COMPILER AND ARCHITECTURE CO-DESIGN FOR RELIABLE COMPUTING

by

Jianping Zeng

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Changhee Jung, Chair

Department of Computer Science

Dr. Xuehai Qian

Department of Computer Science

Dr. Mohammadkazem Taram

Department of Computer Science

Dr. Muhammad Shahbaz

Department of Computer Science

Dr. Yongle Zhang

Department of Computer Science

Approved by:

Dr. Kihong Park

ACKNOWLEDGMENTS

First of all, I am deeply grateful to my advisor Prof. Changhee Jung for his continuous support and patient guidence during the past six years. No matter when I got stuck in my research, he was always there to help me out. It would be impossible to bring this research to reality without his support. In addition, I appreciate him for introducing me to the field of computer architecture, which has been my dream research area since I was an undergraduate student, and for making me an independent researcher.

Also, I appreciate Prof. Xuehai Qian, Prof. Mohammadkazem Taram, Prof. Muhammad Shahbaz, and Prof. Yongle Zhang for serving on my defense committee and spending their time shaping this dissertation. Additionally, I am grateful to my collobrators, Prof. Dongyoon Lee, Dr. Changwoo Min, Prof. Trevor Carlson, for being my letter writers and helping me grow during the last five years. I thank Dr. Lide Duan and Dr. Yang Seok Ki for their support when I interned at Alibaba group and Samsung, respectively. Also, I thank many friends and lab-mates at Purdue and Virginia Tech, Tong Zhang, Xinwei Fu, Bowen Shen, Da Zhang, Xin Wang, Yihan Pang, Xiang Cheng, Jia Chen, Gagandeep Panwar, Jongouk Choi, Hongjune Kim, Shao-Yu Huang, Yuchen Zhou, Byounguk Min, Yida Zhang, Gan Fang, Mingqin Han, Eunice Lee, and Samuel Youssef. I greatly enjoyed hiking, climbing, fishing, and conversing with them.

Last but not least, I thank my family—especially my wife Nichao Jia—for unconditional love. Her support lights the way towards this dissertation. I feel guilty every time she was in need of my help while I was busy with my research. I hope I will have more time to spend with her.

The Introduction (Chapter 1), in part, uses material from the following publications.

Chapter 2, in full, is a reprint of the material as it appears in International Symposium on Microarchitecture (MICRO) 2021. Zeng, Jianping; Kim, Hongjune; Lee, Jaejin; Jung, Changhee. The dissertation author was the first author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in International Symposium on Microarchitecture (MICRO) 2021. Zeng, Jianping; Choi, Jongouk; Fu, Xinwei; P. Shreepathi, Ajay; Lee, Dongyoon; Min, Changwoo; Jung, Changhee. The dissertation author was the first author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in International Conference on Supercomputing (ICS) 2024. Zeng, Jianping; Huang, Shao-Yu; Liu, Jiuyang; Jung, Changhee. The dissertation author was the first author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in International Symposium on Microarchitecture (MICRO) 2023. Zeng, Jianping; Jeong, Jungi; Jung, Changhee. The dissertation author was the first author of this paper.

Chapter 6, in full, is a reprint of the material as it appears in International Symposium on Computer Architecture (ISCA) 2024. Zeng, Jianping; Zhang, Tong; Jung, Changhee. The dissertation author was the first author of this paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Reliability against errors, such as soft errors—transient bit flips in transistors caused by energetic particle strikes—and crash inconsistency arising from power failure, is as crucial as performance and power efficiency for a wide range of computing devices, from embedded systems to datacenters. If not properly addressed, these errors can lead to massive financial losses and even endanger human lives. Furthermore, the dynamic nature of modern computing workloads complicates the implementation of reliable systems as the likelihood and impact of these errors increase. Consequently, system designers often face a dilemma: sacrificing reliability for performance and cost-effectiveness or incurring high manufacturing and/or run-time costs to maintain high system dependability. This trade-off can result in reduced availability and increased vulnerability to errors when reliability is not prioritized or escalated costs when it is.

In light of this, this dissertation, for the first time, demonstrates that with a synergistic compiler and architecture co-design, it is possible to achieve reliability while maintaining high performance and low hardware cost. We begin by illustrating how compiler/architecture co-design achieves near-zero-overhead soft error resilience for embedded cores (Chapter 2). Next, we introduce ReplayCache (Chapter 3), a software-only approach that ensures crash consistency for energy harvesting systems (backed with embedded cores) and outperforms the state-of-the-art by 9x. Apart from embedded cores, reliability for server-class cores is more vital due to their widespread adoption in performance-critical environments. With that in mind, we then propose VeriPipe (Chapter 4), which showcases how a straightforward microarchitectural technique can achieve near-zero-overhead soft error resilience for server-class cores with a storage overhead of just three registers and one countdown timer. Finally, we present two approaches to achieving performant crash consistency for server-class cores by leveraging existing dynamic register renaming in out-of-order cores (Chapter 5) and repurposing Intel's non-temporal path (Chapter 6), respectively. Through these innovations, this dissertation paves the way for more reliable and efficient computing systems, ensuring that reliability does not come at the cost of performance degradation or hardware complexity.

1. INTRODUCTION

Reliability against errors, such as soft errors [1–17]—transient bit flips in transistors caused by energetic particle strikes—and data inconsistency arising from power failure (so-called crash inconsistency) [18–25], is as important as performance and power for a variety of computing devices, ranging from embedded systems to datacenters. As the reliance on computing devices increases, the need for dependable operations also rises, especially when digital circuits become more vulnerable to errors in the era of post-Moore's Law.

If not dealt with properly, these errors could result in massive financial losses and even endanger human lives. For instance, in 2017, a brief power outage followed by a surge caused soft errors in the memory modules of Amazon's US-East-1 data center. This incident led to a system crash and widespread data inconsistency, affecting numerous services and customers who rely on Amazon's cloud infrastructure [26]. The financial impact of such failure could be substantial, as businesses dependent on cloud service would experience operational disruptions and even worse loss of critical customer data. Even more concerning are the potential safety implications of unreliable computing systems. In 2008, an Airbus A330 experienced a sudden uncommanded pitch-down event, which injured several passengers and crew members [27]. It turns out that radiation-induced soft errors caused incorrect data to be processed by the flight control computers, resulting in erroneous commands being issued to the aircraft's control surfaces. Such failure in aviation and other critical domains highlights the dire consequences of unreliable systems.

Unfortunately, achieving the threefold goals of reliability, high performance, and low hardware complexity simultaneously is a challenging, if not impossible, task. Previous solutions to enhance reliability [28–39] often come with significant trade-offs. For example, techniques like dual or triple modular redundancy (DMR/TMR) [36, 37] involve duplicating or triplicating critical components, which increase hardware costs and resource usage. Similarly, advanced error correction mechanisms [38, 40–43] embedded within hardware can recover program from soft errors on the fly, but they add complexity to the processor design and may slow down regular operations. Software-based approaches, such as checkpointing/rollback mechanisms [44–53] and logging techniques [54–58], can also mitigate the effects

of soft errors and crash inconsistency, but they typically require additional computational resources and time, which is not acceptable for performance-critical applications.

Even worse, such errors have a higher impact on modern applications due to their complexity, data intensiveness, and the increased likelihood of encountering errors. Thus, users might need to pay for even higher hardware cost and/or run-time overhead to maintain the reliability of the computing systems. As a result, computing system designers face a dilemma: compromising reliability for run-time performance and cost-effectiveness or incurring high hardware costs to maintain system dependability.

To this end, co-designing the relevant software and hardware would be a plausible approach as it achieves reliability while maintaining high performance and low hardware complexity. It aims to optimize both software algorithms and hardware architectures, thus mitigating the effect of errors. However, prior works in this area [40, 59, 60] still suffer from non-trivial hardware complexity and substantial run-time overhead, hindering their practical implementation in real silicon.

This dissertation steps forward further in this direction. For the first time, it shows that with synergistic compiler and architecture co-design, it is possible to achieve the above three goals at the same time. In particular, this dissertation reuses existing architectural components as much as possible. This not only maximizes the utilization of the existing architectural structures—they are often idle—but also avoids introducing new hardware structures to a large extent, making our techniques highly suitable for real silicon implementation. To better reuse the existing architectural structures, this dissertation devises sophisticated compiler techniques that transparently optimize user applications for the repurposed hardware structures, which would otherwise add more hardware complexity. *As a consequence, we seamlessly enhance the reliability of user applications while maintaining low hardware complexity and high performance.*

Having established the overarching importance of reliability, we now turn our focus to a specific type of computing systems.

1.1 Soft Error-Resilient Embedded Cores

Due to their simplicity and time predictability, in-order cores are widely used in controlling physical systems. Compared to their out-of-order (OoO) counterparts [61–66], however, the urgency to enhance the reliability of in-order cores has often been overlooked. In-order cores often depend on costly DMR/TMR solutions to mitigate the impact of soft errors [67–74]. Nevertheless, mission-critical embedded systems commonly powered by batteries in portable military devices, wearables, and drones must prioritize power efficiency, rendering the use of DMR/TMR impractical. Additionally, the size and weight constraints of these systems, particularly tiny aerial platforms like surveillance drones [75–77] and bionic birds [78, 79], make the integration of DMR/TMR more challenging. This highlights the urgent need for more efficient and lightweight error resilience solutions for in-order cores.

In line with our design objective of maximizing the use of existing architectural components to maintain low hardware complexity, this dissertation presents Turnpike (Chapter 2) that protects in-order cores against soft errors with a near-zero run-time overhead. Turnpike deploys acoustic sensors [80–84] on the chip to detect the occurrence of soft errors by perceiving the generated sound wave as a result of energetic particle striking, which is the only way to prevent silent data corruptions at a low hardware cost. Because of the detection latency, which is due to the propagation delay between striking location and sensor location, soft errors could escape from the core pipeline to the ECC-protected memory system [43, 85, 86], corrupting program states.

To address this issue, Turnpike reuses the core pipeline's existing store buffer (SB), which moves stores off the critical path of program execution, as a gated store buffer to contain soft errors in the core. That way, only error-free data can be released from the SB to the memory system. To recover program execution from soft errors, Turnpike compiler partitions input program into a series of recoverable regions with SB size in mind, which would otherwise overflow the SB. During region execution, Turnpike holds the region's stores in the SB till they are verified to be error-free, i.e., no soft error is detected within the worst-case detection latency (WCDL) since the region end, which is called *region verification*. Upon detecting soft errors, Turnpike discards the stores of unverified regions from the SB and resumes program

execution from the end of the latest verified region, since regions are verified in program order.

For the first time, Turnpike achieves a remarkable breakthrough using the harmonious co-design of its visionary compiler and architecture techniques. This synergy gets run-time overhead down to nearly zero for a vast array of applications, transforming the dream of real silicon implementation into a tangible reality. With this innovation, the cumbersome and costly DMR/TMR would become relics of the past. Building on the discussion of soft error resilience, we now explore another critical aspect of reliability—crash consistency—for embedded systems.

1.2 Crash-Consistent Embedded Cores

Energy harvesting systems (EHSs) [87], which are backed by low-power embedded in-order cores, have been deployed in a wide range of remote environments, e.g., river surveillance [88, 89] and health monitors [90–93]. EHSs are valued for their ability to operate for an ultra-long period of time without maintenance in that they collect energy from ambient sources like solar and thermal, obviating the need for batteries that are bulky, costly, and environment-unfriendly. However, EHSs suffer from unpredictable and frequent power failure as input energy sources are not stable, causing all volatile states to be lost and thus necessitating lightweight crash consistency mechanisms. For this reason, all existing EHSs [59, 94–98] employ register checkpointing—either by software or hardware—and use non-volatile memory (NVM) as main memory to preserve all program states across power failure.

Nevertheless, NVM is slow, leading to low performance of EHSs. To address this issue, a potential solution is to use volatile write-back caches that can hide long latency of NVM stores and improve performance by exploiting data locality through load hits. Unfortunately, volatile write-back caches have been considered non-viable or at least challenging in EHSs. The primary issue is that their data are lost upon power failure, potentially leading to inconsistent NVM states and preventing the interrupted program from resuming correctly.

To illustrate, suppose users try to insert a new node to the beginning of a doubly-linked list, which is done by (1) setting the new node's next pointer to the address of the old head

node and (2) resetting the old head node's previous pointer to the address of the new node. Now, assume a scenario where the second store persists in NVM with the cacheline evicted from the volatile write-back data cache while the first store is cached. If power is suddenly cut off here, the data stored in the cache is lost. This causes the new node's next pointer to become a dangling pointer, leading to inconsistent NVM states.

This concern explains why prior EHSs do not utilize volatile write-back caches. Moreover, ensuring crash consistency in a lightweight manner remains challenging for EHSs. For example, software logging introduces significant performance slowdowns (2-4x) as it flushes the data being stored to NVM in program order by inserting persist barriers, e.g., `clwb` and `sfence` in x86, before each regular store [54–58].

We found out the crux of crash inconsistency issue is the stores left behind power failure. To deal with these stores, this dissertation proposes ReplayCache (Chapter 3) that in the wake of power failure, replays the interrupted stores and resumes program execution from the failure point, ensuring lightweight crash consistency for EHSs in the presence of volatile write-back caches. *Consequently, though as a software-only scheme, ReplayCache, for the first time, enables volatile write-back caches for EHSs and hence significantly enhances their performance.* To ensure correct store replaying, ReplayCache exploits compiler to partition input program into a series of *store-integrity* regions, ensuring that the register operands of stores therein are not overwritten by other following instructions in the region. As such, ReplayCache correctly replays the interrupted region's stores to resume program execution from the power failure point, as volatile register states—including the program counter (PC)—are preserved across the power failure through checkpointing.

With its groundbreaking optimizations, ReplayCache surpasses the state-of-the-art work by an impressive factor of 9, establishing itself as the premier cache architecture for future commodity EHSs. This remarkable performance enhancement highlights ReplayCache's potential to revolutionize the efficiency and reliability of energy harvesting systems, paving the way for its broader adoption and more robust applications in remote and challenging environments.

Shifting our focus from embedded systems to more powerful computing devices, we next consider how to address the reliability issues of server-class cores.

1.3 Soft Error-Resilient Server-Class Cores

Server-class out-of-order cores are extensively used in a wide range of scenarios, from personal computers to datacenters, owing to their capability to meet the high performance demands of complex and dynamic user applications. Despite their widespread adoption, no commercial processors incorporate lightweight yet efficient techniques for protecting the core pipeline against soft errors. This omission is due to the significant slowdown they can introduce to the highly optimized pipeline execution, which is a critical issue given that single-core processor performance has plateaued in the post-Moore's Law era. As a result, the reliability of these processors is compromised, leading to decreased availability and a negative impact on user experience.

To achieve high-performance soft error resilience for out-of-order cores, one might consider adapting Turnpike to them. However, this approach presents significant challenges: (1) it requires overly complex hardware structures to dynamically determine the verification point of each region, i.e., tracking if WCDL cycles are elapsed since the end of each region to release the region's stores from the SB to the L1 data cache; and (2) these intricate structures strain the highly optimized timing of the out-of-order pipeline, resulting in unacceptable performance degradation for performance-sensitive applications.

To address these issues, this dissertation proposes VeriPipe (detailed in Chapter 4), a near-zero-overhead resilience scheme that uses acoustic-sensor-based detection to protect out-of-order cores against soft errors. Unlike Turnpike, VeriPipe ensures that each region is always verified at the end of the following region, which is statically determined at compile time and thus circumvents the need to track the verification point of each region at run-time time. As a consequence, this technique dramatically simplifies the hardware complexity to just three registers and one countdown timer, which relieves the pressure on the core pipeline and thus reduces run-time overhead to only 1% along with compiler optimizations, marking a significant step forward in the development of reliable server-class processors.

We believe that VeriPipe is ready for deployment in real silicon, poised to protect all server-class cores against soft errors with minimal overhead. This advancement heralds a

new era of reliable, high-performance computing, ensuring that server-class cores remain robust and efficient in the face of increasingly complex demands on reliability.

1.4 Crash-Consistent Server-Class Cores

Beyond soft errors, crash inconsistency across power failure is another significant concern in those datacenters [99–106] that are equipped with out-of-order cores and Intel Optane memory (PMEM) [107–109]. In such systems, PMEM is used in memory mode, where DRAM works as an off-chip direct-mapped last-level cache (LLC) while NVM serves as main memory. This configuration deepens the memory hierarchy and enhances performance for memory-intensive applications. However, the naive use of PMEM can lead to potential crash inconsistency, which is the same issue faced when enabling volatile write-back caches for EHSs.

Guided by the design goal of reusing existing architectural structures as much as possible, this dissertation presents two proposals, PPA and cWSP, from the perspectives of hardware and software, respectively, to transparently achieve lightweight yet performant crash consistency and persistency for any program—which is so-called whole-system persistence (WSP) [60, 110, 111]—running on server-class cores backed with PMEM.

Hardware-Driven Scheme: Almost all modern high-performance processors exploit out-of-order execution to extract more instruction-level parallelism (ILP). A pivotal feature of out-of-order execution is dynamic register renaming, which eliminates false register dependencies such as write-after-read (WAR) and write-after-write (WAW) dependencies. By resolving these dependencies, instructions can be executed as soon as their operands are available. With that in mind, this dissertation proposes PPA (Chapter 5) that enforces *store integrity*—the key to correct store replaying and crash consistency guarantee—at the hardware level. Specifically, PPA makes use of the existing dynamic register renaming to achieve *store integrity*. For this purpose, PPA does not reclaim the physical registers associated with prior committed stores, ensuring that the register values remain unchanged. When the physical register file runs out, PPA delineates a region boundary here, forming a store-integrity region. At the end of the region, PPA waits for all prior stores to be persisted

in NVM and then frees their registers for later use. To minimize such pipeline stalls at each region boundary and achieve high performance, PPA asynchronously persists prior stores, overlapping store persistence latency with the execution of other instructions in the region.

Upon power failure, PPA just-in-time checkpoints minimal architectural states, e.g., store registers, to NVM. That way, in the wake of power failure, PPA resumes the interrupted region from the last committed point—instructions after the last committed point are in-flight and their states are not visible to software—after replaying the region's stores, as their register values remain intact across power failure. Consequently, PPA accomplishes performant WSP even for legacy applications owing to hardware-only nature.

Despite the significant advantages, PPA incurs a non-trivial run-time overhead for compute-intensive applications—which demand many physical registers—in that the formed regions are not long enough to cover store persistence latency due to the lack of free physical registers. Additionally, PPA requires securing energy for just-in-time checkpointing, which complicates hardware design. To address these challenges, we propose a complementary solution: a compiler-driven approach.

Compiler-Driven Scheme: This dissertation proposes cWSP (details are deferred to Chapter 6), compiler-directed whole-system persistence, which achieves lightweight yet high-performance WSP for all. cWSP significantly lowers storage overhead to a few hundred bytes while eliminating just-in-time checkpointing, making WSP a viable solution for any type of out-of-order core. To achieve performant store persistence, cWSP repurposes Intel's non-temporal path [112] as a dedicated persist path of NVM stores—and thus drop their dirty cacheline evictions from LLC—to deliver the data to NVM in order, achieving *strict persistency* [113] on the cheap without using expensive persist barriers, e.g., `clwb` and `sfence` in x86 processors. To recover program execution from power failure, cWSP compiler partitions input program into a series of idempotent regions [114–116]—which do not have memory WAR dependence—allowing them to recover from power failure through re-execution. Thus, in the wake of power failure, cWSP can resume the power-interrupted program from the end of the latest persisted region (i.e., its all stores have already been persisted in NVM).

The lightweight yet performant WSP provided by PPA and cWSP not only eliminates the cognitive burden of error-prone persistent programming [20–25, 59, 117–122]—which

often requires manual modifications on program source code—but also accelerates persistent applications through the use of DRAM cache; this is impossible in PMEM's app-direct mode where DRAM and PMEM are managed by OS in separate address spaces. For users leveraging memory mode to benefit from a deep cache hierarchy, PPA and cWSP offer persistency and crash consistency without compromising transparency or performance. This is particularly beneficial for applications such as HPC (high-performance computing) applications and in-memory databases, where states must be regularly saved to storage. The lightweight persistency and recoverability provided by these schemes can enable high-performance application-level resilience, obviating the need for periodic global checkpointing to storage, which typically takes over 26 hours for HPC program and is expected to increase as program data size grows [123].

1.5 Outline

In the rest of this dissertation, Chapter 2 elaborates on how Turnpike achieves performant soft error resilience for in-order cores through compiler/architecture co-design. Chapter 3 explores how ReplayCache enables a volatile writeback cache for EHSs while ensuring crash consistency. Chapter 4 details how VeriPipe achieves soft error resilience for out-of-order cores with minimal hardware costs and a near-zero run-time overhead. Chapter 5 discusses how PPA achieves crash consistency and persistency for all kinds of program at microarchitecture level. Chapter 6 presents how cWSP minimizes hardware complexity while maintaining high performance. Finally, Chapter 7 concludes the dissertation.

2. TURNPIKE: LIGHTWEIGHT SOFT ERROR RESILIENCE FOR IN-ORDER CORES

Soft error resilience is becoming more important than ever. With technology scaling, circuits are likely to be more sensitive to radiation-induced soft errors; they are mostly caused by energetic particles (e.g., cosmic rays) and alpha particles from packaging materials [80–83, 124–126]. Soft errors may lead to a system crash or even worse silent data corruptions (SDC) that are not caught by the error detection logic but end up with incorrect outputs. Due to the high availability requirement of embedded systems, soft error resilience has been one of the most important design considerations.

Among existing soft error resilience schemes, acoustic-sensor-based detection [40, 42, 80–84, 127–131] is particularly promising. To the best of our knowledge, it is the only way to prevent SDC—that is a long-awaited open problem—at a low hardware cost. Since acoustic sensors perceive the sound wave of particle strikes, which is always generated as a physical phenomenon, no resulting soft error is missed. As such, the sensor-based detection can achieve SDC freedom; unlike other schemes, it does not even require any microarchitecture replication. Moreover, sensors occupy only a very small die size area. For example, 300 sensors are enough to achieve 30 cycles of the worst-case detection latency (WCDL) for a 2GHz out-of-order core, and they only cause ∼1% area overhead [80–84, 127].

With that in mind, Liu *et al* [40] show how their solution Turnstile can leverage acoustic sensors for core-level error containment with little architecture change for soft error verification/recovery. The rationale for verifying the absence of soft errors is that since each error is to be detected within WCDL after its occurrence, execution prior to a given time T will be verified to be error-free at a time T+WCDL, if no error is detected during the WCDL.

In light of this, Turnstile verifies every data being stored to memory, ensuring that it has not been affected by soft errors before its write-back. Although a re-order buffer (ROB) retires a store instructions, the data is not written back to memory but held in a store buffer until it turns out to be verified waiting for WCDL. For register verification, Turnstile reformulates it based on the aforementioned memory verification by inserting stores to checkpoint updated live-out registers and holding them in the store buffer. The upshot

is that register write-backs are never delayed for verification, which would otherwise slow down the pipeline execution. Since stores are rarely on the critical path in out-of-order cores, Turnstile can offer lightweight soft error resilience at ≈8% performance overhead on average for SPEC2006/MediaBench/SPLASH2 benchmarks.

Compared to out-of-order (OoO) cores, however, there has been less attention received to enhance the reliability of in-order cores in a low-cost manner—though they are widely deployed in embedded systems to control the physical world. For example, while in-order cores are used for adaptive cruise control, precrash safety alarm, and motion planning systems due to the simple hardware and the time predictability demand excluding complex OoO execution [61–66], they still rely on expensive dual/triple modular redundancy (DMR/TMR) to deal with soft errors [67–74]. Nonetheless, mission-critical embedded systems should pursue power-efficiency as they are often battery operated, e.g., portable military devices, wearables, and drones, preventing the use of DMR/TMR. Apart from that, DMR/TMR could suffer the size and weight issues that are particularly critical for tiny aerial systems such as spying drones [75–77] and bionic birds [78, 79].

With the increasing demand for lightweight soft error resilience for in-order cores, one might want to leverage Turnstile on top of in-order cores. Unfortunately, naively adapting Turnstile to in-order cores causes a significant performance overhead, i.e., 29%-84% for 10-50 cycles of WCDL. The main reason is that the in-order pipeline stalls for the structural/data hazards of stores due to the inability to schedule other independent instructions. Since the store buffer of in-order cores is very small (4 entries) unlike that of out-of-order cores (40 or more entries), it often becomes full during the verification, in which case the pipeline stalls on the next store due to the structural hazard until some of the buffered stores are verified and flushed to L1 cache. Similarly, for the execution of a checkpoint, i.e., essentially a store instruction to save a register value, the in-order pipeline may stall waiting for the value to be available. This data hazard happens a lot leading to significant performance degradation, because Turnstile inserts the checkpoint right after the register-update instruction, e.g., a delinquent load.

