] Fuyang Li, Keni Qiu, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yong Guan, and Chun Jason Xue. Checkpointing-Aware Loop Tiling for Energy Harvesting Powered Nonvolatile Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(1):15–28, January 2019. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[41] Hehe Li, Yongpan Liu, Qinghang Zhao, Yizi Gu, Xiao Sheng, Guangyu Sun, Chao Zhang, Meng-Fan Chang, Rong Luo, and Huazhong Yang. An energy efficient backup scheme with low inrush current for nonvolatile SRAM in energy harvesting sensor nodes. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 7–12, March 2015. ISSN: 1558-1101.

[42] Jinyang Li, Yongpan Liu, Hehe Li, Zhe Yuan, Chenchen Fu, Jinshan Yue, Xiaoyu Feng, Chun Jason Xue, Jingtong Hu, and Huazhong Yang. PATH: Performance-Aware Task Scheduling for Energy-Harvesting Nonvolatile Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(9):1671–1684, September 2018. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[43] Wen Sheng Lim, Chia-Heng Tu, Chun-Feng Wu, and Yuan-Hao Chang. iCheck: Progressive Checkpointing for Intermittent Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(11):2224–2236, November 2021. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[44] Yongpan Liu, Jinshan Yue, Hehe Li, Qinghang Zhao, Mengying Zhao, Chun Jason Xue, Guangyu Sun, Meng-Fan Chang, and Huazhong Yang. Data Backup Optimization for Nonvolatile SRAM in Energy Harvesting Sensor Nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(10):1660–1673, October 2017. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[45] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, June 2015.

[46] Kaisheng Ma, Xueqing Li, Srivatsa Rangachar Srinivasa, Yongpan Liu, John Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 678–683, January 2017. ISSN: 2153-697X.

[47] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution without Checkpoints. *arXiv:1909.06951 [cs]*, September 2019. arXiv: 1909.06951.

[48] Kiwan Maeng and Brandon Lucia. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. pages 129–144, 2018.

[49] Andrea Maioli and Luca Mottola. ALFRED: Virtual Memory for Intermittent Computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, pages 261–273, New York, NY, USA, November 2021. Association for Computing Machinery.

[50] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the Hidden Anomalies of Intermittent Computing. page 12, 2021.

[51] Azalia Mirhoseini, Bita Darvish Rouhani, Ebrahim Songhori, and Farinaz Koushanfar. Chime: Checkpointing Long Computations on Interm ittently Energized IoT Devices. *IEEE Transactions on Multi-Scale Computing Systems*, 2(4):277–290, October 2016. Conference Name: IEEE Transactions on Multi-Scale Computing Systems.

[52] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, February 2018. ISSN: 2378-203X.

[53] RISC-V Org. About RISC-V. In *RISC-V International*, 2022.

[54] Davide Pala, Ivan Miro-Panades, and Olivier Sentieys. Freezer: A Specialized NVM Backup Controller for Intermittently Powered Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(8):1559–1572, August 2021. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[55] Chen Pan, Mimi Xie, Yongpan Liu, Yanzhi Wang, Chun Jason Xue, Yuangang Wang, Yiran Chen, and Jingtong Hu. A lightweight progress maximization scheduler for non-volatile processor under unstable energy harvesting. *ACM SIGPLAN Notices*, 52(5):101–110, June 2017.

[56] Vijay Raghunathan, A. Kansal, J. Hsu, J. Friedman, and Mani Srivastava. Design considerations for solar energy harvesting wireless embedded systems. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 457–462, April 2005.

[57] Arnab Raha, Akhilesh Jaiswal, Syed Shakib Sarwar, Hrishikesh Jayakumar, Vijay Raghunathan, and Kaushik Roy. Designing Energy-Efficient Intermittently Powered Systems Using Spin-Hall-Effect-Based Nonvolatile SRAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):294–307, February 2018. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[58] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: system support for long-running computation on RFID-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 159–170, New York, NY, USA, March 2011. Association for Computing Machinery.

[59] H. Rayo Torres Rodriguez. A lightweight hardware architecture for intermittent computing, August 2019. Publisher: University of Twente.