To address the problems, this chapter presents Turnpike, a compiler/architecture co-design scheme that can achieve a lightweight yet guaranteed soft error resilience for in-order

cores. Turnpike leverages 3 key insights to minimize the pipeline stalls with lowering the store buffer pressure. First, during code generation, it is possible to decrease the number of stores to be verified. Turnpike's compiler optimizations remove unnecessary checkpoint stores, e.g., those whose value can be reconstructed from other checkpointed values at the recovery time [132, 133], without compromising the recoverability. We also propose 2 novel compiler optimizations to suppress the generation of stores: loop induction variable merging for reducing live registers being checkpointed in a loop, and store-aware register allocation for less register-spilling stores.

Second, the compiler can reduce the execution delay of unremoved checkpoint stores with the help of instruction scheduling for resolving the checkpoint data hazard. That is, Turnpike attempts to separate the live register-update instructions from their dependent checkpoint stores by filling the gap with other independent instructions. This gives the in-order core an illusion that it can hide the execution delay of the checkpoint as in out-of-order execution.

Third, many of the remaining stores can be safely released to cache without waiting for verification, no matter if they are regular stores or checkpoint stores. For example, some value being stored is never used for the recovery of a soft error—even if it corrupts the value. To take advantage of this insight, we introduce simple hardware support that can (1) conduct the safety check for the fast (early) release of a given store and, if possible, (2) let it go through the fast path[1], i.e., immediately flushing it to cache bypassing its verification. Along with the above compiler optimizations, this hardware support can relieve the pressure on the small store buffer of in-order cores and thus reduce its structural hazards effectively. Experiments with 36 benchmarks from SPEC2006/2017/SPLASH3 suites highlight Turnpike's low performance overhead, i.e., 0% and 14% on average—while Turnstile's overhead is 29% and 84%—for 10 and 50 cycles of WCDL, respectively. Our contributions are below:

- Turnpike is the first to make acoustic-sensor-based soft error resilience work for in-order cores at a low HW/run-time cost.

[1]↑The fast path can be regarded as an electronic toll collection lane on turnpike. The name Turnpike is inspired by this analogy.

- We show how compiler optimizations are used not only to remove unnecessary checkpoints but also to reduce the execution cycle of the unremoved checkpoints.
- We propose 2 novel compiler optimizations to lower the number of registers being checkpointed and reduce register-spilling stores during register allocation.
- We propose 2 new hardware schemes to bypass store verification without compromising resilience guarantee.

2.1 Background and Challenges

2.1.1 Region-Level Soft Error Verification

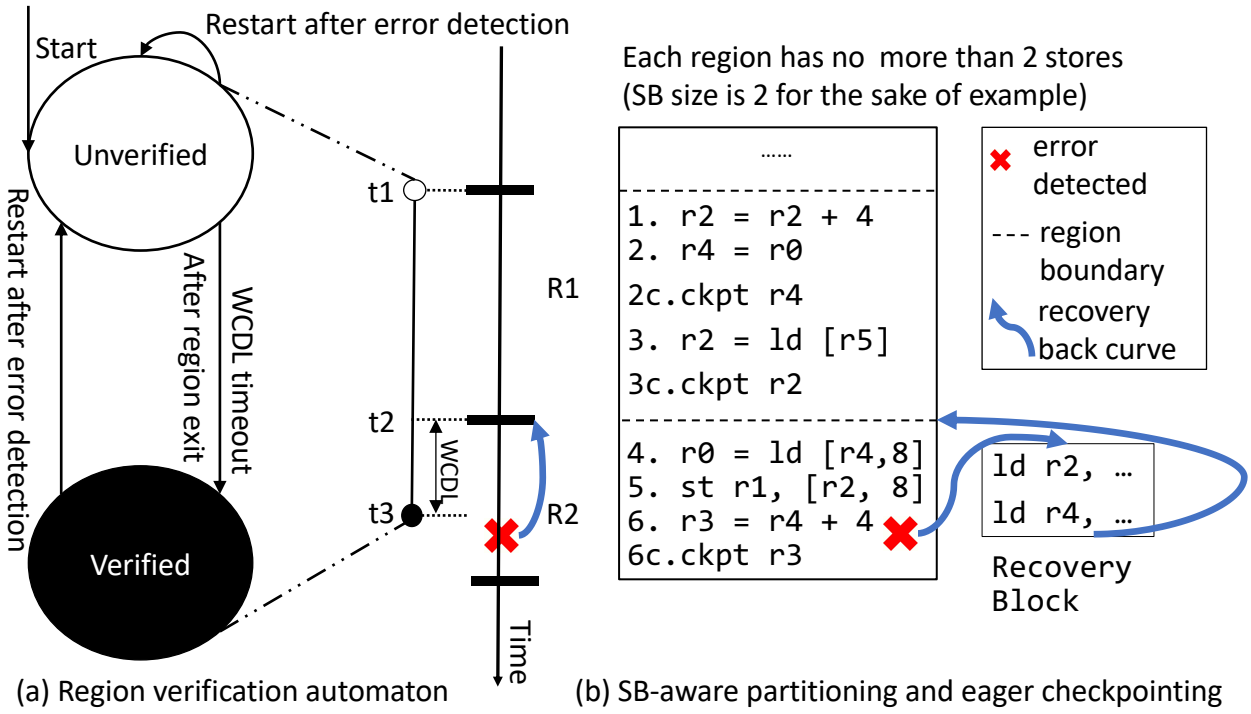


Figure 2.1. (a) Turnstile's region verification automaton; (b) store buffer aware region partitioning; eager checkpointing

To realize the sensor-based soft error verification at a low cost, Turnstile's compiler partitions the entire program into a series of verifiable/recoverable regions with the store buffer (SB) in mind so that each region cannot have more stores than the SB size [40]. As shown in Figure 2.1 (a), each started region is treated as unverified at the beginning, e.g., $R1$

gets $Unverified$ state at time $t1$. Thus, Turnstile prevents all the stores of the region from being merged to cache until the region is verified to be error-free. That is, no sensor detects an error during the worst-case detection latency (WCDL)—e.g., from $t2$ to $t3$—after the region is finished. That way Turnstile can contain all the errors occurred during the execution of a region within the core, keeping cache/memory intact. Furthermore, this allows Turnstile to correct an error by simply reading verified data from cache/memory protected by ECC in modern processors including even low-power cores such as ARM Cortex series.

For the in-core error containment,Turnstile leverages its SB as a gated store buffer (GSB) [59, 134]; hereafter, SB refers to GSB. That is, it holds by default all store write-backs for quarantine even after ROB retires the stores. To get them out of the SB quarantine, if verified (i.e., no error detected during WCDL time after the end of their region), Turnstile devises a region boundary buffer (RBB) shown in Figure 2.2. Whenever a region boundary is encountered, i.e., one region finishes and the next starts as at $t2$ in Figure 2.1 (a), Turnstile allocates the RBB entry to delineate the previously quarantined stores that will be released on their region verification. Especially when a region is verified, e.g., $R1$ at $t3$, the RBB marks the boundary, at which the verified region has ended, as a *recovery PC* in case of a future error.

2.1.2 Eager Checkpointing and Error Recovery

The Turnstile compiler performs so-called eager checkpointing [40] that immediately saves updated live-out registers in memory. That is, it inserts a checkpoint store right after the register-update instruction provided the register is used as the input of later regions, e.g., at line 2c, 3c, and 6c in Figure 2.1 (b). The implication is three-fold. First, even if a region has multiple updates of a register, only the last one is checkpointed as the live-out register, e.g., Turnstile checkpoints only the definition of $r2$ at line 3 though it is pre-defined at line 1 in the figure. Second, Turnstile can turn register verification into memory verification; registers are verified through their checkpoint—which is essentially a store instruction—in the same way as stores are verified, without delaying any register write-back for performance reasons. Third, the checkpointed register values should be loaded to recover from a soft error.

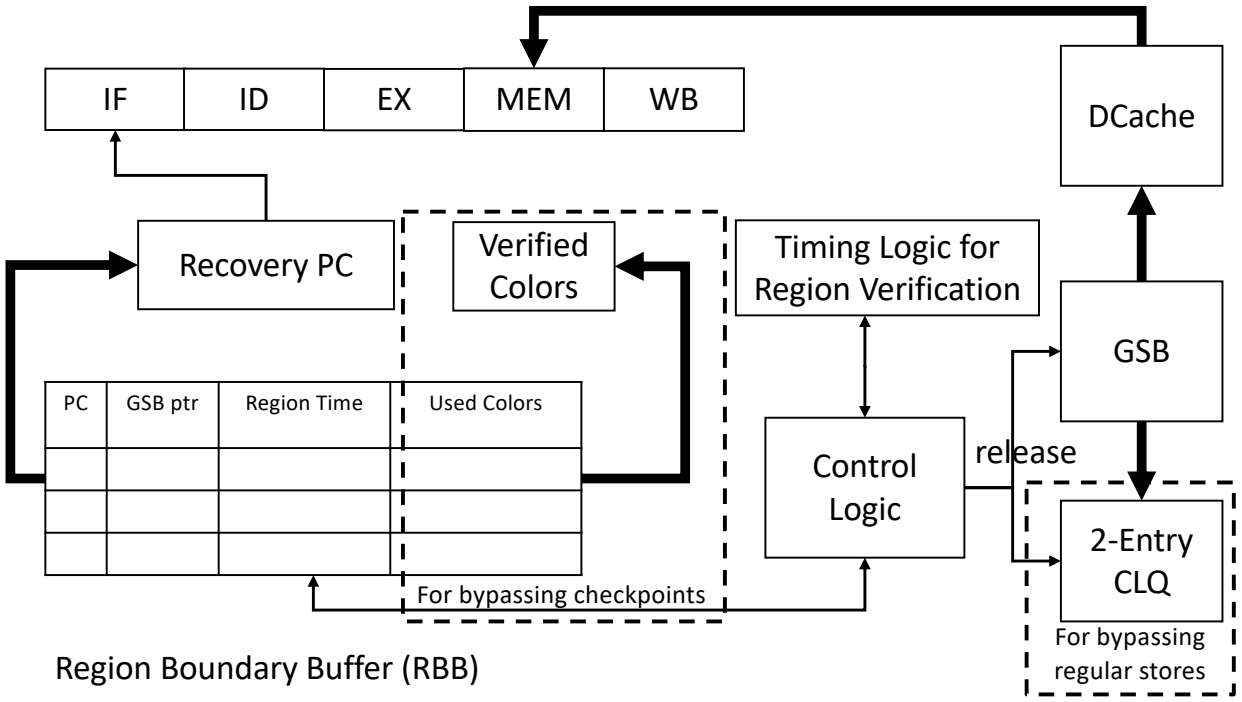


Figure 2.2. The high-level view of **Turnpike**; bold lines correspond to data paths while thin lines to control paths

Upon the detection of a soft error, Turnstile first discards all SB entries—because they could have been corrupted by the error—and identifies the most recently verified region boundary by referring to the *recovery PC* and the region starting thereafter. As shown in Figure 2.1 (b), Turnstile then executes the recovery block of the region to restore its input (live-in) registers, e.g., $r2, r4$ in the figure, from the ECC-protected memory (cache) where their checkpoints have been stored safely with the in-core error containment. Finally, Turnstile restarts the region recovering from the error. This soft error verification of Turnstile works well for out-of-order cores. However, it incurs a significant run-time overhead for in-order cores as shown in the next section.

2.1.3 Limitations of Prior Work

To equip the verifiable regions—partitioned with the SB size in mind—with the recoverability, Turnstile checkpoints live-out register values of each region by logging them to

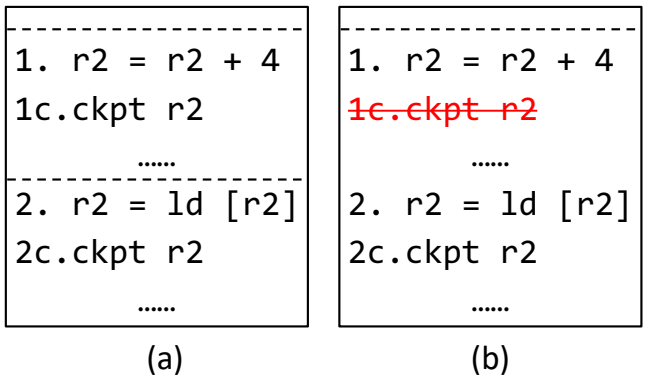


Figure 2.3. Impact of region size

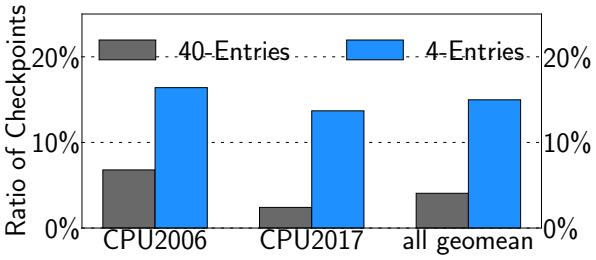


Figure 2.4. Impact of store buffer size on the number of inserted checkpoints

memory (Section 2.1.2). Since the regions are generally short due to the tiny SB (only 4 entries in modern in-order cores), the short regions tend to have more live-out registers updated overall than the long regions for a large SB. For example, while a register $r2$ is live-out in both regions and thus checkpointed twice at line 1c and 2c in Figure 2.3 (a), it is checkpointed only once in the same code that does not have a region boundary in-between as shown in Figure 2.3 (b); that is because $r2$ defined at line 1 is no longer live-out since it is overwritten by the following definition at line 2.

Figure 2.4 confirms that the number of inserted checkpoints (i.e., store instructions logging the live-out registers) significantly increases when the store buffer is shrunk from 40 to 4 entries. When the store buffer (SB) size is 40 as in out-of-order cores, Turnstile's eager checkpointing accounts for 4.1% of the total dynamic instruction count on average for SPEC 2006/2017 benchmark applications. On the other hand, when the SB size is 4 as in in-order cores, the ratio significantly increases to 14.98%. It turns out that such many checkpoints often fill up the SB, making the next store stall the pipeline due to the lack of room in the SB, i.e., the structural hazard. In particular, we found it possible to remove many of the checkpoints without compromising the soft error resilience; Section 2.2.1 discusses it in detail.

In addition, to verify each region (ensuring the absence of soft errors during the region execution), Turnstile holds all the data being stored in the SB till the region is verified to

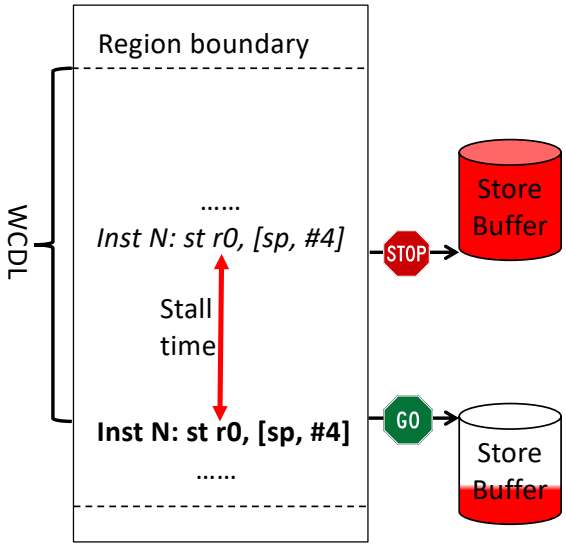


Figure 2.5. Stall due to the lack of room in the SB during the region verification

be error-free. Hence, no allocated SB entries of stores can be released to L1 cache during the execution of their region; rather, they can only be released WCDL (10-50) cycles later after the region ends. The implication is that stores cannot but reside in the SB for such a long period of verification time, keeping the high pressure on the SB. In essence, this may cause a structural hazard if the SB has already been full when the pipeline encounters a new store, e.g., `inst N: st` in Figure 2.5. Unfortunately, the hazard cannot be resolved until the prior region is verified with its stores released to cache. In other words, the pipeline stall continues all the way to the region verification point—where the WCDL time elapses in the figure. Here, due to the in-order nature of the pipeline, it cannot schedule any of the following instructions thus being unable to hide such a long stall latency. As such, it postpones not only the stalled store instruction, e.g., `inst N: st` in the figure, but also all the subsequent instructions. As will be shown in Section 2.2.3, Turnpike can safely release some stores from the SB without holding them for verification, thereby relieving the store buffer pressure.

Moreover, Turnstile's eager checkpointing introduces a read-after-write dependence between the instruction, that updates a live-out register, and its checkpoint store. This is

particularly harmful for the in-order pipeline because of the inability to dynamically schedule other independent instructions—unlike the out-of-order pipeline that can overlap their execution with the live-register-update instruction. In an in-order core, checkpoint stores can often be stalled since the register being checkpointed is not available for their execution.

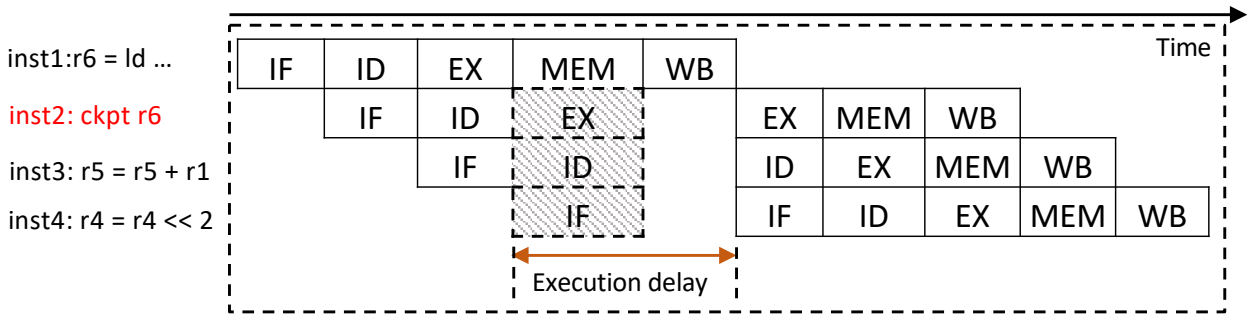


Figure 2.6. Checkpoint's execution delay on in-order pipeline

In such a case, the in-order pipeline must be delayed for a certain time to resolve the data hazard, e.g., till the value of register $r6$ becomes available in Figure 2.6 where checkpoint store is marked in red. Since it is the load instruction that updates the $r6$ in this example, the execution delay of the checkpoint store could be significant on cache misses. The takeaway is that such a checkpoint execution delay translates to the significant extension of the program execution time; Section 2.2.2 shows how Turnpike reduces the delay to make the checkpoint store instruction execute faster.

2.2 Implementation

To address the 3 problems in Section 2.1.3, our proposal, Turnpike, leverages compiler and architectural optimizations; Figure 2.7 shows the workflow of the 3 optimization phases.

In the first phase (Section 2.2.1), Turnpike's 2 new compiler optimizations reduce the traffic to store buffer by (1) generating less spilling stores during register allocation and (2) eliminating a loop induction variable being checkpointed if it can be merged with others. Likewise, Turnpike removes unnecessary checkpoints with two existing compiler optimizations, checkpoint pruning [133] and loop invariant code motion (LICM) [135] to further lower the store buffer traffic. Second, for the remaining checkpoints that cannot be removed by the

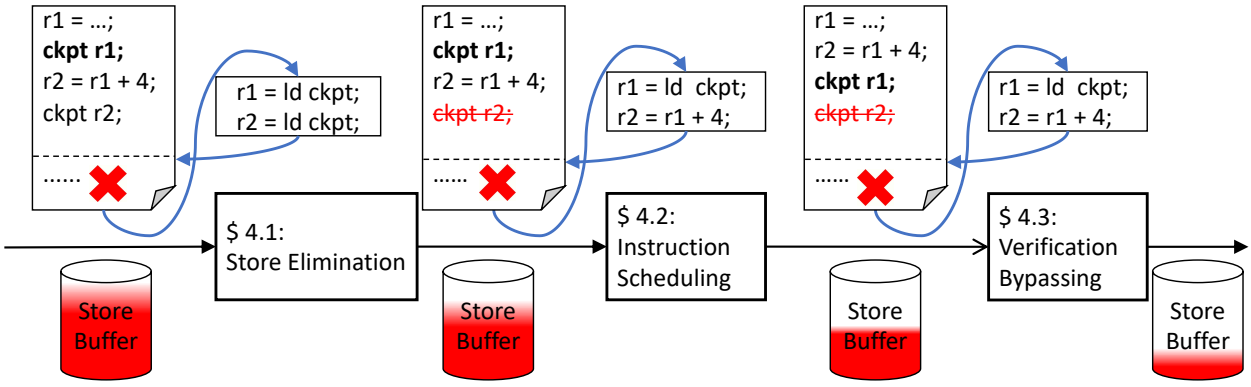


Figure 2.7. The 3 phases of Turnpike SW/HW optimizations

first phase, the Turnpike compiler performs checkpoint-aware instruction scheduling to hide the delay caused by data hazards (Section 2.2.2). Finally, to directly reduce the pressure on the store buffer (SB), Turnpike leverages 2 novel hardware techniques—in Section 2.2.3—that can (1) skip the verification of the remaining checkpoint stores and the regular stores and (2) merge them to cache right after they are committed. Note that the above 3 optimization phases have a synergistic impact on reducing the SB pressure. The rest of this section details the 3 optimizations.

2.2.1 Reducing the Traffic to the Store Buffer

This section shows how to reduce the SB traffic with 2 new compiler optimizations and 2 other existing ones. While the first addresses regular stores, the next 3 optimizations do checkpoint stores.

Store-Aware Register Allocation: In addition to application stores, the other source of regular stores is register allocation. Since it is done in a best effort manner, some variables end up being spilled to stack memory when architectural registers run out during the register allocation. To avoid spilling performance-critical variables, traditional register allocators takes a heuristic approach to determine what to spill. More precisely, they maintain the spill cost (weight) of variables which summarizes the execution frequency of their use points (reads and writes). Unfortunately, since the spill code models of traditional register allocators do

not differentiate writes from reads, they may generate superfluous spilling stores. While this is not a concern for most processors where stores are off the critical path, in-order cores equipped with sensor based soft error verification can suffer a significant performance degradation. To address this problem, the Turnpike compiler increases the cost for the write operation of each variable in the spill candidate decision logic. Note that care must be taken to maintain the original register allocation quality in terms of the number of spilled variables, which would otherwise degrade the performance of the resulting code. As a result, Turnpike can keep those variables, that are frequently written, in architectural registers, and thus all the writes to the variables just become register writes other than memory stores.

Loop Induction Variable Merging (LIVM): We found that traditional compilers often generate additional loop induction variables—that must be checkpointed each loop iteration—and the resulting checkpoint stores increase the store buffer traffic significantly. There are two kinds of induction variables [135]: basic induction variable, e.g., i in Figure 2.8 (a) and induced induction variable whose value is a (linear) function of a basic induction variable, e.g., the address expression of A[i] in Figure 2.8 (a).

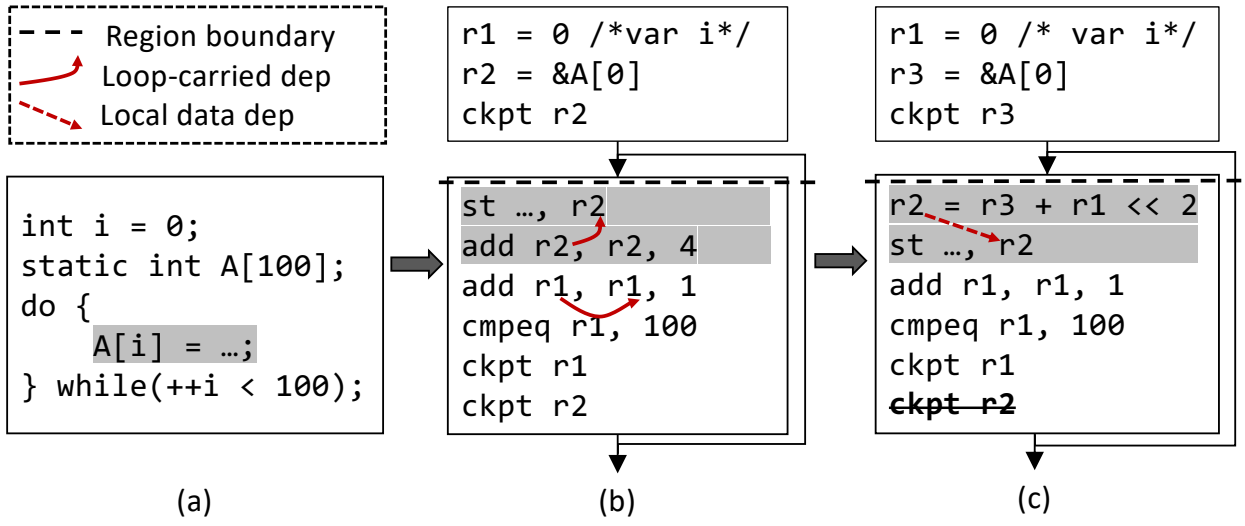


Figure 2.8. (a) original C code, (b) strength reduction code, and (c) LIVM enabled code eliminating $r2$'s checkpoint

Here, the compiler's loop strength reduction [135] turns the expression ($\&A[0]+\mathrm{i}*4$) into a separate basic induction variable that is initialized as $\&A[0]$ and increased by 4 as shown

in Figure 2.8 (b). The problem is that the strength reduction results in loop-carried data dependence, i.e., $r2$ is used in the next iteration as the address operand of a store, rendering $r2$ live-out [2] and checkpointed in the loop thus degrading the performance; Figure 2.8 (b) highlights the strength reduction enabled code in the shaded box and shows the resulting checkpoint, i.e., *ckpt* $r2$, in the bottom.

To deal with the problem, Turnpike proposes a new optimization called loop induction variable merging (LIVM). It investigates basic induction variables in a loop to see if one can be merged to some other basic induction variable in form of an expression derived from the basic induction variable. In other words, LIVM makes the merged variable become an induced induction variable so that it can eliminate the loop-carried data dependence. As shown in Figure 2.8 (c), $r2$ has only local data dependence with the help of LIVM; since $r2$ is no longer live-out, Turnpike eliminates the *ckpt* $r2$ in the bottom of the figure. Since it used to be executed every loop iteration, the impact of its elimination on the store buffer traffic reduction should be very significant if enabled.

Optimal Checkpoint Pruning: To further reduces checkpoint stores, Turnpike leverages optimal checkpoint pruning [132] in the recent advance of GPU register file protection called Penny. We found the pruning technique effective for reducing the store buffer pressure, though it is originally devised for GPUs—that have never had a store buffer (SB)—and idempotent regions [132] that are intrinsically different from Turnpike's SB-size aware partitioned regions.

It turns out that Penny's checkpoint pruning removes a large number of checkpoints without compromising the recoverability guarantee. The key idea is that it is safe to remove those checkpoints, provided the value to be checkpointed can be reconstructed from a constant or the value of other checkpoints at the recovery time of an error detected. Turnpike exploits Penny's optimal pruning algorithm that can detect unnecessary checkpoints in polynomial time with the recovery code generated to reconstruct the pruned checkpoint value.

Figure 2.9 shows how the checkpoint pruning works and ensures safe recovery. Suppose an error is detected in a region R2 in the bottom of the figure; R2's input registers $r1, r2, r3$ have been checkpointed by prior regions, e.g., the first region R0 checkpoints $r2$ and $r3$.

[2]↑A region boundary is placed in a loop header as in Turnstile [40].

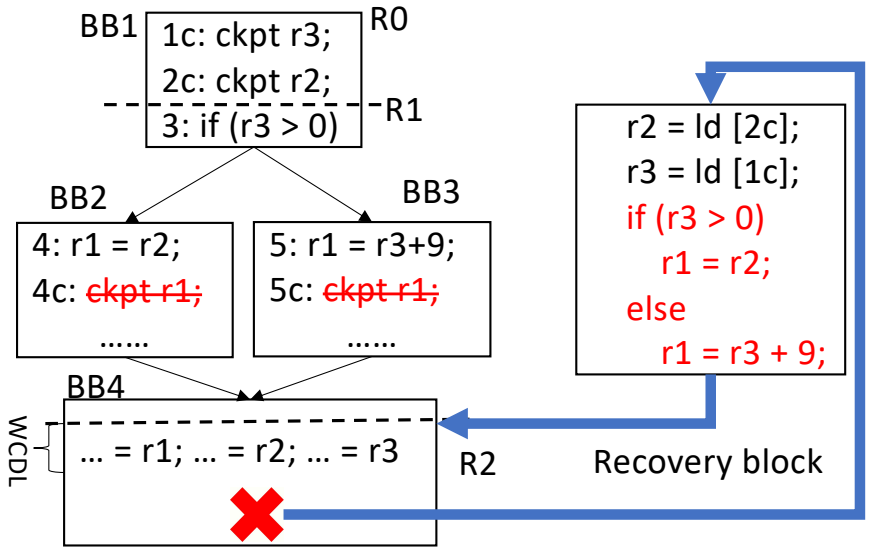


Figure 2.9. Checkpoint pruning for eliminating $4c$ and $5c$

Similarly, without the checkpoint pruning, $r1$ would be checkpointed by the middle region R1 where either $r2$ or $r3$ is used to update $r1$ depending on the path taken in the branch. Here, either way, $r1$'s checkpoint ($4c$ and $5c$) can be removed since it can be reconstructed by using the checkpointed value of $r2$ or $r3$ in the recovery block. To recover from the error here, the recovery block of the region R2—starting from the recovery PC (Section 2.1)—executes the backward slice of the pruned checkpoint, which includes the branch to reconstruct $r1$ differently according to the checkpointed predicate $r3$, and jumps back to the recovery PC, i.e., the beginning of the region R2.

Moving a Checkpoint out of a Loop with LICM: Although the pruning scheme investigates if a checkpoint can be safely eliminated, it never tries to move the location of the checkpoint. For those checkpoints that cannot be eliminated, the pruning scheme leaves them at their original checkpointing location, thus losing a chance to move a checkpoint out of the loop body. The main reason for this is that under the eager checkpointing policy, a checkpoint cannot but be placed right after the instruction that updates the live-out register in each region.

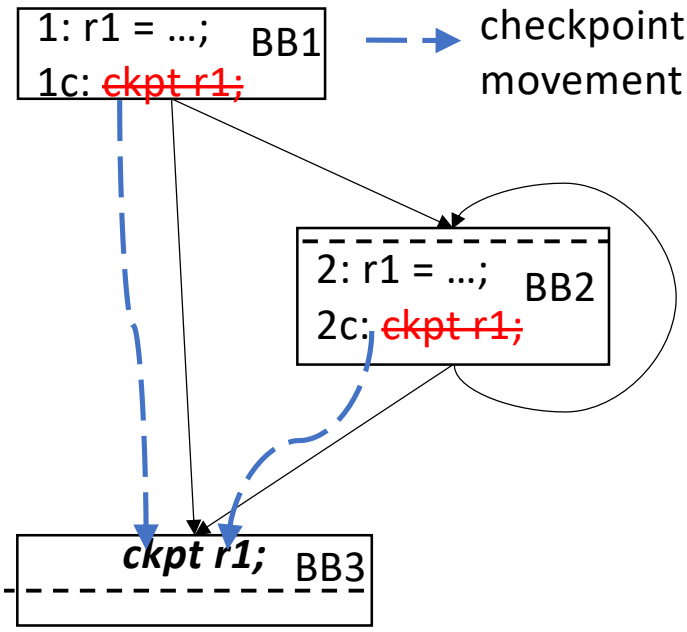


Figure 2.10. LICM at $2c$

Interestingly, the eager checkpointing can be relaxed without compromising the recoverability. Recall that a checkpoint is necessary for 2 reasons: (1) saving the registers that are input to some later regions and (2) verifying the integrity of the register value, i.e., no register corruption. In the input-saving point of view alone, Turnpike only has to checkpoint the register before it is used. On the other hand, to verify the register, it must be saved before the region is finished due to the region-level verification. As a result, for each checkpoint in a given region, the checkpoint can be safely moved from the original eager checkpointing location down to any points before the region boundary.

Figure 2.10 shows how Turnpike leverages this insight to move $r1$'s checkpoint at line $2c$ out of a loop as in LICM (loop invariant code motion[3]). By moving the checkpoint down to near the region boundary below, Turnpike takes the checkpoint off from the loop body. Moreover, since the checkpoint is now placed in the bottom basic block, another checkpoint at line $1c$ becomes redundant—as they both checkpoint the same value of $r1$—and can be safely

[3]↑ While LICM is to hoist the invariant code out of a loop [136, 137], most of the production compilers (GCC/LLVM) have extended LICM to support code sinking too. We modified the LLVM passes to move down checkpoints in a loop as they are guaranteed not to be aliased with other memory operations.

eliminated as well. That way Turnpike can reduce the performance overhead significantly for some applications (Section 2.4.2).

2.2.2 Hiding the Execution Delay of Checkpoints

There are still many remaining checkpoints that cannot be removed by the prior 2 optimizations. Since Turnstile inserts each checkpoint store right after the register-update instruction, the store's dependence on the register (data hazard) often makes the in-order pipeline stall till the register gets ready (see Section 2.1.3). To address this problem, Turnpike leverages another compiler optimization, i.e., instruction scheduling [135–137]. It attempts to separate the register-update instruction from the dependent checkpoint store instruction by hoisting some of the following independent instructions beyond the store instruction.

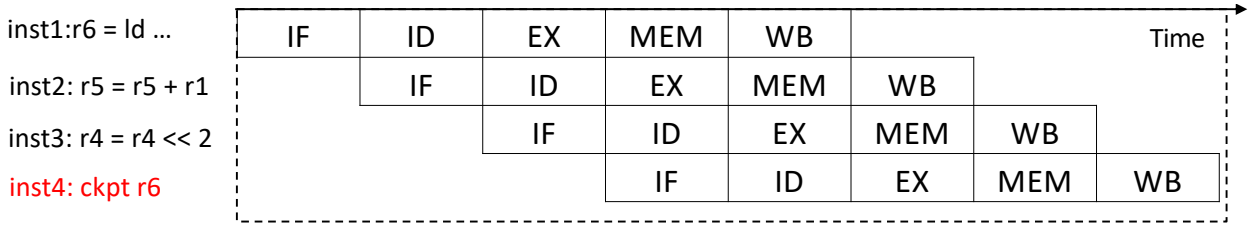


Figure 2.11. Execution delay of checkpoint gets reduced by rescheduling instruction stream

Figure 2.11 shows how the instruction scheduling can handle the checkpoint data hazard in Figure 2.6. With the scheduling, the checkpoint store for a register $r6$ being loaded is moved down in Figure 2.11; that way the store can be executed with no stall, i.e., its operand $r6$ is ready from the load—because the load latency is overlapped with the execution of 2 other intervening instructions before the store. Since the register becomes available when the reordered store is about to execute, it can avoid the data hazard. Note that the instruction scheduling helps the pressure on the store buffer (SB) to be relieved as well. The reason is that the reordered stores can eventually reduce the execution time of their region, which is the part of the region verification time; it consists of the region execution time and the WCDL as shown in Figure 4.2(a). Hence, this also reduces the time during which stores stay in the SB for verification, keeping the pressure for a shorter amount of time.

2.2.3 Relieving the Store Buffer Pressure

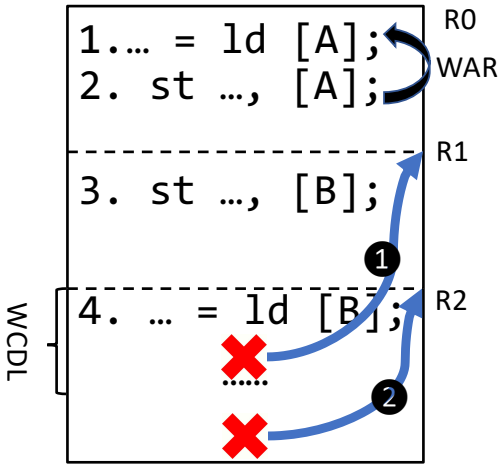


Figure 2.12. Fast release of a WAR-free regular store

Unlike other optimizations, the next two novel hardware schemes in this section can directly relieve the store buffer pressure. Turnpike achieves that by releasing some of the buffered stores to cache without verification yet in a manner that still guarantees the soft error resilience. To begin with, we classify store instructions into two kinds: (1) regular stores stemming from the program itself or register allocation, i.e., spill stores to stack, and (2) checkpoint stores generated to save updated live-out registers. The rest of this section discusses the two kinds and how they are addressed by our 2 hardware schemes, respectively.

Fast Release of Regular Stores: Prior work, Turnstile, has all stores of a region quarantined in a store buffer (SB) till the region turns out to be error-free for both region verification and in-core error containment purposes (Section 2.1). However, we found out that not all the data being stored are going to be read for the verification of a region. For example, the data stored at line 3 in Figure 2.12 is never read in the region R1 when R1 is restarted upon an error due to the absence of write-after-read (WAR) dependence; we refer to such a store as a WAR-free store. Thus, even if the data is corrupted due to an error and written to cache, R1's re-execution can correctly recover from the error. With that in mind, Turnpike releases such a WAR-free store—without verification—immediately after its commit, thereby relieving the store buffer pressure.

One might suspect that due to the fast release of unverified data to cache, the next region might read it making the error recovery fail, e.g., data stored at line 3 by a region R1 is loaded at line 4 by the following region R2 in Figure 2.12. Fortunately, it turns out that this is not a problem at all. For the unverified yet corrupted data to be read by the region R2's load, the error must be detected before R1's verification point, i.e., within WCDL (e.g., 10) cycles after the prior region R1 is finished. However, it is not R2 but R1 that the original region-level soft error verification (Section 2.1) restarts to recover from the error (❶ in Figure 2.12). Again, R1 does not read the data, i.e., no WAR dependence, and therefore restarting R1 can correct the error with no harm. On the other hand, if the error is detected after R1's verification point, then R2 is restarted for recovery (❷ in Figure 2.12). In this case, R2's load is guaranteed to read the correct data—because it was written by the region R1 which has already been verified.

The takeaway is that WAR-free stores can bypass the verification and thus can be immediately merged to cache after their commit, whether the error is detected during the execution of their region or within WCDL cycles after the region is finished. To realize this, Turnpike proposes a novel microarchitectural technique called committed load queue (CLQ)—shown in Figure 4.3—to dynamically check the absence of WAR dependence for each regular store.

Ideal CLQ Design with Address Matching: For each committed load, Turnpike allocates an entry in the CLQ to keep the address of the load. When the in-order pipeline tries to commit a regular store, Turnpike compares its address to all the entries of CLQ to check whether the store has WAR dependence on any prior load in the current region. If there is no address conflict, i.e., no WAR dependence, Turnpike releases the WAR-free store immediately instead of holding it in the store buffer. Otherwise, it is quarantined in the store buffer as is for the original region-level verification.

Once each region gets verified, Turnpike clears only the CLQ entries that were populated during the execution of the region. In particular, if the CLQ is full, Turnpike does not stall the pipeline. Whenever CLQ overflows, it instead disables the fast release logic for WAR-free stores, i.e., load address insertions to CLQ are blocked and it is wiped out, making the following stores go through the SB quarantine as is. When a new region starts thereafter,

Turnpike resumes the CLQ insertion so that the region can leverage the fast release of its WAR-free stores unless the CLQ overflows. More precisely, to ensure in-order store release to L1 cache, Turnpike does not enable the fast release logic until the prior region is verified with its stored released. Figure 2.13 illustrates how Turnpike selectively controls (enables/disables) the fast release of WAR-free stores according to the CLQ status, i.e., whether it is full or not.

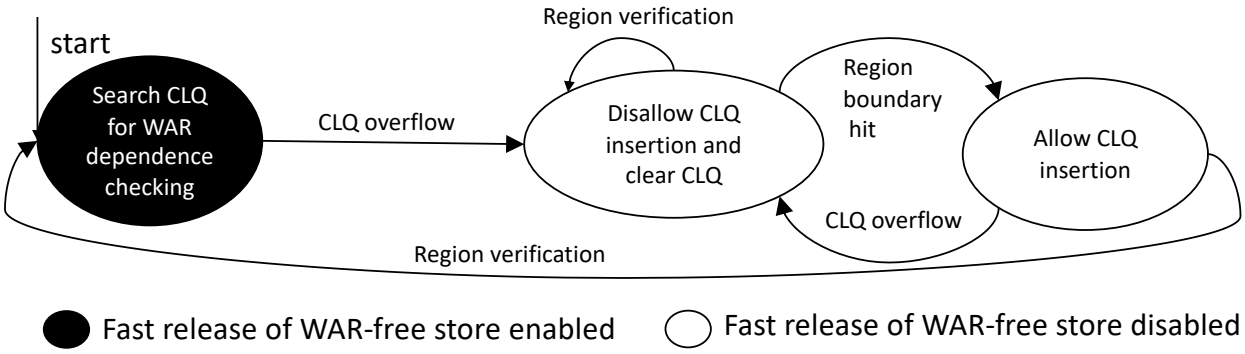


Figure 2.13. Selective control for WAR-free stores' fast release

Compact CLQ Design with Range Checking: In general, the bigger CLQ size is, the more often the fast release logic is enabled—leading to more WAR-free stores that can be merged to cache without their region verification. However, we found out that the WAR dependence is rarely found in each region. Taking this into account, we propose a range-based address checking that can compress all the addresses of the loads executed in each region by keeping the range of the minimum and maximum addresses during the execution. In this way, Turnpike only needs to allocate a single CLQ entry for each region without hurting the precision significantly.

Furthermore, such a per-region range-based CLQ entry renders the WAR dependence checking logic faster and simpler, which would otherwise require CAM (content-addressed memory) search for multiple entries. That is, for each regular store of a given region, Turnpike (1) looks up the CLQ entry corresponding to the region and (2) checks if the store address falls into the address range of the entry. The upshot is that Turnpike can significantly reduce the hardware cost for CLQ with neither the significant loss of the precision to detect WAR-

free stores nor the visible performance degradation compared to the address matching based ideal CLQ.

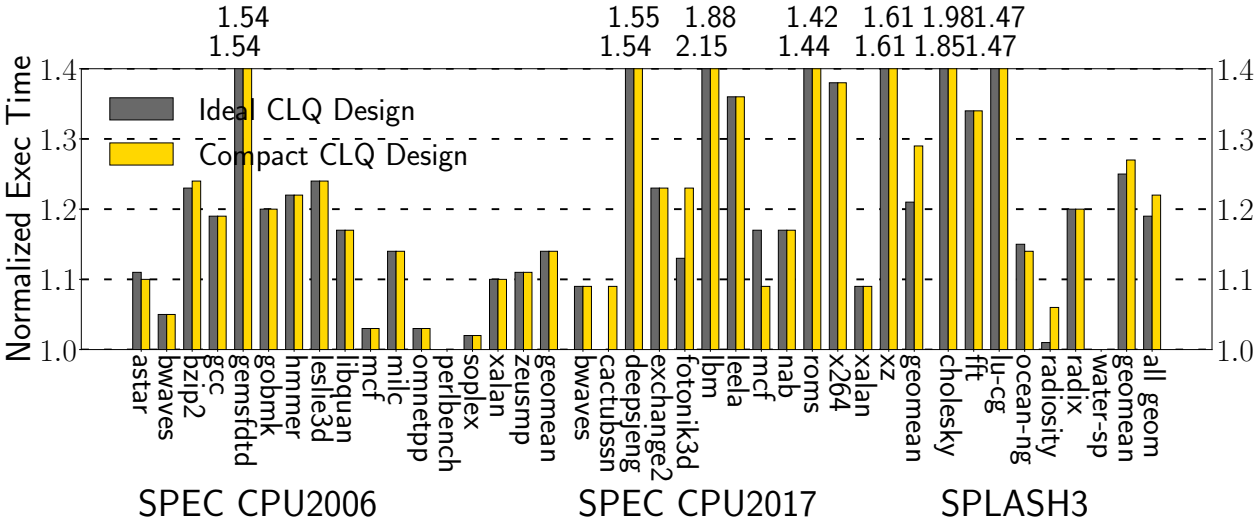


Figure 2.14. Run-time overhead compared to original program without resilience support between an ideal CLQ (infinite-size CLQ) and Turnpike's compact 2-entry CLQ; note that we only enable WAR-free checking and hardware coloring to exclude the impacts of Turnpike compiler optimizations

To confirm this, we compare the performance of Turnpike's compact CLQ against the ideal (100%-accurate) CLQ that performs address matching to identify WAR-free stores with an infinite number of CLQ entries. Figure 2.14 shows the performance overhead of the 2 designs which is normalized to the original application execution time that has no soft error resilience. It turns out that Turnpike's compact CLQ design only incurs 3% performance loss on average compared to the infinite-size ideal CLQ. As shown in Figure 2.15, that is because the infinite-size ideal CLQ leads to 10.58% higher detection accuracy than Turnpike's compact CLQ.

Finally, since Turnpike's compact CLQ has only 2 entries by default, it is technically possible to encounter the CLQ overflow, that is handled by the selective fast release control (Figure 2.13). For example, suppose that Turnpike executes three consecutive regions; while the first region is being verified, Turnpike can reach the end of the second region in which case CLQ does not have an available entry—due to the overflow—for accommodating the addresses of the last region's loads. In fact, Turnpike's compiler ensures that each region

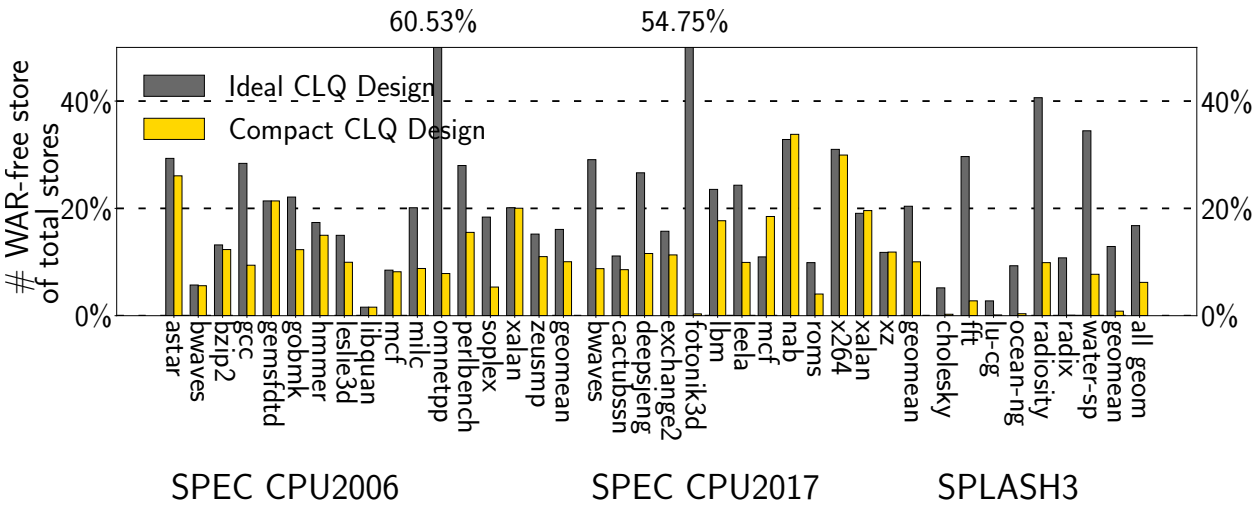


Figure 2.15. Ratio of detected WAR-free stores to all stores including checkpoints (higher is better) for the ideal basic CLQ (infinite-size) and Turnpike's compact CLQ (2 entries)

cannot have more than half of the SB size so that the verification of one region can be overlapped with the execution of the next region. However, since the region partitioning [40] is based on a path-insensitive analysis [135], some regions might have even a smaller number of stores than the half of the SB size, e.g., regions could have only one store when the SB has 4 entries as with ARM Cortex A53. As will be shown in Figure 2.23, the compact CLQ needs 3-4 entries to prevent the overflow for all our benchmarks.

Fast Release of Checkpoint Stores: By definition, all checkpoints are a WAR-free store in their region because the register stored by a checkpoint is never read by its own region; rather, the register is only used as an input to some later region. For this reason, one might think it is ok to release checkpoint stores without the SB quarantined for verification. However, we found it impossible because the error recovery could fail sometimes.

Figure 2.16 describes such a corner case with two regions R0 and R1 that both checkpoint the same register $r2$. Suppose $r2$ at line 2 is corrupted due to a soft error, and the error is detected after the verification point of the prior region R0. That is, the next region R1 is to be re-executed for the error recovery. Here, if a checkpoint of $r2$ at line 2c is merged to cache without verification—though it is corrupted, the checkpoint memory location is going to be

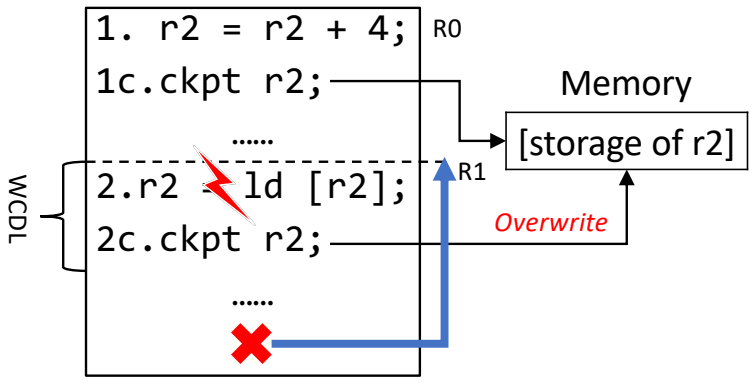


Figure 2.16. Problem of releasing of checkpoints without verification

overwritten by the corrupted value of $r2$. Hence, the re-execution of R1 ends up restoring its input register $r2$ from the corrupted value, thereby failing to correct the error.