[60] Douglas Almeida Santos, Lucas Matana Luza, Cesar Albenes Zeferino, Luigi Dilillo, and Douglas Rossi Melo. A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems. In *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, April 2020.

[61] SenniSophiane, TorresLionel, SassatelliGilles, and GamatieAbdoulaye. Non-Volatile Processor Based on MRAM for Ultra-Low-Power IoT Devices. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, December 2016. Publisher: ACM PUB27 New York, NY, USA.

[62] Neha Sharma and Deepak Panwar. Green IoT: Advancements and Sustainability with Environment by 2050. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 1127–1132, June 2020.

[63] Sivert T. Sliper, William Wang, Nikos Nikoleris, Alex S. Weddell, Anand Savanth, Pranay Prabhat, and Geoff V. Merrett. Pragmatic Memory-System Support for Intermittent Computing using Emerging Non-Volatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated*

Circuits and Systems, pages 1–1, 2022. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[64] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. Nonvolatile processors: Why is it trending? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 966–971, March 2017. ISSN: 1558-1101.

[65] Sujesha Sudevalayam and Purushottam Kulkarni. Energy Harvesting Sensor Nodes: Survey and Implications. *IEEE Communications Surveys & Tutorials*, 13(3):443–461, 2011. Conference Name: IEEE Communications Surveys & Tutorials.

[66] Dan Wang, Dong Chen, Bin Song, Nadra Guizani, Xiaoyan Yu, and Xiaojiang Du. From IoT to 5G I-IoT: The Next Generation IoT-Based Intelligent Algorithms and 5G Technologies. *IEEE Communications Magazine*, 56(10):114–120, October 2018. Conference Name: IEEE Communications Magazine.

[67] Harrison Williams, Michael Moukarzel, and Matthew Hicks. Failure Sentinels: Ubiquitous Just-in-time Intermittent Computation via Low-cost Hardware Support for Voltage Monitoring. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 665–678, June 2021. ISSN: 2575-713X.

[68] Joel Van Der Woude and Matthew Hicks. Intermittent Computation without Hardware Support or Programmer Intervention. pages 17–32, 2016.

[69] Mimi Xie, Chen Pan, Youtao Zhang, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. A Novel STT-RAM-Based Hybrid Cache for Intermittently Powered Processors in IoT Devices. *IEEE Micro*, 39(1):24–32, January 2019. Conference Name: IEEE Micro.

[70] Mimi Xie, Mengying Zhao, Chen Pan, Jingtong Hu, Yongpan Liu, and Chun Jason 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*, DAC '15, pages 1–6, New York, NY, USA, June 2015. Association for Computing Machinery.

[71] Mimi Xie, Mengying Zhao, Chen Pan, Hehe Li, Yongpan Liu, Youtao Zhang, Chun Jason Xue, and Jingtong 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)*, pages 1–10, October 2016.

[72] Yue Yang, Emenike G. Okonkwo, Guoyong Huang, Shengming Xu, Wei Sun, and Yinghe He. On the sustainability of lithium ion battery industry – A review and perspective. *Energy Storage Materials*, 36:186–212, April 2021.

[73] Bahram Yarahmadi and Erven Rohou. So Far So Good: Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying

Code. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '21, pages 29–34, New York, NY, USA, November 2021. Association for Computing Machinery.

[74] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. InK: Reactive Kernel for Tiny Batteryless Sensors. pages 41–53, Shenzhen, China, 2018. Association for Computing Machinery.

[75] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 170–182. Association for Computing Machinery, New York, NY, USA, October 2021.

[76] Mengying Zhao, Qingan Li, Mimi Xie, Yongpan Liu, Jingtong Hu, and Chun Jason Xue. Software assisted non-volatile register reduction for energy harvesting based cyber-physical system. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 567–572, March 2015. ISSN: 1558-1101.

[77] Yang Zhou, Mengying Zhao, Lei Ju, Chun Jason Xue, Xin Li, and Zhiping Jia. Energy-aware morphable cache management for self-powered non-volatile processors. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–7, August 2017. ISSN: 2325-1301.