The crux of the problem is that the checkpoint storage (location) is overwritten. With that in mind, we prevent the overwriting with alternative storage. This allows Turnpike to safely release even checkpoint stores immediately after their commit bypassing the verification. To achieve this, Turnpike leverages simple microarchitectural support called hardware coloring that can dynamically assign a distinct memory location (i.e., color) to a checkpoint. As such, Turnpike prepares a coloring pool, i.e., a set of memory locations as checkpoint storages, to manage the available colors for each register.

Implementation Details of Hardware Coloring: To ensure that a distinct color is assigned to each checkpoint, Turnpike prepares a 4-color pool, i.e., there are 4 checkpoint memory locations for each register, and manages 3 register maps: Available_Colors (AC), Used_Colors (UC) and Verified_Colors (VC). For a given register, AC maps it to the next available color while UC to the color that is used (assigned) for each region; Turnpike maintains UC as a part of RBB entry as shown in Figure 4.3. Likewise, VC maps a given register to the verified color (the checkpoint storage)—from which Turnpike restores the input register of the region being restarted on recovery.

Initially, VC is empty since there is nothing verified. When the in-order pipeline encounters a checkpoint, Turnpike tries to assign a color to the checkpoint by referring to AC with

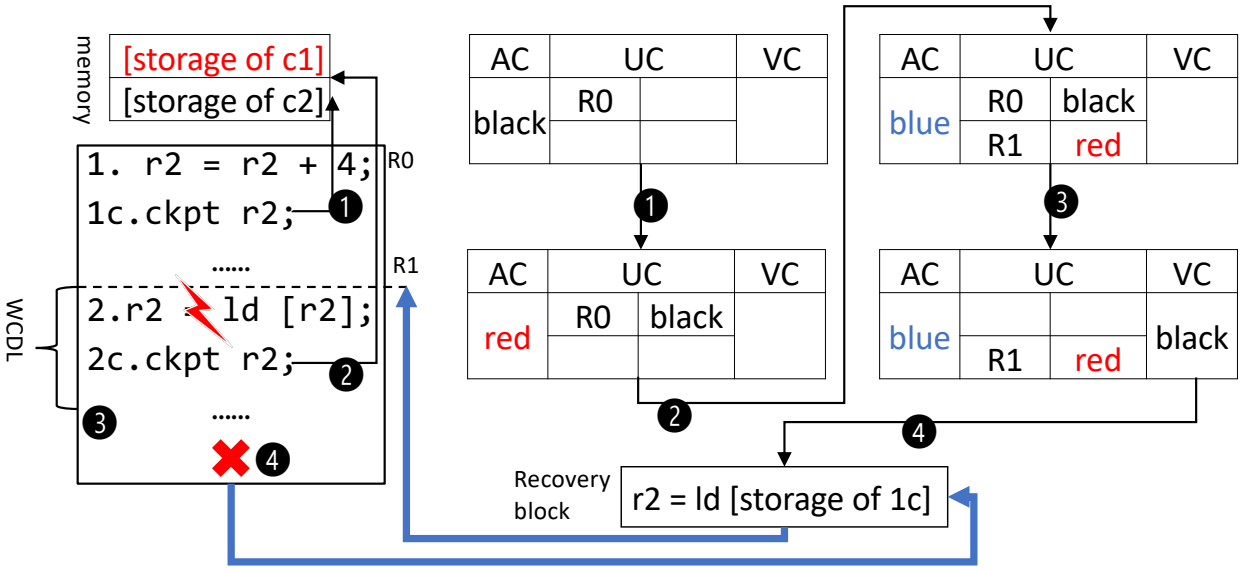


Figure 2.17. Fast release of a checkpoint store

the register being checkpointed. If there is an available color, it is inserted to the UC of the region in which the checkpoint store exists; otherwise, Turnpike simply gives up the fast release of the checkpoint store and falls back to the store buffer quarantine for verification.

Figure 2.17 shows how the status of AC, UC, and VC changes for register $r2$ being checkpointed. Here, checkpoint at line 1c is assigned **black** from AC, and thus the UC of a region R0 is updated with **black** (❶ in the figure). Similarly another checkpoint 2c is assigned **red**, and the corresponding register mapping in the UC of region R1 is updated with **red** (❷).

Once a region is verified, Turnpike flushes every color of VC to AC for reclamation purpose and updates the VC with the used colors of the verified region which are obtained by searching the UC with the region as a key. As shown in Figure 2.17, once R0 is verified at the end of WCDL after it is finished, Turnpike updates the VC with the color(s) that UC holds for R0, i.e., **black** (❸).

When an error is detected in a region R1 at some point after WCDL (❹) in the figure, Turnpike invokes the recovery block to restore the value of the input register $r2$ from the

black checkpoint storage, and then jumps back to the recovery PC, i.e., the entry of R1. Overall, Turnpike's hardware cost is not significant as will be shown in Section 2.4.4.

2.3 Discussion

Fault Model: We assume that both SB and RBB are hardened to be robust against soft errors as in prior work [40]. Besides, the 2 entries of the committed load queue (CLQ) and the three color maps (total 6 bits per register) are to be protected. Like prior work and commodity RAS (reliability/availability/serviceability) processors, caches and the address generation unit (AGU) should be hardened. Finally, a single parity bit is necessary for each register in case it holds store's address operand whose corruption ends up altering random memory location under Turnpike's fast release. Turnpike prevents this problem by causing any parity-detected error upon each register access to trigger its recovery process—as if it were detected by acoustic sensors.

Store Buffer Scaling: In general, it is challenging to enlarge a store buffer because SB's store-to-load forwarding impacts the length of a pipeline clock tick. SB must provide data within L1-hit time to avoid complications of scheduling loads with variable latency, e.g., for a 16/32-entries SB in Alpha AXP processor clocked at 3GHz, the store-to-load forwarding latency increases to 3-4 cycles[138]. Especially for in-order cores, it is even more challenging due to the power-hungry nature of the CAM (content-addressed memory) search for the store-to-load forwarding. That is why commodity in-order cores have only a few SB entries, e.g., ARM Cortex-A53 has 4 entries. In addition, Section 2.4.4 discusses the area and energy overheads of a large SB design in detail.

2.4 Evaluation

We implemented our optimizations presented in Section 2.2 with LLVM compiler [139]. To evaluate Turnpike's performance, we used SPEC2006[140]/SPEC2017[141] and SPLASH3[142] compiling all benchmarks with -O3. We conducted simulation using gem5 [143] which is configured with 2-issue, 2.5GHz dual-core processor with 32KB/64KB 2-way set-associative L1 instruction/data caches (2 cycles hit) and a unified 128KB 16-way set-associative L2 cache

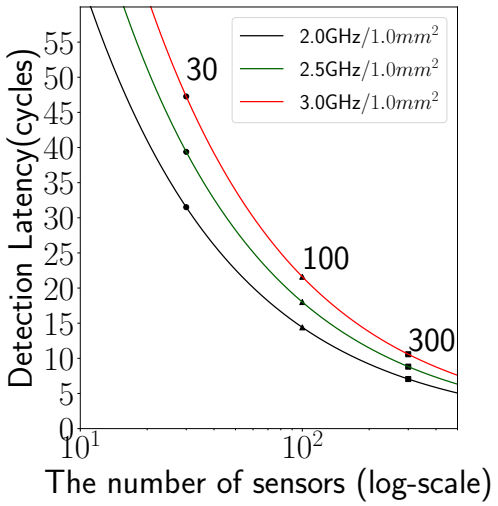


Figure 2.18. Detection latency across the number of deployed sensors

(20 cycles hit) to model an ARM Cortex-A53 processor [144]. The store buffer size is set to 4 as with the recent work that simulates the Cortex-A53 core [145], and the default CLQ size is 2. According to prior works [80–83], 300-30 deployed acoustic sensors can achieve 10-30 cycles of the worst-case detection latency (WCDL) with the area cost of less than 1% of die size, and therefore we set the default WCDL to 10 cycles.

For SPEC CPU2006 and SPEC CPU2017, we synchronized the number of simulated instructions by measuring the number of the function call instructions which is a constant across binary versions generated by different compiler optimizations. All benchmarks were fast-forwarded through the number of function calls to execute at least 5 billion instructions on the original executable without soft error resilience support, then we simulated the next 1 billion instructions with the gem5 in-order pipelined processor model. To be more practical, we simulated all SPEC CPU benchmarks with the reference inputs. For SPLASH3 benchmarks, we simulated the entire program with full system model of gem5. In the following, all performance results are presented as a slowdown, i.e., the inverse of a speedup, to the baseline that has no soft error resilience support.

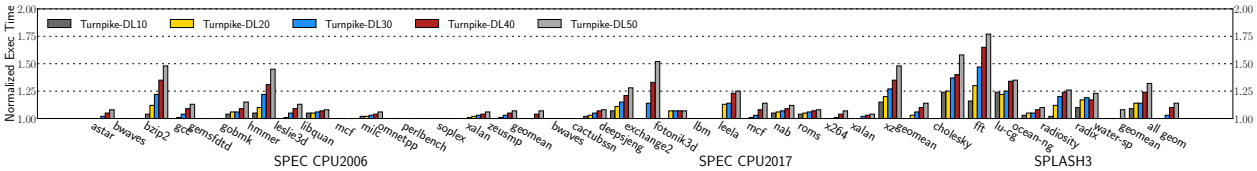


Figure 2.19. Performance overhead of Turnpike with varying WCDL from 10 to 50 cycles

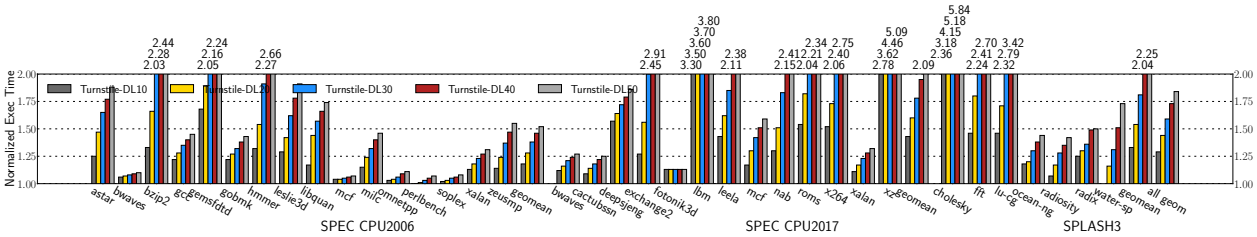


Figure 2.20. Performance overhead of Turnstile with varying WCDL from 10 to 50 cycles

2.4.1 Run-time Overhead with Varying WCDL

WCDL (worst-case detection latency) is inversely proportional to the number of sensors deployed and affected by the underlying clock frequency; the higher frequency the clock is, the longer the WCDL is. Figure 2.18 shows these trends for 300-30 sensors deployed on top of $1mm^2$ core die, e.g., 10 cycles WCDL for 2.5GHz core with 300 sensors. Due to the process technology and the fabrication issue, deploying all 300 sensors might not be possible under the budget of 1% die size overhead. Thus, we vary WCDL from 10 cycles up to 50 cycles to cover other possible fabrication cases and evaluate the general trend of Turnpike's overhead across the different WCDLs.

Figure 2.19 presents the run-time overhead of Turnpike for 5 WCDLs: 10/20/30/40/50. Turnpike incurs only 0-14% overheads on average for the varying WCDLs from 10 to 50 cycles. In contrast, Turnstile suffers 29-84% average overheads for the 5 WCDLs (see Figure 2.20). It is worth noting that Turnpike significantly outperforms Turnstile for all the benchmarks. In particular, when 10-cycle WCDL is used by default, Turnpike's overhead is only around 1% for most of the benchmarks, thereby delivering 0% average overhead!

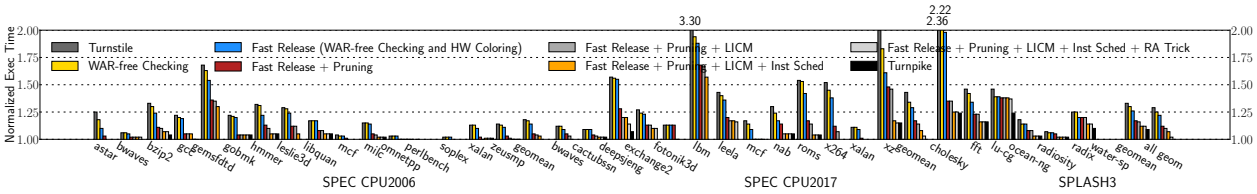


Figure 2.21. Performance comparison between Turnstile and Turnpike's optimizations with 10-cycle WCDL.

2.4.2 Impact of Turnpike's Optimizations

This section presents the performance impact of Turnpike's optimizations. Figure 2.21 shows the performance results of the following 8 cases for the default 10-cycle WCDL.

Turnstile: is the state-of-the-art work, that does not use our optimizations, and incurs a 29% overhead on average.

WAR-free Checking: uses the fast release of regular stores; it reduces the overhead to 25%.

Fast Release (WAR-free checking and HW coloring): enables the fast release of both regular stores and checkpoint stores; this reduces the overhead to 22%.

Fast Release + Pruning: is the combination of the above fast release and checkpoint pruning, achieving 12% overhead; the latter removes many unnecessary checkpoints (Figure 2.22).

Fast release + Pruning + LICM: is the combination of the fast release, checkpoint pruning, and LICM, achieving 10% overhead. LICM particularly works well for `deepsjeng`, `fotonik3d`, `nab`, and `x264`, reducing their overhead by >5%.

Fast release + Pruning + LICM + Inst Sched: is the combination of the fast release, checkpoint pruning, LICM, and instruction scheduling. The resulting average overhead is 7%.

Fast release + Pruning + LICM + Inst Sched + RA Trick: is the combination of the fast release, checkpoint pruning, LICM, instruction scheduling, and store-aware register allocation. On average, it reduces the overhead to 2%. Significant overhead reduction is

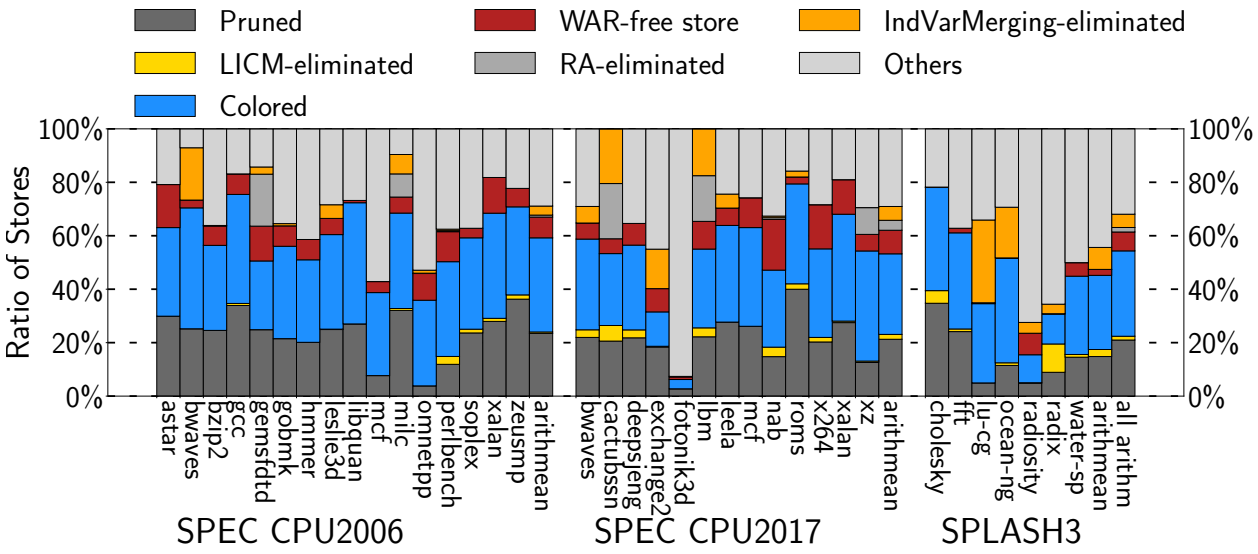


Figure 2.22. Store breakdown; 2-entry CLQ; 10-cycle WCDL

found in `gemsfdtd` and `lbm`; as shown by Figure 2.22, the register allocation trick eliminates the stores of the 2 benchmarks by 19% and 17%, respectively.

Turnpike: uses all above optimizations along with loop induction variable merging, eliminating Turnstile's overhead completely, i.e., Turnpike's average overhead is 0%! It turns out that loop induction variable merging is particularly effective for `exchange2`, `leela`, `lu-contigous`, and `radix`.

2.4.3 Impact of SB Pressure Reduction Schemes

Figure 2.22 shows the detailed breakdown of all stores with 8 categories: **Pruned** corresponds to the checkpoint stores eliminated by the optimal checkpoint pruning while **LICM-eliminated** to those removed by the loop-invariant code motion. Among the remaining checkpoint stores, **Colored** corresponds to those that can be merged to cache without the SB quarantine. Similarly, **WAR-free** corresponds to the regular stores that can be merged to cache without verification. Next, **RA-eliminated** and **IndVarMerging-eliminated** correspond to those stores that can be removed by our store-aware register allocation and loop induction variable merging optimization respectively. Finally, **Others** represents the

rest of the stores which cannot be removed or fast released by Turnpike thus going through the verification. As shown in the figure, the checkpoint pruning removes 21% of all stores while LICM removes 1.4% of them on average. Although LICM has little impact for the majority of the benchmarks, its checkpoint removal is significant for `cactubssn`, `lbm`, `cholesky` and `radix`. Meanwhile, 1.7% and 5% of all stores are removed by store-aware register allocation and loop induction variable merging respectively. Finally, 39% of all stores can be released to cache without going through the SB quarantine, highlighting the effectiveness of Turnpike's fast release.

2.4.4 Hardware Cost Analysis

Table 2.1. Cost comparison of Turnpike and a large SB design

	Area (μm^2)	Dynamic access (pJ)
4-entry SB (CAM)	621.28	0.43099
Color maps in Turnpike (RAM)	36.651	0.02518
2-entry CLQ in Turnpike (RAM)	24.434	0.01679
Turnpike in total (color maps + 2-entry CLQ)	61.085	0.04197
40-entry SB (CAM)	3132.50	2.11525
Turnpike in total / 4-entry SB	9.8%	9.7%
40-entry SB / 4-entry SB	504%	497%

Turnpike incurs a very small hardware overhead; the 2-entry CLQ requires 16 bytes while the three 4-color maps (AC, UC, and VC in Section 2.2.3) need 6 bits ($3 \cdot \log_2 4$) per register—requiring 24 bytes for 32 registers as in ARM Cortex A53. In summary, Turnpike only needs total 40 bytes for such an in-order processor.

To further evaluate the area and the power overheads of Turnpike, we used CACTI [146] with $22nm$ technology. Table 2.1 highlights the area/power-efficiency of Turnpike's compiler/architecture co-design. Compared to ARM Cortex A53's 4-entry store buffer as a baseline, Turnpike only incurs 9.8% area and 9.7% energy overheads (see the second last low of the table). In contrast, simply increasing the store buffer size to 40 causes 504%/497% area/energy overheads, which is unrealistic for low-power in-order cores.

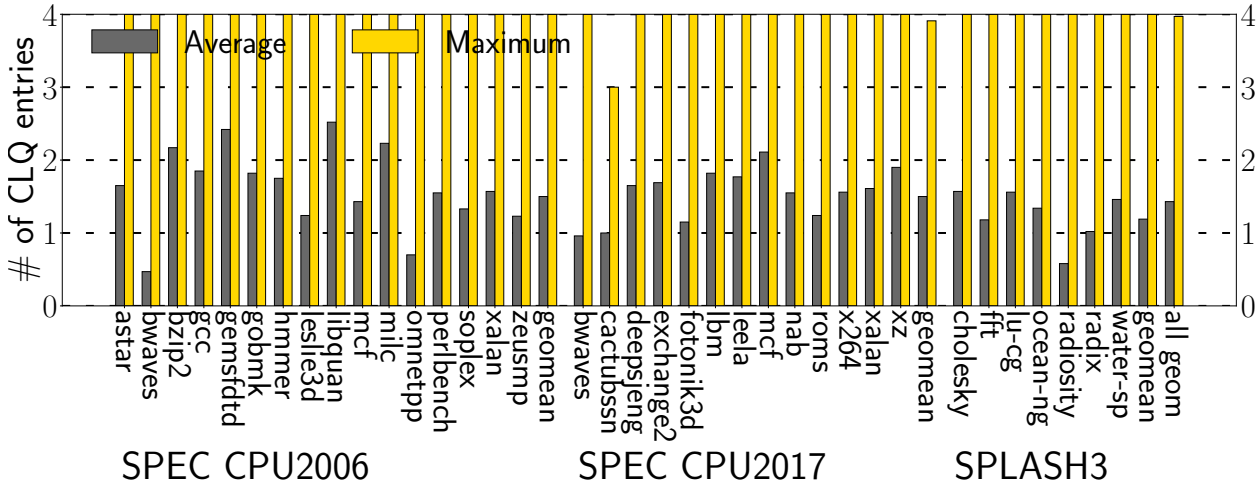


Figure 2.23. Dynamic CLQ entries populated; 10-cycle WCDL

2.4.5 Sensitivity Analysis

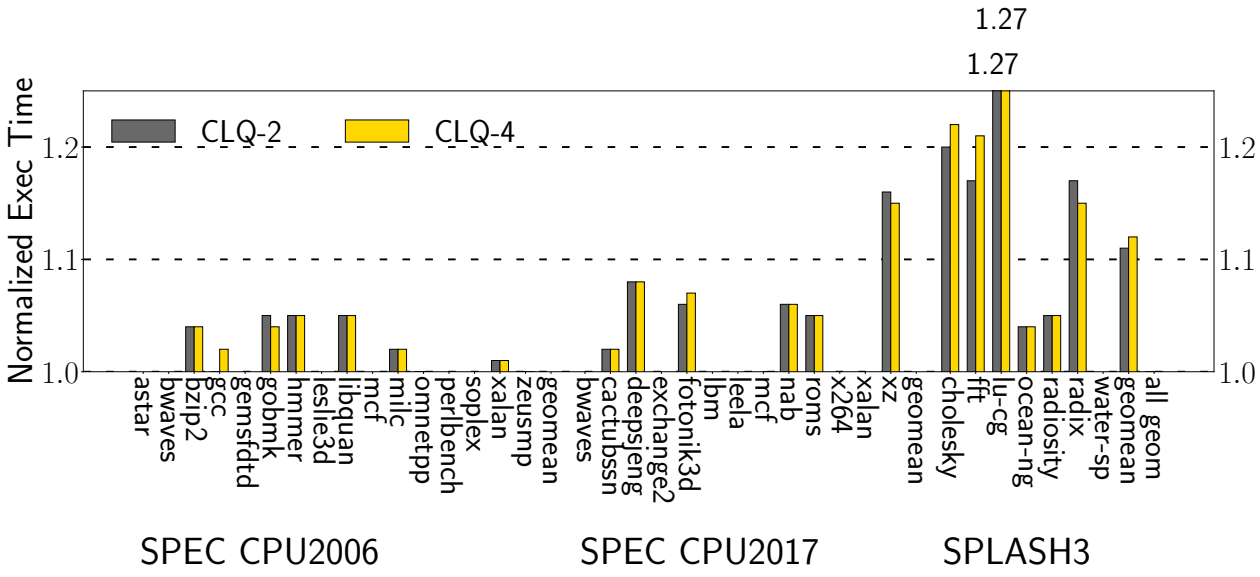


Figure 2.24. 2-entry vs 4-entry CLQs with 10-cycle WCDL

Sensitivity to CLQ size: Recall that CLQ is a critical hardware structure, since its dependence checking logic is essential for the fast release of WAR-free stores. Figure 2.23 shows the average and maximum numbers of dynamic CLQ entries populated at run time. The average number of populated CLQ entries is about 1 though the maximum number goes

up to 3 or 4 for some applications. Further investigation confirms that the peak number is scarcely observed. That is why Turnpike's CLQ size is set to 2 (by default), and its performance is almost the same as that of a bigger CLQ with 4 entries as shown in Figure 2.24. The takeaway is that our compact CLQ design is not only low-cost but also high-performance.

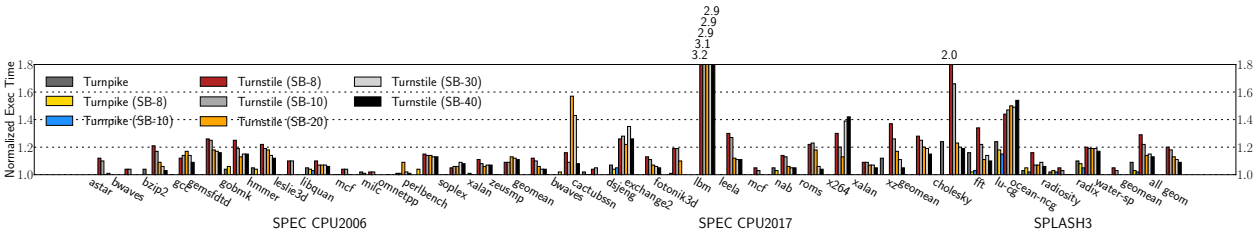


Figure 2.25. Performance comparison of Turnpike and Turnstile with different SB sizes (8, 10, 20, 30, 40) using 10-cycle WCDL

Sensitivity to SB Size: It is hard to increase the size of a store buffer (SB) especially for in-order cores. Nonetheless, to highlight the performance of Turnpike, we enlarge the store buffer of Turnstile—though it performs poorly for the 4-entry SB of ARM Cortex A53 which is Turnpike's SB size. In addition to the default size, we tested 5 more SB sizes from 8 to 40 with 10-cycle WCDL. As shown in Figure 2.25, for 5 SB sizes (8/10/20/30/40) with 10-cycle WCDL, Turnstile's average performance overheads are 20%, 18%, 13%, 11%, and 9%, respectively. Note that although Turnstile is equipped with a much larger SB, it performs significantly worse than Turnpike. Even with the 40-entry SB that is 10x bigger than Turnpike's SB, the average slowdown of Turnstile is 9% whereas that of Turnpike is 0% (see Figure 2.21). We also tested Turnpike for bigger SB sizes. Figure 2.25 shows that the average overhead of Turnpike is still 0% with the SB sizes of 8 and 10 and decreases as the SB size increases.

2.4.6 Region Size and Code Size Analysis

Figure 2.26 shows dynamic region size and binary code size increase. On average, there are 11.2 instructions per region, and code size increases by 0.4% compared to the baseline. Overall, long regions lead to less code size increase; `bwaves` has 35 instructions per region

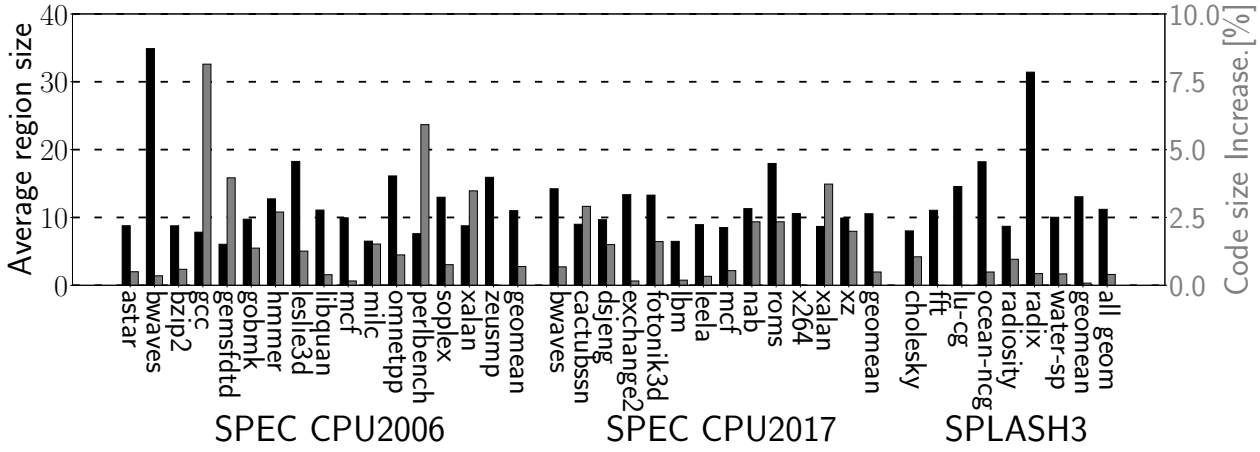


Figure 2.26. Region size (left) and binary overhead (right bar)

leading to 0.35% increase, while `gcc` increases the size by 8.15% due to many small regions (7.8 instructions per region).

2.5 Other Related Work

Many prior works use redundant-computation-based detection for high error coverage. Instruction-level duplication replicates instructions and detects errors by comparing the results of the original and replica instructions [28–30, 32, 147–149]. In contrast, redundant multithreading simultaneously runs a redundant thread with the original thread on available cores. Some schemes use SW techniques to realize the redundant multithreading without hardware modification [31, 34, 35, 150, 151], while others exploit HW support to reduce the performance overhead [37, 150, 152–155], Another schemes with process-level redundancy duplicate the application process to compare the outputs [156–158] between the parent and child processes. Finally, non-duplication schemes detects errors by catching abnormal symptoms caused by a soft error [38, 159–161].

To recover from detected errors, triple module redundancy (TMR) adopts a majority voting between the 3 executions, increasing the hardware cost. The most common error recovery scheme is to use checkpointing or logging program status (register and memory). Prior work on coarse-grained recovery requires expensive hardware support for incrementally

checkpointing memory status [82] or equipping the core with a large store buffer for memory logging [162]. To reduce the checkpointing cost in a fine-grained manner, recent studies partition the program into small idempotent regions to reduce the number of data to be checkpointed [57, 115, 116, 133]. However, the idempotent recovery schemes still incur a significant run-time overhead due to register spilling or checkpointing.

All prior works either impose a large hardware cost or suffer a high run-time cost. To the best of our knowledge, Turnpike is the first, that reduces both costs effectively for in-order cores, requiring little hardware cost though it achieves almost 0% run-time overhead.

2.6 Summary

This chapter presents Turnpike that achieves lightweight soft error resilience for in-order cores with acoustic-sensor-based detection. Using compiler optimizations and simple hardware support, Turnpike incurs near-zero performance overhead.

3. REPLAYCACHE: ENABLING VOLATILE CACHES FOR ENERGY HARVESTING SYSTEMS

Energy harvesting systems [87] have been deployed in a wide range of application domains, such as Internet of Things (IoT) devices [163–166], wearables [167–171], stream and river surveillance [88, 89], health and wellness monitors [90–93], etc. Energy harvesting systems are well-suited to these domains with the superb property of ultra-long operation time without maintenance by collecting energy from variant ambient sources such as solar, thermal, piezoelectric, and radio-frequency radiation.

However, due to the batteryless nature, energy harvesting systems suffer unpredictable frequent power failure and thus require some form of crash consistency which must be lightweight; otherwise checkpointing/restoring consistent program states across the failure can limit forward progress by consuming hard-won energy. Thus, existing systems [59, 94–98] have been designed with byte-addressable non-volatile memory (NVM), where data are immediately persisted and thus recoverable at the cost of long latency. While volatile write-back caches can hide the store latency and improve performance with a load hit exploiting data locality, they have been assumed to be not viable or at least challenging in energy harvesting systems.

The crux of the problem is that volatile write-back cache states are not preserved across a power outage. This may lead to an inconsistent NVM state, and therefore the power-interrupted program may fail to resume correctly. That is why existing energy harvesting systems do not use volatile data caches; prior work [95] uses a read-only NVM-based instruction cache where a crash consistency (without stores) is not an issue. Unfortunately, it is a challenging problem to ensure correct data cache persistence in a lightweight manner to maintain forward progress. For example, software logging causes serious performance degradation (100-300% slowdown) since each regular store is preceded by the log store, cacheline flush, and store fence [54–58].

One possible hardware solution is to use a volatile write-through cache. It allows energy harvesting systems to benefit from load hits and to ensure crash consistency by enforcing that the completion of a store instruction guarantees the persistence of the data in NVM. However,

write-through cache comes with a performance penalty on each store as conventional cache-free energy harvesting processors. Since they use a simple in-order core without any form of speculation, they cannot hide the data persistence latency.

Alternatively, one can design a persistent write-back data cache, e.g., non-volatile cache (NVCache) [172–178] and non-volatile SRAM cache (NVSRAMCache) [179–185]. However, both cache designs have their own problems. Due to the NVM-based design, NVCaches incur high latency and power consumption for each access. NVSRAMCaches embed NVM to backup an SRAM-based cache, and checkpoint/restore the entire SRAM to/from the NVM backup across power failure, leading to consume high energy. While NVSRAMCaches may be as fast as a volatile SRAM cache without power failure, it is hard to maintain the performance with frequent failure—i.e., the norm of energy harvesting—unless they use a lower-power yet fast non-volatile technology which has not been commercialized yet.

With that in mind, we propose ReplayCache, a *software-only scheme* that enables commodity energy harvesting systems to exploit a volatile write-back data cache for performance, yet ensures lightweight crash consistency of the NVM state for correctness. ReplayCache does not ensure the persistence of dirty cachelines or record their logs at run time: i.e., no write amplification. Instead, ReplayCache *re-executes the potentially unpersisted stores* in the wake of power failure to restore the consistent NVM state from which interrupted program can safely resume.

To support the store replay, ReplayCache partitions program into a series of regions so that the operand registers of store instructions are intact (i.e., not overwritten by the other following instructions) in each region. We refer to this process *store-register-preserving region formation*. Then, at run time, ReplayCache checkpoints all registers just before power failure to secure the store operand registers. We note that the just-in-time register checkpointing is already available in energy harvesting systems: e.g., QuickRecall [94], Hibunus [96], and NVP [95]. During recovery, these checkpointed registers are used to *re-execute the stores* along the same program path as the one before a power failure; for the store replay, a recovery code block is generated for each region, i.e., ReplayCache directs program control to the recovery code in the wake of the power failure. After that, ReplayCache can safely resume from

the interrupted program point with the checkpointed registers and the recovered consistent NVM.

Experiements with 23 applications from Mibench [186] and Mediabench [187] benchmarks show that compared to the baseline with no caches, ReplayCache can make them 10.72x and 8.5x-8.9x faster (on geometric mean) for the scenarios without and with power outages, respectively. ReplayCache makes the following contribution:

- ReplayCache is the first to enable volatile caches for commodity energy harvesting systems; its software-only design allows them to use traditional SRAM cache as is with crash consistency guarantee.
- ReplayCache proposes a new resumption scheme that recovers consistent NVM states across power failure by re-executing potentially unpersisted stores before the failure during the recovery, without write amplification.
- ReplayCache achieves the high performance despite its software-only design; its performance is comparable to an ideal NVSRAMCache for realistic power failure traces.

3.1 Background and Challenges

This section discusses the architectures of existing energy harvesting systems (§3.1.1), the potential crash consistency problem of using a volatile write-back data cache as is (§3.1.2) and the limitations of existing cache solutions (§3.1.3).

3.1.1 Architecture of Energy Harvesting Systems

Energy harvesting systems derive energy from external sources (e.g., solar power, thermal energy, ambient electromagnetic radiation) and mostly store it in a tiny capacitor for small IoT devices such as wearables. Due to the nature of unreliable power supply, energy harvesting systems should be able to save (checkpoint) the current state upon power failure, and restore the program state and seamlessly resume the execution when the power comes backs as if nothing had happened. A power interruption in energy harvesting systems is a frequent, normal event, unlike in high performance computing context. It is thus crucial

to design systems for whole system persistence (WSP) [110, 188] so that they efficiently save/restore the program state and make a progress no matter where power failure happens.

The above requirements motivate existing energy harvesting systems to adopt NVM as main memory. However, the registers in a processor still remain volatile for performance reasons. Broadly speaking, existing mechanisms to checkpoint/restore registers can be classified into two groups.

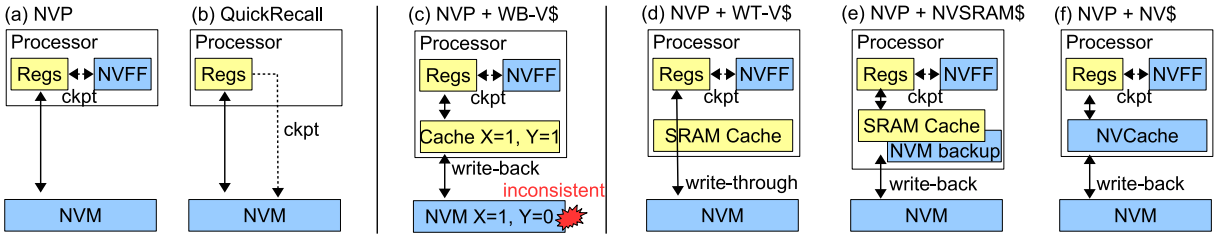


Figure 3.1. The architectures of existing energy harvesting systems

Figure 3.1(a) shows the architecture of Non-Volatile Processor (NVP) [189], representing the first group that checkpoints and restores registers in place with some additional hardware support [98, 189, 190]. NVP is equipped with an energy harvester, a voltage monitor, and capacitors (not shown). When the monitor detects impending power failure, i.e., the voltage is about to drop below a certain threshold, it signals the processor to checkpoint all the registers (so-called just-in-time checkpointing) into their neighboring non-volatile flip-flops (NVFF) [191–194]. When power is secured enough across the failure, the processor restores the register states from the NVFF and resumes the execution from the power-interruption point. As both register and memory states on the resumption point are guaranteed to be the same as the states before a power failure, there is no crash consistency problem. A downside of NVP is the use of additional (expensive) hardware NVFF.

Figure 3.1(b) illustrate the architecture of QuickRecall [94], representing the second group that checkpoints/restores the registers to/from the NVM. Similar to NVP (and others), QuickRecall also implements just-in-time (JIT) register checkpointing with a voltage monitor and a capacitor (not shown). When the monitor detects upcoming power failure, it triggers an interrupt whose handler checkpoints all the registers into the NVM. When the power comes back, the recovery runtime reads the checkpointed states from the NVM in order to

restore the registers. As in NVP, QuickRecall (and others [96, 195] in this group) has no crash consistency issue. A drawback of QuickRecall is that it should secure a lot more energy than NVP to atomically checkpoint all registers in NVM before impending power failure.

3.1.2 Crash Inconsistency of Write-back Caches

Adding a cache to energy harvesting systems has a high potential to improve their performance (with load hits) and allow them to make more progress for a given energy harvested. However, a naive integration of volatile write-back data cache with existing energy harvesting systems (e.g., NVP, QuickRecall) for performance, may lead to a crash consistency problem, as depicted in Figure 3.1(c).

Suppose the NVM has the memory state $X = 0$ and $Y = 0$ initially. And suppose a program has a power outage after executing two stores $W(X) = 1$ and $W(Y) = 1$. Before the outage, the cache had the updated state $X = 1$ and $Y = 1$, but the NVM may not, depending on whether the cache lines holding X and/or Y are evicted or not, which is varying according to cache replacement policy and thus unpredictable. Since the volatile cache state disappears upon a power loss, i.e., any unpersisted dirty cacheline is completely lost, the system may restart from an inconsistent state (e.g., $X = 1$ and $Y = 0$) failing to resume or producing wrong output later.

3.1.3 Limitations of Prior Work

There are four possible solutions to address the crash consistency problem. The first approach is to use a write-through cache. Figure 3.1(d) illustrates a case in which NVP is configured with a volatile write-through cache (a traditional SRAM-based one). The write-through policy ensures data consistency as the completion of a store instruction ensures the data persistence to NVM. However, the downside is a long store latency (as in the case without a cache); more precisely, for a write miss, the critical path is lengthened due to the write-allocation policy. Since most of the energy harvesting systems are designed with a simple in-order processor, it is impossible hide the store latency.

As shown in Figure 3.1(e), the second approach is to equip the processor with the NVSRAMCache that embeds NVM (e.g., ReRAM) to traditional SRAM cache for its backup and restoration [196–199]. As with NVP, NVSRAMCache also relies on a voltage monitor for just-in-time checkpointing of the SRAM cache. When power is about to be cut, NVSRAMCache triggers a copy from SRAM to NVM for all the cachelines. Along with their restoration, the entire cache backup makes NVSRAMCache consume high energy across power failure. Moreover, NVSRAMCache significantly postpones the booting time due to the high amount of energy that must be secured for failure-atomic cache checkpointing. Although researchers attempt to improve the backup latency [180, 181], their NVSRAMCaches are more of a forward-looking technology in an ideal form—since none of current non-volatile materials can provide comparable latency to SRAM [59].

The third approach is NVCache [182, 200] that leverages a pure non-volatile technology as the cache material; see Figure 3.1(f). Since NVCache usually uses a slight faster NVM technology for the cache than the non-volatile main memory, the NVCache accesses are a lot slower—consuming more energy—than those of traditional SRAM cache. Thus, NVCache-equipped energy harvesting systems only occasionally outperform cache-free systems when there is very high locality. In sum, the second and third approaches—Figures 3.1(e) and (f)—are to make a cache itself persistent surviving power failure, but they suffer from their own problems.

Finally, data loggings are another approach to crash consistency in the presence of a volatile cache. However, they dramatically increase execution time (or power consumption if implemented in hardware), prohibiting their use in energy harvesting systems. For example, iDO [57] and Mnemosyne [58] incur 100-300% slowdown, prohibiting their use in an energy harvesting system. Furthermore, since they only supports crash consistency for a few transactions or failure-atomic sections, additional overheads should be paid for whole system persistence (WSP) [110, 188]. Similarly, existing WSP schemes for cache-free harvesting systems such as Alpaca [201] and Ratchet [202] also cause unacceptable slowdown (60% - 500%). Since they assume no cache, their overheads would be even worse for cache-enabled systems due to the additional cacheline flush and fence overhead.

3.2 Design

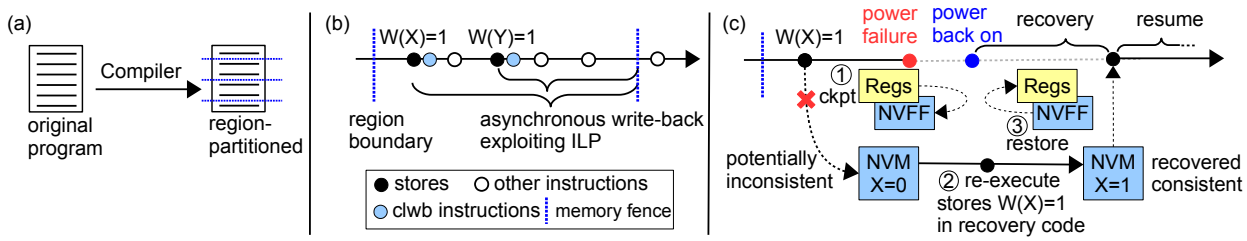


Figure 3.2. An overview of ReplayCache

The goal of ReplayCache is to guarantee crash consistency (i.e., an ability to restart from a consistent state) of energy harvesting systems in the presence of a volatile write-back data cache, allowing them to make the most of data locality and to achieve more progress given an energy budget. ReplayCache employs software-only design that provides (A) program region partitioning, (B) region-level persistence, (C) register checkpointing before a power outage, and (D) recovering a consistent NVM state.

3.2.1 Program Region Partitioning

As shown in Figure 3.2(a), ReplayCache compiler partitions entire program input to a series of regions. Each region ensures that the operand registers (e.g., base address, new value) of a store therein are not overwritten by any other succeeding instructions in that region.

3.2.2 Region-level Persistence

ReplayCache asynchronously writes back the stored value to the NVM, and overlaps the write-back operations with the executions of other following instructions, effectively exploiting instruction-level parallelism (ILP).

Unlike a traditional write-back cache, ReplayCache ensures that all the stores in a region are persisted (written back to the NVM) before the region ends; this paper calls this *region-level* persistence guarantee in which the persistence latency of in-region stores can be

naturally hidden by ILP; Figure 3.2(b) illustrates the window of potential ILP gain, and the unpersisted state of each store. This region-level persistence assures that at the moment of a power outage, all the stores in the preceding program regions have already been persisted, and only the stores in the interrupted region could not potentially be unpersisted.

The processor stalls if there exists an outstanding unpersisted store at the end of a region, until it becomes persisted to the NVM. ReplayCache compiler dedicates a single register (e.g., $r12$) to be acted as *region register* to track the most recent region boundary information for recovery. That is, the register is updated with a program counter at each region boundary.

3.2.3 Register Checkpointing

Across a power outage, ReplayCache saves register states just before the outage and restores them in the wake of the outage using the voltage monitor based JIT checkpointing mechanism (§3.1.1) in commodity energy harvesting systems. For instance, NVP and QuickRecall can both checkpoint register states before the power off and to restore them after the power on as discussed in §3.1.1. In Figure 3.2(c), step ① illustrates that ReplayCache checkpoints the registers when power is about to be cut off.

3.2.4 Power Failure Recovery

The recovery protocol works as follows. Upon a power outage, the interrupted region's stores before the outage may or may not be persisted, e.g., $W(X) = 1$ in Figure 3.2(c) unpersisted till the outage—while all preceding regions' stores are guaranteed to be persisted and thus consistent (due to the region-level persistence). In the wake of the outage, ReplayCache jumps to the recovery code block of the interrupted region to replay all the stores left behind the outage. The recovery code block re-executes such unpersisted stores using the checkpointed register values in either NVFF (NVP) or NVM (QuickRecall). This is shown as a step ② of Figure 3.2(c). Finally, the recovery code sets off a restoration signal to restore all registers (including PC) from NVFF or NVM, and then resumes the program from the outage point with the restored register and the recovered NVM states as in step ③

of Figure 3.2(c). In this way, ReplayCache allows energy harvesting systems to seamlessly leverage a data cache without amplifying NVM stores.

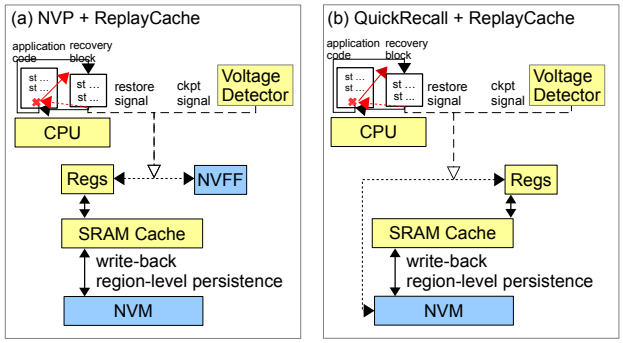


Figure 3.3. The ReplayCache architecture based on a volatile write-back cache. ReplayCache can be combined with existing energy harvesting systems such as (a) NVP and (b) QuickRecall. Voltage detector is available in every energy harvesting system

Figure 3.3 depicts how ReplayCache works for existing energy harvesting systems, i.e., NVP and QuickRecall, using the aforementioned recovery protocol. The takeaway is that ReplayCache enables the commodity systems to leverage write-back volatile data caches as is with help of the region-level persistence and the recovery code based recovery. The details of recovery code block generation is presented in Section 3.4.

3.3 Implementation

This section describes how ReplayCache compiler realizes the *store-register-preserving region formation*. The role of the compiler is three-fold: (1) region formation (2) CLWB insertion after each store, and (3) recovery code generation whose discussion is deferred to Section 3.4.

For region formation, the compiler partitions program into a series of small regions so that in each region, no operand registers of a store instruction are overwritten by the other following instructions. In this way, store registers remain intact from the execution of their region all the way to the power failure recovery time on which ReplayCache replays the same

stores in case they were not persisted before the failure. We refer to this property as *store integrity*.

Figure 3.4 shows a high-level workflow of ReplayCache compiler which introduces 3 additional phases (shaded in the figure) to the standard backend compilation passes. This region formation is performed in a whole-program manner to cover the entire program stores, i.e., every single program point belongs to one of the regions.

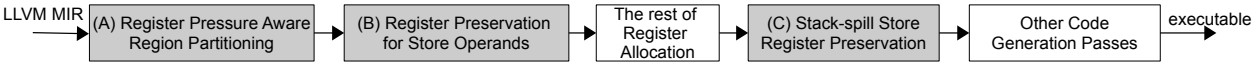


Figure 3.4. The workflow of ReplayCache compiler

At first glance, forming regions appears to be as simple as counting the store registers while traversing the control flow graph (CFG) and placing boundaries before the count exceeds the number of (physical) registers in the processor (e.g., 16 for NVP and QuickRecall). However, it turns out that two problems below make the region formation challenging.

Problem 1. Circular Dependence: Intuitively, the store-register-preserving region formation can be realized with two phases: (1) region partitioning that counts stores to place a region boundary, i.e., store fence instruction, in program and then (2) register preservation that extends the live interval of store operands to the end of each region for their exclusive register use. Thus, the register preservation depends on the region partitioning. However, since the partitioning counts the stores to determine where to place a region boundary, it also depends on the register preservation—forming a circular dependence; the live interval extension of the register preservation increases the register pressure, i.e., the number of necessary registers. Due to the register file size limitation, some registers could be spilled (written) to stack through stores. We call them stack-spill stores.

Problem 2. Stack-Spill Stores: In addition to regular stores, ReplayCache also needs to ensure the integrity of stack-spill stores for correct failure recovery. However, it is hard for the region partitioning to figure out in advance what variables are to be spilled to stack. That is because stack-spill stores are determined in the later register allocation pass assigning physical registers. One might try to perform the region partitioning after the

register preservation to exactly count the number of stores. However, this is not a viable option since the region partitioning depends on the register preservation in the first place.

ReplayCache Approach to the Problems: To break the circular dependence between the region partitioning and the register preservation, ReplayCache first considers a function call boundary as initial regions and conducts (A) *register-pressure aware region partitioning* (the first box of Figure 3.4) to fine-cut the initial regions as needed. Our register-pressure tracking algorithm allows the region partitioning phase not only to estimate the number of stack-spill stores, breaking the dependence on the register preservation, but also possibly to form a region with no spill in a best-effort manner. In case register allocation actually generates stack-spill stores in the formed region after the (B) register preservation phase, ReplayCache runs a post-processing (C) *stack-store register preservation* phase (the fourth box of the figure) that runs through the register-allocated code to find those stack-spill stores whose registers are overwritten in their region, and places a region boundary before the register updates. The rest of this section details the three phases with referring to them with (A), (B), and (C), respectively.

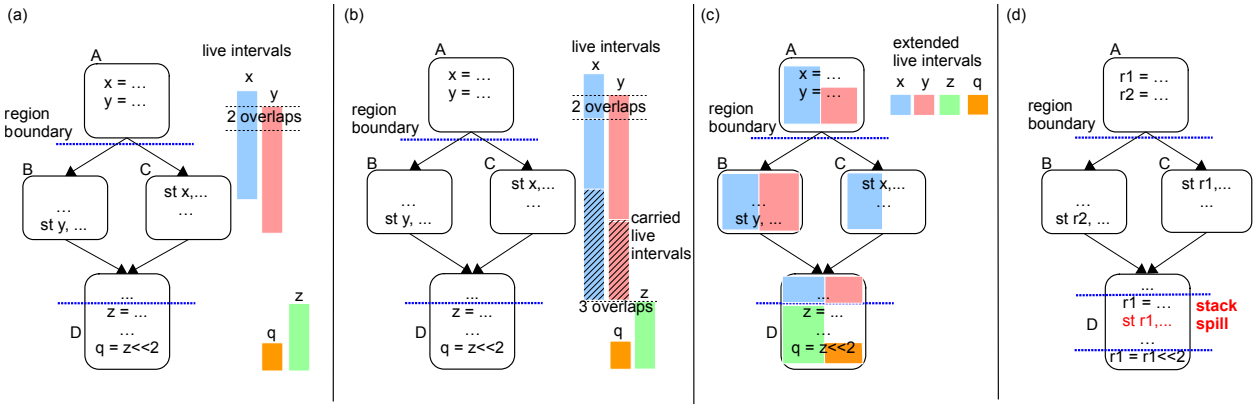


Figure 3.5. An example partitioned program with live intervals; (a) shows an initial region boundary at the function beginning (basic block A) and live intervals of variables x, y, and z; (b) shows the second boundary inserted in basic block D over which live intervals of x, y are carried, when the partitioning threshold (physical registers) is two; (c) shows extended intervals of all three variables towards the second region boundary; and (d) shows the case of redefining the register $r1$ of spill store in basic block D after variables are assigned to the physical register

3.3.1 Register Pressure Aware Partitioning

ReplayCache initially forms regions at function call boundaries and the end of conditional branches, and then runs the register-pressure aware region partitioning algorithm, which aims to achieve two goals. First, it attempts to maximize the length of a region to provide ReplayCache with long potential ILP window for its region-level persistence; see Figure 3.2 (b). Second, it tries to minimize stack spills generated by the later register allocation phase.

For this purpose, the partitioning algorithm keeps track of the register pressure by traversing the control flow graph (CFG) of each initial region. ReplayCache counts the number of overlapping live intervals at each program point visited during the CFG traversal. In particular, if store instructions are encountered, ReplayCache carries their live intervals along the way beyond the original live intervals. This serves as a proxy for the actual live interval extension of the next (B) register preservation phase. When the number of the overlapping live intervals becomes greater than the number of physical registers available in the underlying processor, a stack-spill store might be generated thereafter. Therefore, a region boundary, i.e., store fence, is placed at that point. That way ReplayCache can maximize the size of the store-register-preserving region, likely with no spill.

Figure 3.5(a) shows an example code where there are variables x, y, z and their live intervals; x and y are used as store operands, and their live intervals overlap in basic block A as shown in the left of the figure. Suppose there are only 2 physical registers. Figure 3.5(b) demonstrates how the register-pressure aware region partitioning works for the example code. Basically, whenever stores are encountered, the algorithm carries the live interval of their operands for the rest of the CFG traversal. For example, when the traversal hits the store y at the end point of basic block B in the left control path, the algorithm will start carrying the live interval of y thereafter (illustrated as a hatched box in the figure); the same action is taken with the store x in the right path. Thus, when the traversal hits the point where z's assignment is found in the join basic block D, the live intervals of both x and y have been carried to the point. Since z's live interval starts there, the algorithm places a region boundary at that point, which would otherwise end up making the number of overlapping intervals (3 thereafter) bigger than the number of physical registers (2).

3.3.2 Regular Store Register Preservation

Once regions are formed by the register-pressure aware region partitioning, ReplayCache compiler enters register allocation. Then, this register preservation phase "preserves" the variables used for the operands of stores. The goal is to ensure that no other variables are assigned to those registers that are supposed to be occupied only by store operands. To achieve this, this phase extends the live interval of store operand variables from their last use point *to the end of the region* to which they belong, along the control path.

For example, as shown in Figure 3.5(c), the actual live intervals of x stops at its last use point in basic block C, the resulting interval is extended to the next region boundary placed in the middle of the bottom basic block D; similarly, y's interval is extended to the same following region boundary. In this way, x and y never share their physical registers—even after their last use point—with other variables. In other words, the next register allocation phase ensures that neither x nor y is assigned to any physical register used by other variables. Consequently, ReplayCache guarantees the integrity of the regular stores' registers.

3.3.3 Stack-Spill Store Register Preservation

The register allocation might spill some variable to stack and generate the stack-spill stores. This actually happens since register allocation performs in a function level (not a region level) and makes a global decision across all the regions in a function—though the (A) register-pressure aware region partitioning tries to form spill-free regions in a best-effort manner. Just in case, this stack-spill store register preservation phase searches the register-allocated code of each region for any update on the spill store registers. For example, in Figure 3.5(d), a $r1$ is spilled to the stack in basic block D, i.e., the stack-spill store of $r1$ is generated there. However, in the region, the spill store is followed by the instruction that changes the $r1$, i.e., $r1 = r1 \ll 2$. Thus, the region cannot guarantee the integrity of $r1$ used by the stack-spill store from that moment. To deal with this problem, this phase places an additional region boundary right before the register updating instruction to separate it from the stack-spill store; the resulting boundary is shown near the bottom of basic block D in

Figure 3.5(d). Consequently, ReplayCache compiler guarantees the integrity of all the store registers in all regions.

CLWB insertion: Once register allocation ends, after which no store is generated, the compiler inserts a CLWB instruction right after each store in regions. Since CLWB instructions reuse the address operand of the preceding store, they make no side effect other than the instruction count increase.

3.4 Recovery Protocol

This section describes (A) how ReplayCache compiler generates recovery code and (B) the details of recovery procedure, and (C) finally explains a running example.

Recovery Code Generation: To recover from power failure, as a software-only design without hardware support, ReplayCache compiler generates a recovery code block for each region, which contains all the necessary information and code for the recovery of the region. A recovery code block consists of *Recovery Code*, which is a code to re-execute all stores in the corresponding region, and two maps—*Recovery Map (RM)* and *Store Counting Map* (CM)—to locate the corresponding recovery block and the number of stores to be re-executed for recovery. An RM is a map from a region boundary PC to an address of region recovery code. A CM is a map from a region boundary PC to a *Store Counting Table (SC table)*, which is an array of store addresses and the number of store instructions from the beginning of the region to this store. With these generated recovery code and maps, ReplayCache's recovery protocol figures out where the recovery code of the interrupted region is and how many stores should be re-executed in the interrupted region before the failure point.

In particular, to ensure the absence of power failure during the recovery process, ReplayCache compiler leverages the EH model [203] to estimate the worst-case execution energy of the recovery code block. If the energy is greater than what the underlying capacitor can deliver with it full capacitance[1], the compiler splits the corresponding region into two smaller regions and generate their recovery code blocks; this process is repeated unless the resulting code blocks are small enough to complete with the fully charged capacitor. In this way,

[1]↑Energy harvesting systems do not reboot across power failure until the capacitor is fully charged, which is the case for commodity systems such as NVP, WISP, and QuickRecall.

ReplayCache guarantees the power-failure-free recovery. According to experimental results (§3.5), ReplayCache regions are not that long; we have not encountered any regions that must be split during our evaluation of total 23 benchmark applications.

Recovery by Re-execution: ReplayCache's region-level persistence guarantees that all the stores in preceding regions are persisted. However, stores in the interrupted region before the power outage may or may not be persisted. ReplayCache recovery protocol relies on two properties: First, upon power outage, ReplayCache processor checkpoints registers (including PC) just-in-time by signaling voltage monitor (NVP) or runtime (QuickRecall). The register checkpoint is thus available in either NVFF (NVP) or checkpointing storage in NVM (QuickRecall). Next, ReplayCache compiler ensures that registers used for store operands are never overwritten within a region. This implies that ReplayCache can restore memory status from potential corruption by re-executing the recovery code generated by the compiler.

When the power comes back, ReplayCache first finds out the start address of an interrupted region. It loads the checkpointed *region register* – a dedicated general-purpose register by compiler as mentioned in §3.2.2 – from NVFF or checkpointing storage, and locates the recovery code and the SC table of the interrupted region by looking up the RM and CM, respectively. ReplayCache gets the number of store instructions to be re-executed from the beginning of the region to the failure PC by performing binary search of the SC table with the region register as a key. Subsequently, ReplayCache runtime jumps to the recovery code of the interrupted region with the re-executing store count in a register. As illustrated in Figure 3.6, the recovery code is a series of re-executing the store instruction, decrementing the store counter, and checking if the counter is zero. After executing the specified number of store instructions (i.e., the store counter becomes zero), ReplayCache runtime signals voltage monitor to restore register files from either NVFF or checkpointing storage and thus goes back to the failure point because PC now points to the failure point.

A Running Example: We illustrate a recovery example in Figure 3.6. ReplayCache compiler ensures that registers that are used for store operand ($r1$, $r2$, and $r3$) are never updated in region $R1$. When entering into $R1$, ReplayCache sets the region register to the beginning of $R1$. When a power outage happens in the region indicated by a red cross, all

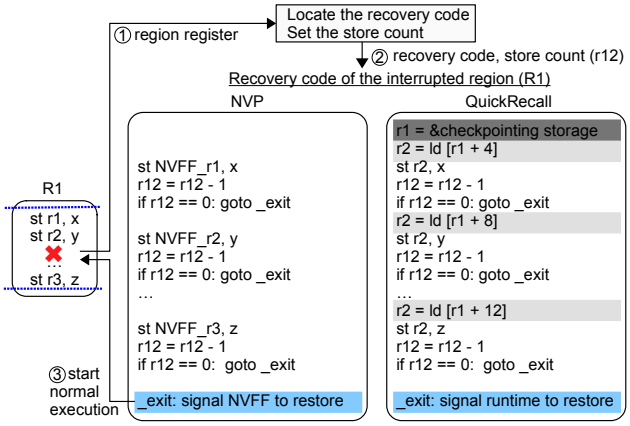


Figure 3.6. Failure recovery of region $R1$ when the power outage happens in the middle of basic block A. Upon recovery, ReplayCache locates a recovery code and counts the number of stores needed to be re-executed ①. Then it re-plays all stores in the recovery block by using checkpointed store operand registers in NVFF ②. Finally, it goes back to failure the point by restoring registers from NVFF and continues the normal execution ③

registers, including the region register and PC, are checkpointed. At this point, the stores to memory locations x and y may or may not be persisted due to the volatile cache.

When the power comes back, ReplayCache first loads the region register, which points to the beginning of the interrupted region. Then it locates the corresponding recovery code and the number of stores to be re-executed from the RM and SC table ①. ReplayCache jumps to the recovery code to re-execute the same number of store instructions in the region before the failure ②. In the recovery code examples in Figure 3.6, $r12$ is the number of stores to be re-executed during recovery. In the recovery code, ReplayCache runtime loads the checkpointed store operand registers (e.g., $NVFF\_r1$ in NVM, and $ld\ [r1 + 4]$ in QuickRecall) and re-executes store instructions. Once ReplayCache runtime re-executes the same number of store instructions – i.e., all store instructions to the failure PC are re-executed, the store counter ($r12$) becomes zero and the runtime prepares to resume the normal execution ($goto\_exit$ colored in blue). The runtime signals voltage monitor to restore register files from NVFF and jumps to failure point ③. The recovery code are slight different between NVP and

QuickRecall. As shown in the right, QuickRecall loads the checkpointed registers from the storage (colored in gray).

3.5 Evaluation

We implemented all ReplayCache compiler passes using the LLVM compiler infrastructure [139]. In particular, we implemented our LLVM passes on MIR (Machine IR) level after instruction selection to precisely measure the number of live intervals during the region construction. The all compiler passes consist of about 1700 LOC excluding comments.

We evaluate ReplayCache using a gem5 simulator [204] with ARM ISA, modeling a single core in-order processor with 16 registers, based on the NVPsim [205]; Table 3.1 summarizes our NVM write/read latency based on [205–208]. In particular, we only modified L1D cache leaving L1I cache as NVM cache as with the original NVP [189]. Note that ReplayCache works for any energy harvesting processors that support just-in-time (JIT) register checkpointing. In addition to NVP, we test ReplayCache on top of QuickRecall whose simulation configuration follows that of NVP other than the JIT checkpointing/restoration parameters. Table 3.2 shows the detailed simulation parameters of NVP and QuickRecall. Since QuickRecall checkpoints registers in NVM, its checkpoint/restore voltage thresholds are higher than those used by NVP.

In addition to ReplayCache, we test 3 alternative cache designs: non-volatile cache (NVCache), non-volatile SRAM cache (NVSRAM), and volatile write-through cache (WTVCache). All 4 cache designs are assumed to run with NVP unless noted otherwise. Especially for NVSRAM, we use the same configuration used by NVPsim [205], which is based on advanced ReRAM technology. That is, it writes 3x faster with 5x less energy compared to conventional ReRAM based non-volatile main memory does. Similarly, it reads 2x faster with 24x less energy compared to the main memory does. Thus, NVSRAM here serves as the upper bound for performance comparison due to the forward-looking technology used. As our default setting, we set the size of all the caches to 8KB, and they are all 2-way set-associative. For non-volatile main memory, we used Re-RAM by default and set its size

as 16MB by leveraging NVMain [206]. We also perform sensitivity studies with STT-RAM and PCM using the parameters in Table 3.1.

Table 3.1. The timing parameters (ns) of different NVM technologies: e.g., tCK stands for clock period

NVM	tCK	tBURST	tRCD	tCL	tWTR	tWR	tXAW
ReRAM (default)	0.94	7.5	18.0	15.0	7.5	150	30
STT-RAM	1.5	6	35	15	12.5	25	50
PCM	1.88	7.5	48.0	15.0	7.5	300	50

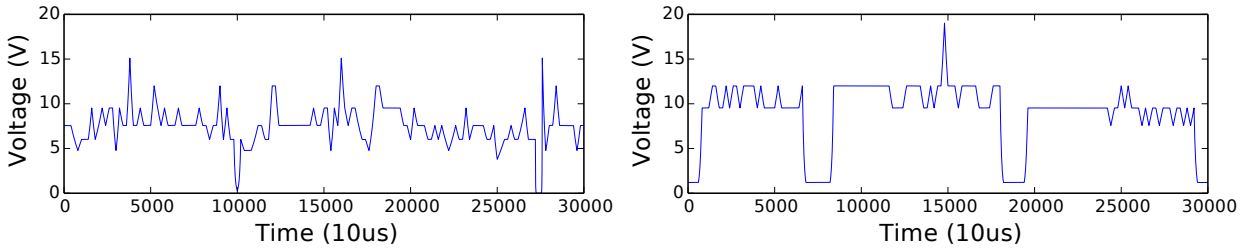
Table 3.2. Simulation configuration

	NVP (default)	**NVP** (NVSRAM)	**QuickRecall**
Vmax/Vmin[98]	3.3/2.8	3.5/2.8	3.5/2.8
Ckpt/Restore[98]	2.9/3.2	3.2/3.4	3.1/3.3
Recovery	NVFF+Cache	NVFF+Cache	VFF+Cache

We use 8 applications in Mibench [186] and 15 applications in Mediabench [187] benchmark suites. All the applications are compiled by ReplayCache compiler with -O3 optimization level. To evaluate ReplayCache for realistic energy harvesting environment with frequent power outages, we use two power traces of the NVPsim which were collected from real RF energy harvesting systems [205]. Figure 3.7 describes the shape of those two power traces; (a) shows the voltage fluctuations across time in home, and (b) shows those in office. Trace 2 (office) has more power outages than Trace 1 (home); in every 30 seconds, Trace 1 and 2 incur ≈20 and ≈400 power outages, respectively.

3.5.1 Performance Comparison

Performance without Power Outage: Figure 3.8 shows the performance results of power-failure-free executions. The Y-axis shows the normalized speedup over the baseline without a cache. Overall, ReplayCache improves the performance of all the applications, achieving 11x speedup on (geometric) average. It turns out that NVCache is the worst design



(a) Power Trace 1 (Home)

(b) Power Trace 2 (Office)

Figure 3.7. Energy harvesting traces showing voltage input fluctuations in two different places within about 250∼400ms from an RF energy harvesting reader [205]

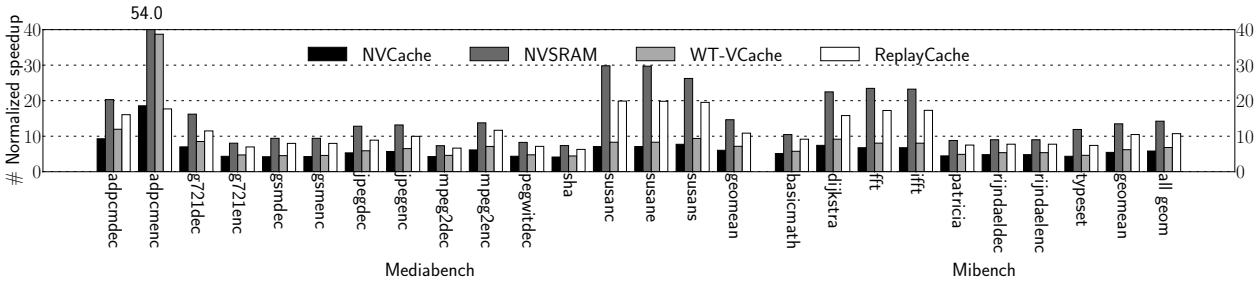


Figure 3.8. Performance results "without" power outages. We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache. The higher, the faster

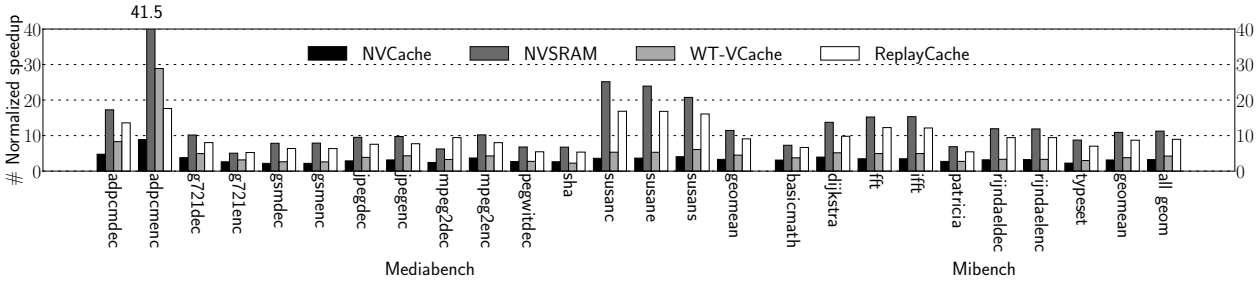


Figure 3.9. Performance results "with" power outages, simulated with Power Trace 1 in Figure 3.7(a). We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache

as expected because of higher latency (especially stores) then SRAM, but it still improve the performance due to locality exploitation.

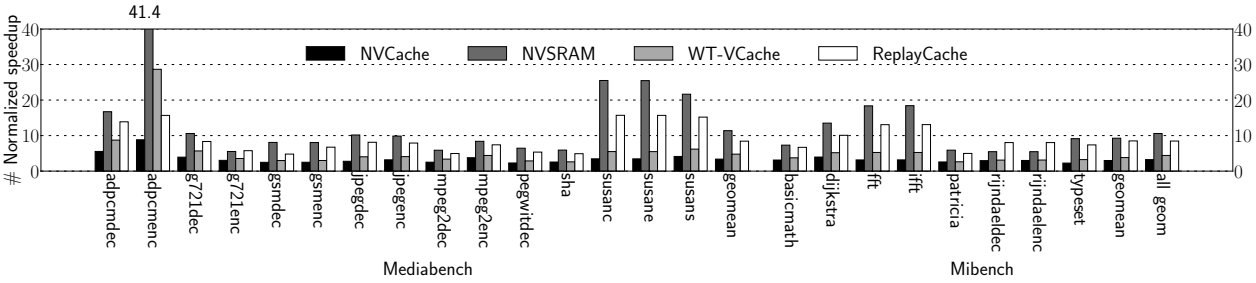


Figure 3.10. Performance results "with" power outages, simulated with Trace 2 in Figure 3.7(b). We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache

Recall that NVSRAM uses a traditional SRAM cache with an NVM (advanced ReRAM) backup, and checkpoints/restores the whole cache state to/from the NVM backup across power failure. Thus, with no power outage, NVSRAM should perform as an original write-back volatile cache. NVSRAM performs the best as expected achieving 14x speedup compared to the baseline. Here, the performannce gap between NVSRAM and ReplayCache results from the store write-back latency that our region-level persistence did not manage to fully hide with ILP. Later in §3.5.2, we present the detailed results on ReplayCache's ILP efficiency, reflecting the amount of stalls at the region boundary.

WT-VCache shows some improvement over the baseline without a cache. The performance benefits mostly come from load hits, though the write-through policy makes the cost of store the same as the baseline. ReplayCache outperforms WT-VCache, i.e., achieving an average speedup of 1.57x, by hiding the latency of stores with region-level persistence.

Performance with Power Outages: Figures 3.9 and 3.10 show the performance results with power failure, simulated on Power Traces 1 and 2 in Figure 3.7. The simulation includes different sequences of power up/down and downtime during charging. Again, the Y-axis is the normalized speedup over the baseline without a cache.

Although NVCache uses the same NVM technology as main memory, it can be placed close to a core as cache so that access latency to NVCache is fast than NVM, exploiting the

locality. NVCache remains the worst mainly due to a long cache access latency and higher energy consumption of NVM access wasting hard-won energy.

With power outages, ReplayCache achieves almost 80% performance of NVSRAM. This is a promising result given that ReplayCache is a software-only scheme that allows commodity systems to use a volatile data cache as is with no other additional hardware support. Note that NVSRAM cache can retain the cache data across a power outage while ReplayCache cannot since it uses a traditional SRAM cache that loses all the content upon the outage; due to this advantage, NVSRAM beats all other cache schemes. In contrast, when power comes back, ReplayCache has to start with a cold cache reloading all necessary data from NVM. Nevertheless, the cache warming-up cost can be amortized by the benefit of cache hits, unless the program execution is too frequently interrupted by power failure.

WT-VCache shows only comparable performance to the expensive NVCache design due to the cost of warming up the volatile cache across power failure and serializing stores with the write through policy. However, WT-VCache still outperforms the baseline with exploiting certain degree of locality. In particular, WT-VCache outperforms ReplayCache for *adpcmencode*. That is because the ReplayCache ended up increasing the instruction count due to a register spilling in a hot loop along with the stack memory access cost. On average, WT-VCache performance happens to be almost same as NVCache design.

On average, ReplayCache achieves 8.95x (Trace 1) and 8.46x (Trace 2) speedups compared to the baseline without a cache, significantly outperforming both NVCache and WT-VCache. The reason for the performance gain over them is two-fold. First, ReplayCache costs less cache power consumption compared to the NVCache and WT-VCache as shown in Figure 3.11. Second, due to the ILP nature, ReplayCache can hide the most of write-back latency as will be shown Figure 3.12.

Energy Consumption Breakdown: To figure out the energy consumption behavior of ReplayCache, we measured how much energy was consumed for each part of the system, i.e., cache, memory, and core (NVP computation), by using the power model provided by NVPsim [205]. Figure 3.11 shows the resulting energy consumption breakdown, normalized to the same no-cache baseline, using the Power Trace 2. Overall, ReplayCache turns out to be very effective, allowing NVP to spend more energy for computation rather than memory

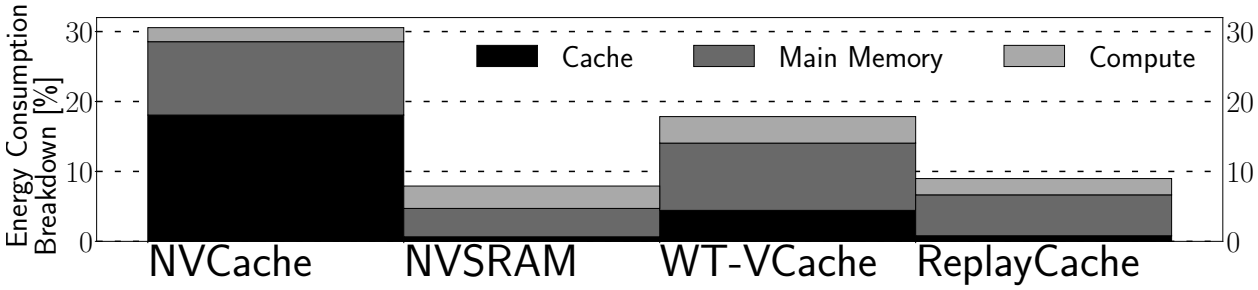


Figure 3.11. Normalized energy consumption breakdown (trace 2) compared to the baseline without a cache

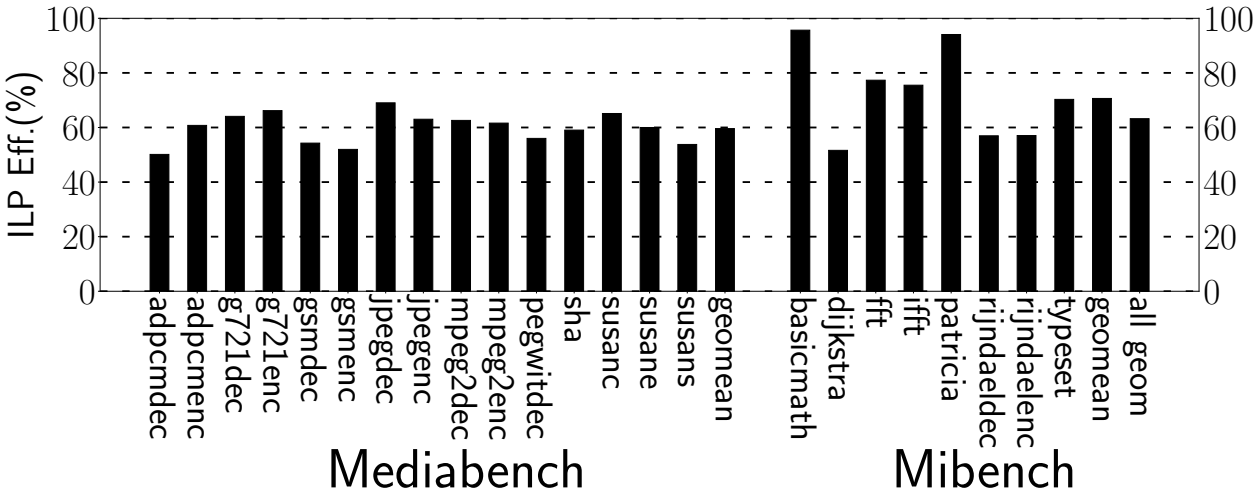


Figure 3.12. Instruction-level parallelism efficiency "without" power failure

access compared to other schemes. Also, ReplayCache's energy consumption is on par with the ideal NVSRAM. As a result, ReplayCache enables NVP to make a significantly further forward progress than the no-cache baseline.

3.5.2 Instruction Level Parallelism Efficiency

ReplayCache exploits ILP for stores and thus is faster than a volatile write-through cache. Nevertheless, its ILP can be bounded by region-level persistence guarantee, e.g., a region end is reached before the preceding store completes the NVM persistence, in which case ReplayCache is slower than an ideal write-back cache. With that in mind, we investigate

the amount of ILP that ReplayCache can exploit, based on the power-failure-free simulation results, to reason about ReplayCache's high performance.

Let N be the total number (dynamic instances) of stores in a region. Among them, N_{no_stall} represents the number of stores that do not stall, and N_{stall} represents the number of stores that stall at the region boundary for region-level persistence guarantee. Let C be the cycles required for a store to be persisted in the NVM (i.e., the write-through NVM store latency; 31 cycles in our evaluation for default ReRAM); and $S(\mathrm{i})$ be the stall cycles of i's store in the region. We then calculate the ILP efficiency at a 0-to-100% scale. For each store, the worst efficiency 0% is made when the processor waits for C cycles after the region finishes, and the best efficiency 100% reflects 0 stall cycle. Equation (3.1) defines the ILP efficiency for N stores in a region as follows.

$$ILP_{eff}(\%) = \frac{1}{N}\{\sum_{\mathrm{i}=1}^{N_{no\_stall}} 1 + \sum_{\mathrm{i}=1}^{N_{stall}} (1 - \frac{S(\mathrm{i})}{C})\} * 100 \tag{3.1}$$

Figure 3.12 shows the ILP efficiency of the tested applications. On average, ReplayCache achieves 63% ILP across the evaluated applications, and the ILP efficiency explains why ReplayCache achieves the performance shown in Figure 3.8. Again, in our evaluation, the write-through store latency takes 31 cycles [205], i.e., $C = 31$. This implies that ReplayCache can hide about 20 cycles out of the 31 cycles on average.

3.5.3 Binary Size Analysis

Figure 3.13 demonstrates the breakdown of binary size increase of ReplayCache binaries as a percentage increase compared to the baseline binary. Overall, ReplayCache incurs only 1.2% binary size overhead on average. Metadata operations are comprised of roughly 110 instructions, leading to near-zero overhead. Only 2 applications, e.g., `jpeg` and `typeset`, have observable binary size increase because they have lots of small regions. It is important to note that the binary size overhead never puts pressure on application's memory usage at run time. That is because the metadata is accessed only at boot time on which ReplayCache's recovery starts with empty cache—already wiped out upon the prior failure—without cache pollution.

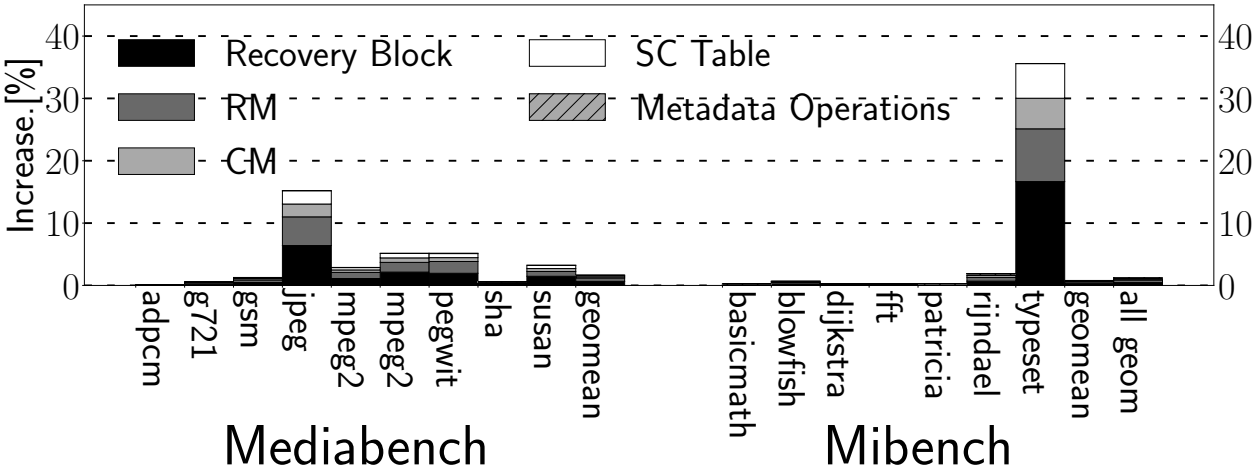


Figure 3.13. Binary size increase due to recovery block, metadata (RM, CM, SC table), and metadata operations (code)

3.5.4 Dynamic Instruction Count Analysis

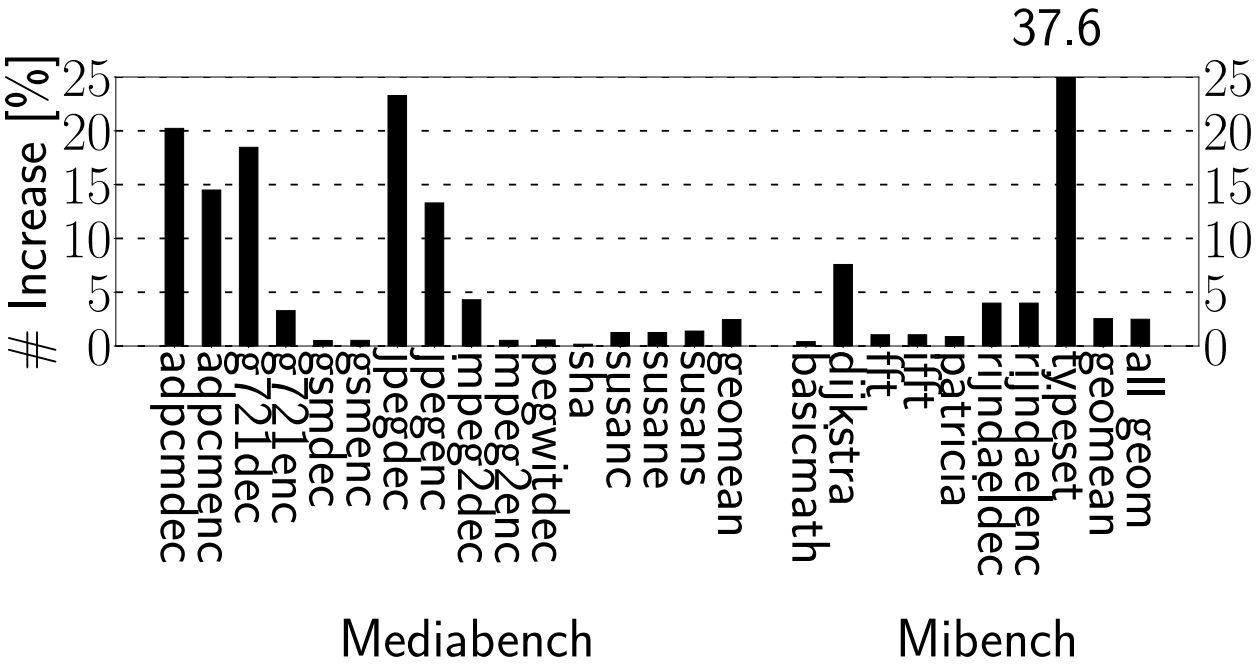


Figure 3.14. Dynamic instruction count increase due to ReplayCache compiler code generation; lower is better

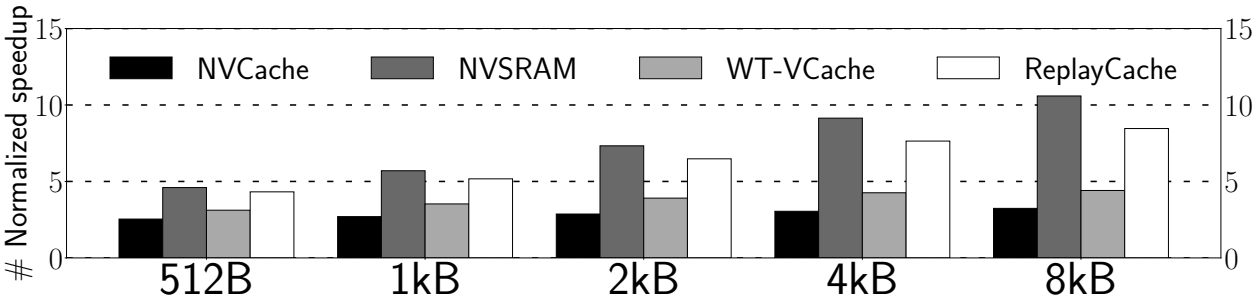


Figure 3.15. Cache size sensitivity analysis for Trace 2

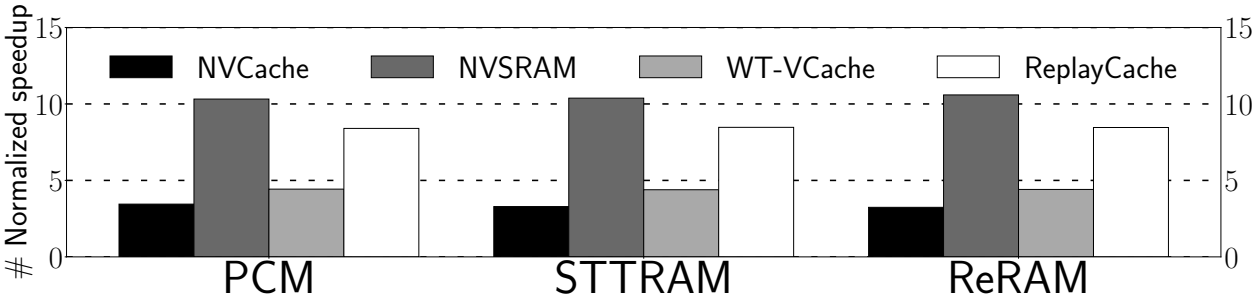


Figure 3.16. Sensitivity study on different NVMs with trace 2 used

Figure 3.14 demonstrates that ReplayCache compiler only increases dynamic instruction count by 2.49% on average compared to the baseline binary. Note that this is not a critical performance limiting factor as confirmed in Figure 3.8-3.10 where ReplayCache consistently shows significant speedups.

3.5.5 Sensitivity Study

Cache Size: Figure 3.15 shows the normalized execution time (to the baseline without a cache) of alternative cache schemes with a different cache size from 512B to 8KB using Power Trace 2. The results show that ReplayCache matches the performance of NVSRAM cache (that is an ideal write-back cache in power-failure-free scenarios) for small cache size, such as 512B and 1KB.

NVM Technology: Different NVM technologies (e.g.,, ReRAM, PCM, and STT-RAM) have different write/read latency properties and their respective ratio, as summarized in Ta-

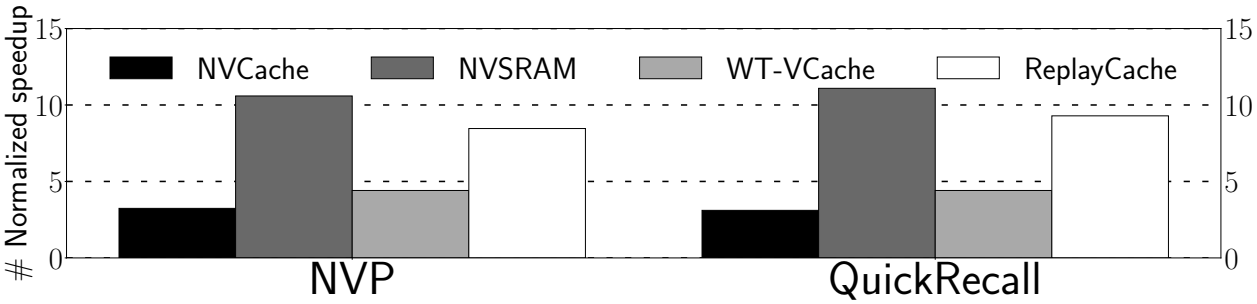


Figure 3.17. Performance overhead comparison with trace 2

ble 3.1. For ReRAM, PCM, and STT-RAM (as the main memory), Figure 3.16 shows the normalized speedup of alternative cache schemes, compared to their 3 baselines without a cache. It turns out that ReplayCache consistently achieves significant speedups across the NVM technologies (8.4x-8.46x).

NVP versus QuickRecall: To analyze the impact of the underlying just-in-time register checkpointing on ReplayCache's performance, we tested all four cache schemes on top of QuickRecall and compared the results with those of NVP. Again, we used the Power Trace 2 and normalized the speedup over their baselines, i.e., NVP/QuickRecall without cache. Figure 3.17 describes that the performance trend is similar to NVP; however, it is worth noting that QuickRecall requires higher checkpoint/restoration voltage due to data backup as shown in Table 3.2—though it is a less expensive system than NVP due to the lack of non-volatile flip-flops.

3.5.6 Region Statistics

We study the region statistics, statically calculated from the binary built by ReplayCache compiler. Figure 3.18 presents the average number of instructions per region. On average, there are 16.4 instructions per region. We also break them down into two categories: stores and other instructions. On average, there are 2.18 stores and 14.35 others per region. This implies that the recover code blocks are not long either, even smaller than their regions. In

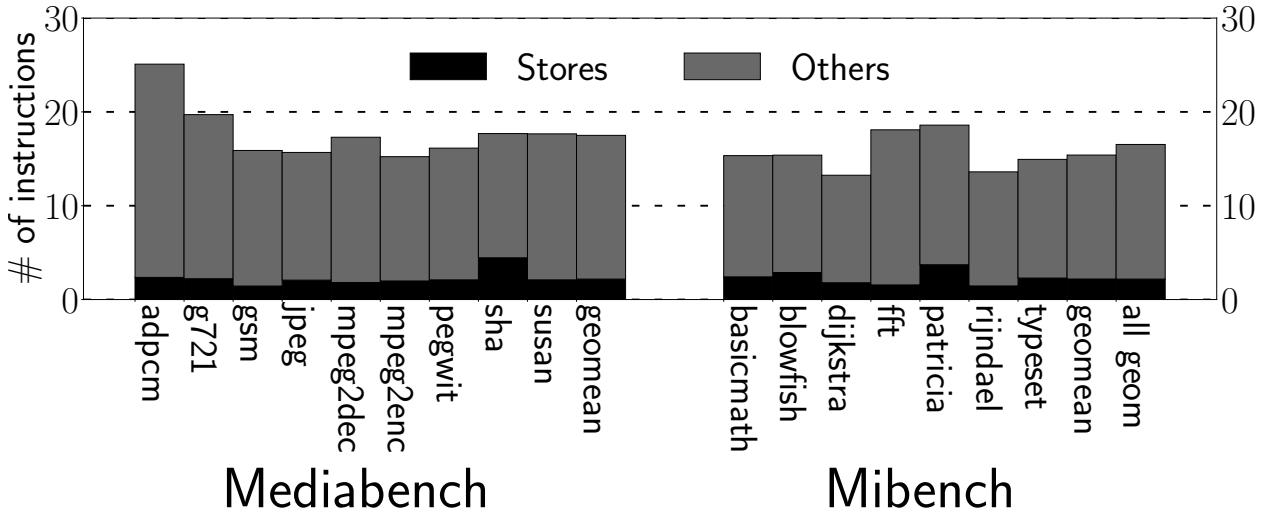


Figure 3.18. Breakdown of per-region instructions on average

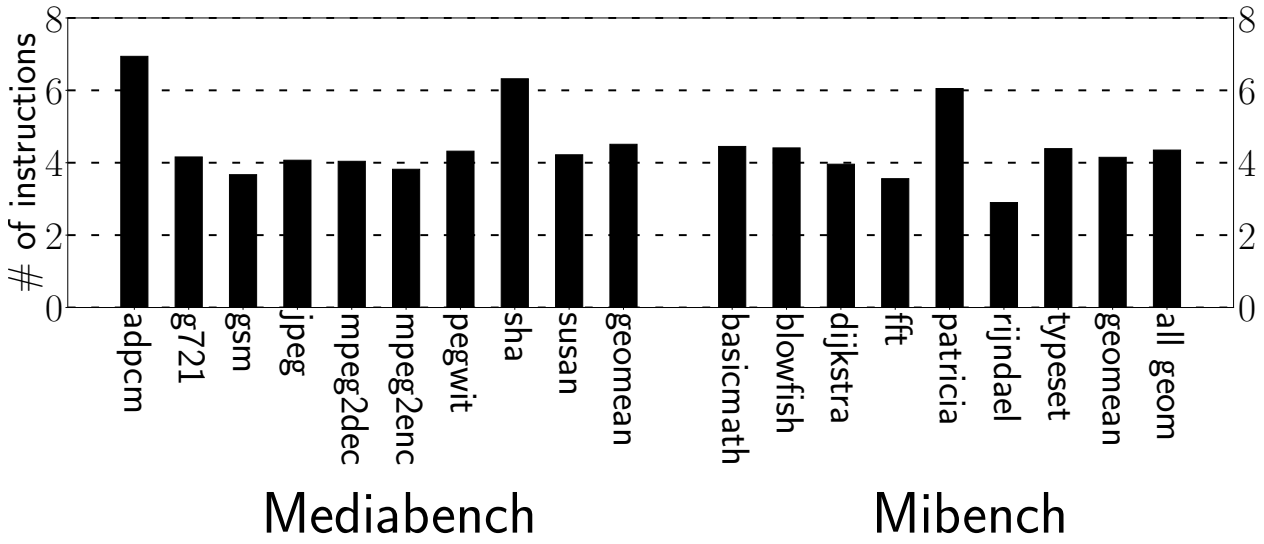


Figure 3.19. Average distance (the number of instructions) between region's last store and the following region boundary

fact, we did not encounter any recovery code block that requires the corresponding region to be split to ensure the absence of power failure during the recovery.

Moreover, Figure 3.19 shows the average distance (the number of instructions) between the last store of a region and the following region boundary, i.e., 4.35 instructions on average. The distance here reflects ReplayCache's ILP opportunities.

3.6 Other Related Work

Many prior works [172–178] have been proposed to leverage non-volatile caches to speed up the performance and leverage their zero standby leakage and crash consistency free properties. However, the cell endurance of NVM techniques ranges from 10^5 in flash to 10^{12} in STT-RAM. Non-volatile caches may only be able to endure few months for most of real applications [174]. Thus, prior works focus on increasing the lifetime of NVM cells. Furthermore, NVM has the asymmetric performance property. A write is considerably slower than a read, compared to the SRAM counterpart. Both the short lifetime and the long write latency severely limit the use of NVM as L1 cache in practice.

To combine the benefits from NVM and SRAM, many researches [180–185, 196, 197, 199, 200, 209] proposed to incorporate different NVM technologies (e.g., STT-RAM, ReRAM, etc.) with SRAM. Many proposals leverage the NVM part as a just-in-time checkpointing storage of the traditional SRAM-based cache in case of power failure. Thus, the NVM speed is the critical aspect for the success of such SRAM/NVM hybrid design. Although researchers attempt to improve the NVM backup/restoration latency [180, 181], they assume forward-looking technologies; no current NVM technologies provide comparable latency to SRAM [59, 188].

The idea of partitioning a program into multiple regions to design more efficient energy harvesting systems has been explored. Ratchet [202] proposed to partition program into a series of anti-dependence-free (i.e., write-after-read dependence free) regions for idempotent processing as with others [57, 115, 116, 130, 132, 134]. Since idempotent regions can be safely re-executed multiple times, it can recover a power-interrupted region by rolling back to the beginning in the wake of power failure, provided the inputs value of the region can survive the power failure. Due to the absence of the anti-dependence, Ratchet only needs to checkpoint all live-in registers of the region at its entry point. Unfortunately, such consecutive NVM writes are not only expensive but also dangerous increasing the chance of power failure in the middle of their writes. To address the issues in Ratchet, Clank [97] proposed hardware-based idempotent processing. Despite its improved performance, Clank requires relatively heavy and complex hardware components such as a fast scratchpad memory for speeding

up the writes to the underlying NVM and an expensive CAM (content-addressed matching) search based load/store address tables to dynamically detect anti-dependence. Alternatively, CoSpec [59] proposed power failure speculation assuming that power failure is not likely to occur. Thus, it buffers all the application writes in a store buffer in case of misspeculation, i.e., actual power failure. Also, the CoSpec compiler partitions program into a series of regions so that they never overflow the store buffer. When power failure occurs in the middle of a region, it is rolled back to the beginning in the wake of power failure. As with Ratchet, CoSpec needs to pay the overhead of checkpointing all live-in registers of every region. Unlike ReplayCache, neither Clank nor CoSpec supports a volatile data cache. Thus, we suspect that ReplayCache can significantly outperform them.

3.7 Summary

This chapter proposes ReplayCache, a software-only scheme that enables energy harvesting systems to take advantage of a volatile data cache efficiently and correctly. To achieve crash consistency with the volatile data cache, ReplayCache proposes a replay-based solution that restores the operands of potentially unpersisted stores from the register checkpoint and then re-executes them to restore consistent non-volatile memory status. Experimental results show that compared to the baseline with no cache, ReplayCache significantly improves the performance by 8.46x-8.95x speedup on geometric mean, while ensuring correct resumptions even in the presence of unpredictable and frequent power outages.

4. VERIPIPE: NEAR-ZERO COST SOFT ERROR RESILIENCE

Soft errors have been the root cause of many real-world failure, especially for the large-scale high-performance computing (HPC) systems and datacenters [1–17]. In general, soft errors—also known as transient faults—are predominantly caused by the striking of high-energy particles[1], e.g., cosmic ray and alpha particle from packaging materials, resulting in program crash or even worse silent data corruption (SDC) [210]. Soft error resilience becomes more important in the era of post-Moore's Law where near-threshold computing (NTC) plays a critical role in improving energy efficiency [211]. However, NTC makes the systems more vulnerable to soft errors, increasing their rate up to 30x higher [212, 213] than that of those systems with nominal voltage [211]. Thus, effective yet lightweight methods for detecting and mitigating soft errors are absolutely necessary not only to ensure program correctness but also to realize the full potential of NTC.

This necessity has sparked interest in innovative detection techniques. Among them, acoustic-sensor-based detection [84] is particularly prominent as the acoustic sensors eliminate silent data corruption (SDC); they directly sense the sound wave, which is always produced by particle striking as a physical phenomenon, thereby guaranteeing the full detection of soft errors, i.e., none of them is missed. More importantly, the acoustic-sensor-based detection incurs minimal hardware costs [84]; only 0.0001% areal cost of the chip area ($0.7mm^2$; see Section 3.5) needs to be paid to deploy 30 sensors without requiring an extra metal layer for their interconnection.

Given the guaranteed error detection capability of acoustic sensors, researchers have leveraged them to realize soft error verification with the sensing latency in mind [40, 84, 214, 215]. The idea is that program execution prior to a given time T can be verified to be error-free the worst-case-detection-latency (WCDL) cycles later, provided that no sensor alarms in between. This makes senses because every soft error is to be detected within the WCDL cycles after its occurrence. In light of this, Liu et al. [40] proposed Turnstile to achieve core-level error containment. It enforces that data to be written must be error-free when

[1]↑Because of this, VeriPipe targets only such radiation-induced errors which we refer to as soft errors hereafter for simplicity.

they leave the core—with caches and memory protected by error-correcting code (ECC)—for no error to escape from the core.

To contain soft errors in the core, Turnstile delays writing the data of retired stores back to the L1 data cache, until they are verified to be error-free. That is, Turnstile leverages the store queue (SQ)[2] of each core as a redo buffer [121] to hold the retired stores for at least WCDL cycles till their data turn out to be verified. To avoid the SQ overflow during region execution, which would otherwise lead to incorrect error recovery, Turnstile compiler partitions program into a series of regions—comprising a sequence of instructions possibly including branches. As such, the stores of each region are verified as a whole once WCDL cycles are passed since the end of the region. This is so-called *region-level error verification*. Notably, Turnstile turns the verification of registers into the memory verification by inserting stores to checkpoint region's *live-out* [136] registers—used by some following regions as inputs—to the memory.

Even though soft error resilience is vital nowadays, there is no practical solution for it yet. The state-of-the-art work Turnstile is not practically implementable. The reason is twofold: (1) its microarchitectural modifications are intrusive pressuring out-of-order pipeline optimization at design time; (2) it incurs a non-trivial run-time overhead, i.e., 9% on average and up to 53%. To unveil why Turnstile leads to the high hardware complexity and the significant performance degradation, VeriPipe presents a new viewpoint of the *region-level error verification*. In Turnstile, all regions go through a three-stage (*Execute*, *Verify*, and *Commit*) verification pipeline. Each region begins with the *Execute* stage and transits to the *Verify* stage at the end of the region. Spending WCDL cycles thereafter (more precisely if no error is detected in the *Verify* stage for the WCDL cycles), the region finally moves to the *Commit* stage finishing the region verification. Upon the *Commit* of each region, Turnstile signals the store queue (SQ) to release the region's stores to the L1 data cache. With the help of this verification pipeline, Turnstile could overlap the *Verify* of an old region (i.e., executed but unverified) with the *Execute* of a younger region, thus hiding the verification latency and achieving instruction-level parallelism (ILP).

[2]↑We assume a unified store queue in out-of-order cores without differentiating store buffer from store queue.

Nonetheless, the *Execute* of a younger region is not always long enough to fully cover the *Verify* of the prior region, which causes the CPU pipeline to stall degrading the performance (see Section 4.1). Moreover, the *Verify* latency increases as WCDL goes up, i.e., the CPU pipeline stalls more frequently, and each stall takes longer. In case the *Verify* delay cannot be fully hidden by a single region's *Execute* latency, Turnstile introduces a special hardware FIFO queue called region boundary buffer (RBB) that schedules multiple following younger regions for their execution time to overlap with the *Verify* of the oldest unverified region sitting at the RBB head.

Apart from the chip area occupancy, Turnstiles RBB puts significant pressure on realizing high-performance out-of-order pipeline—whose timing is already highly optimized—with the related control/signal and the critical path extension due to RBB-overflows stalling the pipeline. The crux of the problem is that this design challenge eventually prevents Turnstile from being fabricated on top of real silicon. In addition, Turnstiles register checkpoints (essentially stores), inserted for turning register verification into memory verification, are sometimes too many, thus incurring a non-negligible run-time overhead.

Thanks to our 3-stage pipeline modeling of Turnstile, we found out that its verification hardware can be dramatically simplified to only three registers and one countdown timer. In fact, only one region on the *Execute* stage is enough—if it is sufficiently long—to fully cover the latency of the prior regions *Verify* stage. With the above finding in mind, VeriPipe proposes a simple yet efficient verification pipeline where the *Verify* latency of a region can be hidden by only one following region execution, which obviates the need of complex RBB-like hardware. The key idea is to let each region get verified at the end of the next region. Figure 4.1 shows how VeriPipe's simplified three-stage verification pipeline works. As usual, each region starts with the *Execute* and moves to the *Verify* when reaching its end; however, the region here reaches the *Commit* as soon as the next region finishes with no error detected.

However, it is challenging to ensure that the execution time of each region is never smaller than WCDL cycles. To overcome this challenge, VeriPipe proposes a simple yet effective hardware technique called *region stitching*. At run time, it combines any short region (whose execution time is less than WCDL) with the following regions so that the stitched region's execution cycles are at least WCDL. This allows VeriPipe to fully hide the *Verify* latency of

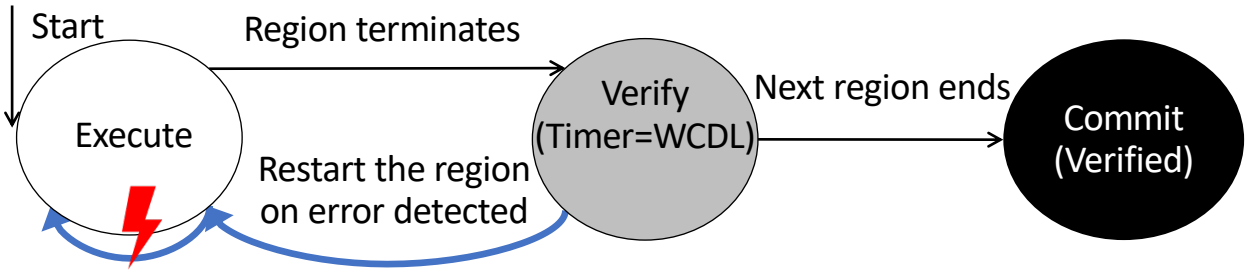


Figure 4.1. VeriPipe's region verification automaton

every region! *The beauty of region stitching is that it scales to arbitrarily long WCDL with neither storage overhead—other than only one logic gate for control—nor run-time overhead.* In addition, VeriPipe compiler eliminates redundant checkpoint stores, lowering the run-time overhead further without jeopardizing the soft error resilience guarantee.

The experiments with 43 applications from SPEC2006/2017 [140, 141], NPB-CPP [216], SPLASH3 [217], and DoE Mini-Apps [218, 219] demonstrate that VeriPipe incurs only a 1% run-time overhead on average regardless of WCDLs. In summary, VeriPipe:

- Incurs near-zero run-time overhead with the intelligent compiler/architecture co-design regardless of long WCDL.
- Incurs only a 0.018% areal cost according to the RTL synthesis results with TSMC 7nm technology, reducing Turnstile's hardware cost and power consumption by ≈91%.
- Is the first to achieve near-zero-hardware-cost soft error resilience, making its commercialization possible in silicon.

4.1 Background and Challenges

4.1.1 Acoustic-Sensor-Based Soft Error Detection

Recently, Upasani et al. [84] proposed to detect soft errors using acoustic sensors. In the event of energetic particle striking (e.g., cosmic ray and alpha particle), the sensors perceive the acoustic wave—generated by the physical phenomenon of the striking—and thus always

detect the resulting soft error. According to the prior work [84], an acoustic sensor only occupies the same die size as one 6T SRAM bit, i.e., $0.027\mu m^2$ with TSMC $7nm$ node [220].

Nevertheless, this detection scheme cannot immediately detect soft errors due to inherent sensing delays. Consequently, errors might bypass the detection and eventually propagate the corrupted data to ECC-protected memory, causing detected-but-uncorrectable errors (DUEs). To address this issue, the prior work [84] includes caches in the error containment domain. It holds L1 dirty cachelines for WCDL until they are verified to be error-free before their writeback to L2. Unfortunately, this design requires significant changes to the existing cache structures and their cache coherence protocol.

4.1.2 Region-Level Soft Error Verification

To tolerate the detection latency without changing caches, Liu et al. proposed Turnstile [40] that contains errors in the core. Turnstile holds data being stored in the SQ for WCDL before merging them to the L1 data cache. To avoid the SQ overflow that leads to incorrect error recovery, Turnstile compiler partitions input program into a series of verifiable/recoverable regions with SQ size in mind. More precisely, each region is formed so that the number of its stores never exceeds the half of SQ size; that way while a region is under verification occupying a half of the SQ, the other half can be used to accommodate the stores of the following region(s). For example, Figure 4.2 (a) and (b) shows that input program is divided into $Rg1$ and $Rg2$ such that each region has at most 2 stores since SQ size is 4 here. Once region boundaries are delineated, Turnstile compiler identifies each region's live-out registers and checkpoints them to the memory for recovery purpose, e.g., checkpoint $1c$ and $2c$ that are essentially stores.

Turnstile hardware checks the integrity of the regions, i.e., the absence of a soft error during their execution, by leveraging the verification pipeline as shown in Figure 4.2 (a). For example, $Rg1$ enters into the *Execute* at $t1$ and moves to the *Verify* as soon as it finishes execution at $t2$. Later, $Rg1$ reaches the *Commit* at $t3$ after spending WCDL cycles since its end, provided none of deployed acoustic sensors alarms the occurrence of soft errors in between.

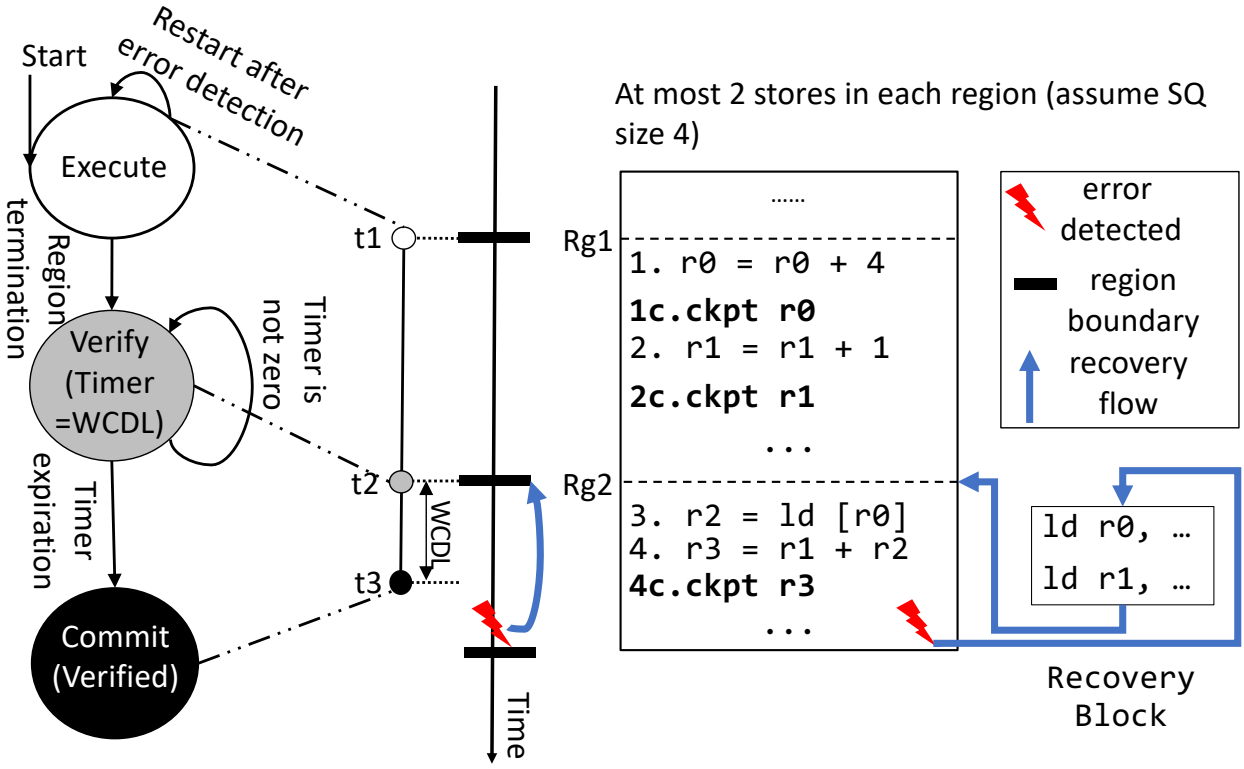


Figure 4.2. (a) Turnstile's region verification automaton; (b) store queue aware region partitioning; eager checkpointing

To implement the above verification pipeline, Turnstile proposes several hardware components—marked gray in Figure 4.3—with the existing store queue (SQ) repurposed as a gated store queue (GSQ). During the execution of each region, the GSQ holds the region's stores until it is verified, though they are already retired from the reorder buffer (ROB). Subsequently, if no errors are detected within WCDL, the verified GSQ entries are to be merged to the L1 data cache. To figure this out, Turnstile introduces a FIFO queue called region boundary buffer (RBB) that tracks the progress of regions being executed or verified. When the ROB commits a region boundary, Turnstile allocates the corresponding entry in the RBB; the core pipeline stalls when encountering a region boundary if RBB is already full. Each RBB entry contains 3 items: (1) the PC of the next instruction, (2) GSQ Ptr—pointing to the tail of the GSQ—for releasing the stores of the region when it reaches the *Commit*, and (3) timing information for determining when the region verification completes, i.e., *Commit*.

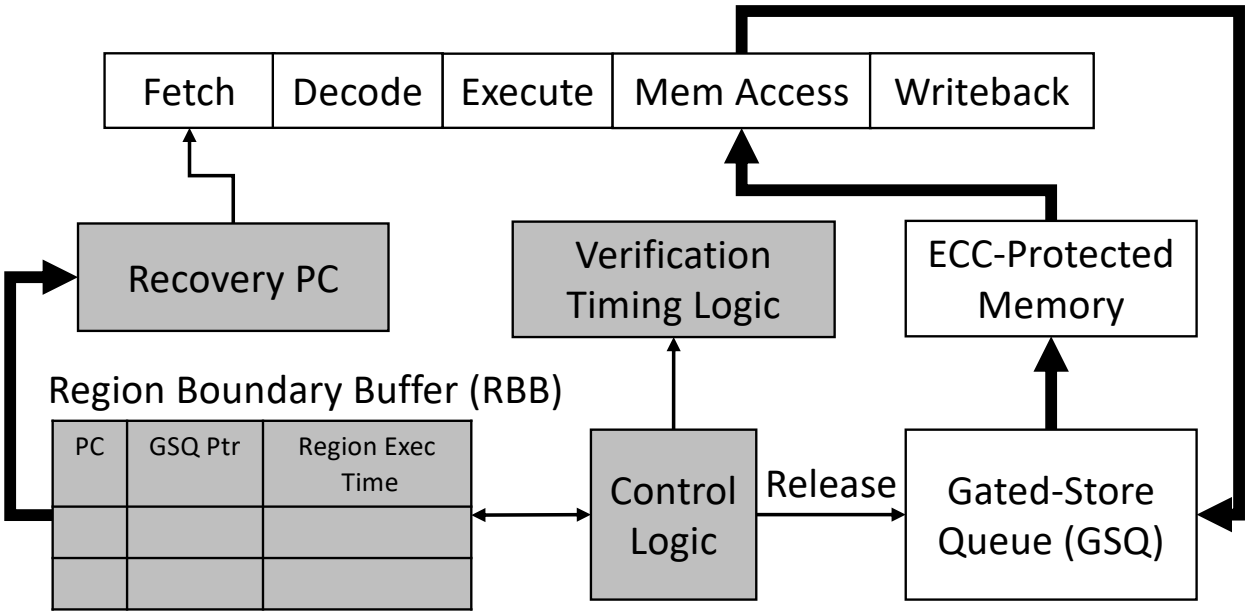


Figure 4.3. Turnstile architectural diagram with marking newly added components gray; bold lines correspond to data paths while thin lines to control paths

With the FIFO nature of RBB, Turnstile keeps the head of the RBB up-to-date with the oldest unverified region. For example, Figure 4.2 shows that $Rg1$ becomes error-free at $t3$. As a result, Turnstile (1) signals the GSQ to write back $Rg1$'s stores (e.g., $1c$ and $2c$) to the L1 data cache, (2) copies the PC field of the head RBB entry to *Recovery PC* in case an error occurs thereafter, and (3) deallocates the head entry with it set to the next entry becoming the oldest among unverified regions.

Upon soft errors (⚡) detected, Turnstile performs three actions to recover from them: (1) discarding (unverified) GSQ entries which are younger than the *GSQ Pointer*, (2) executing the code in a recovery block—shown in Figure 4.2 (b)—to reload the oldest unverified region's live-in registers from a dedicated checkpoint storage in the memory, and (3) resuming the program execution from the region's beginning pointed to by *Recovery PC*.

4.1.3 Limitations of Prior Work

Unfortunately, Turnstile's RBB and its control logic require complex peripheral circuitry, resulting in way longer access latency than VeriPipe, e.g., Turnstile ($0.26ns$) vs VeriPipe ($0.07ns$) as shown in Table 4.1. This implies that Turnstile restricts the core frequency to a maximum of 3.86GHz, making it impossible to be used for current and future high-end processors. Even if future process technologies might be ready for higher clock frequency, Turnstile's intricate peripheral circuitry poses a significant challenge in reducing its wire delay—scaling more slowly compared to transistor delay—as highlighted by prior work [221, 222] on processor core design. Consequently, it is a daunting challenge to increase the clock frequency of Turnstile-enabled processors in the future.

Another prior work Turnpike [214] also leverages GSQ to contain soft errors in an in-order that usually has a tiny SQ, e.g., 4 SQ entries in ARM Cortex-A53 [223]. To lower the pressure on the tiny GSQ caused by store verification, Turnpike compiler eliminates unnecessary stores, while its hardware early releases certain stores to the L1 data cache without holding them in the GSQ for verification. However, Turnpike incurs a high run-time overhead for out-of-order cores, despite its additional hardware support for the fast store release. This is because Turnpike compiler fails to form large regions for out-of-order cores—as with Turnstile compiler—and thus quickly overflows RBB , causing the core pipeline to stall frequently. Moreover, Turnpike's fast store release turns out to be unnecessary for two reasons: (1) stores are off-the-critical-path in out-of-order cores thanks to their large SQs, e.g., SQ size of ARM Cortex-A77 is 90 [85]; (2) their dynamic scheduling easily finds non-store instructions even if the SQ is full.

4.2 Design

What makes VeriPipe stand out from prior schemes [40, 214] is that it always verifies a region to be error-free at the end of the next region. As such, each stage of VeriPipe's *3-stage verification pipeline* is always occupied by at most one region (see Figure 4.1). This allows VeriPipe to track the region verification with minimal hardware cost, unlike the prior schemes that require expensive RBB whose overflow leads to significant core pipeline stalls.

To realize the low-cost *3-stage verification pipeline*, VeriPipe only introduces 3 registers, e.g., *GSQ Pointer*, *Region Register*, and *Recovery PC*, and 1 countdown timer, as shown in Figure 4.4. *Recovery PC* refers to the end of the latest verified region, i.e., whenever *Commit* stage gets a new region, it is pointed to by *Recovery PC* to serve as a recovery point. *Region Register* is a pointer referring to the last instruction of the region on the *Verify*; this is technically a region boundary instruction and thus points to the beginning of the next region that is currently on the *Execute*. *GSQ Pointer* refers to the tail of gated store queue (GSQ), indicating that all the following younger stores are not verified yet and thus must be squashed in case of a soft error. Finally, to track the remaining cycles for a region on the *Verify* to transit to *Commit*, the timer is reset to WCDL cycles at each region boundary and counts down each cycle thereafter.

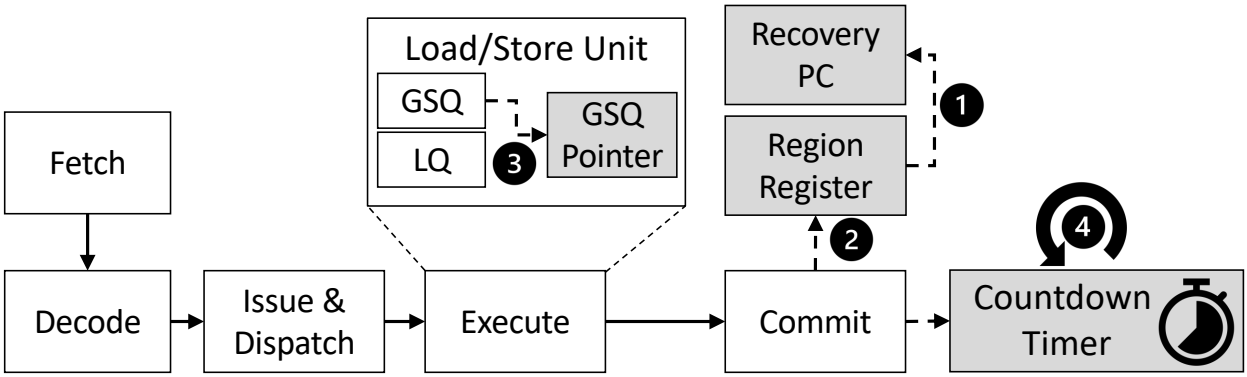


Figure 4.4. VeriPipe microarchitecture; shaded are newly added

When a region finishes its execution, i.e., the region boundary instruction is committed from the core pipeline, the prior region—whose end has been pointed to by *Region Register*—moves on to the *Commit*. VeriPipe then updates its registers and countdown timer accordingly as shown in Figure 4.4: (❶) updating *Recovery PC* with the current *Region Register*, (❷) resetting it to the address of the region boundary instruction, (❸) resetting *GSQ Pointer* to the current tail of the GSQ with the stores older than *GSQ Pointer* released to the L1 data cache, and (❹) resetting the countdown timer to WCDL cycles.

Figure 4.5 shows how the *3-stage verification pipeline* works using the three registers and the countdown timer. Suppose all regions *Rg*0-*Rg*2 are long enough to cover WCDL cycles.

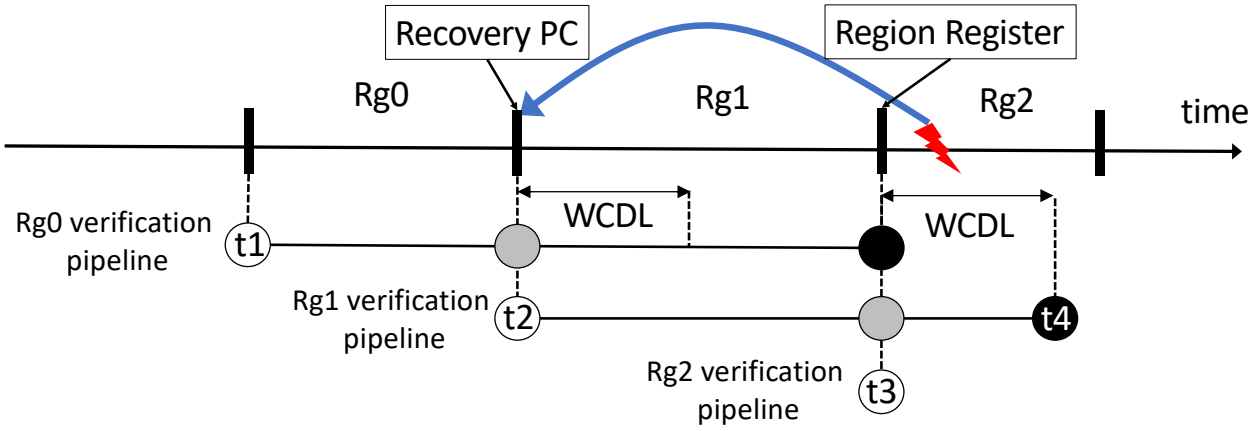


Figure 4.5. Verification pipeline status at time $t3$; $Rg0$ has been verified to be error-free; $Rg1$ on *Verify*, while $Rg2$ on *Execute*

At $t1$, $Rg0$ enters into the *Execute*, and VeriPipe resets the timer to WCDL cycles. At $t2$, $Rg0$ moves to the *Verify*, $Rg1$ starts with the *Execute*, and VeriPipe resets the timer again. While the core pipeline reaches the end of $Rg1$ at $t3$, $Rg0$ reaches the *Commit*. Accordingly, VeriPipe updates *Recovery PC* with the end of $Rg0$ and sets *Region Register* to the end of $Rg1$. After resetting the timer, VeriPipe starts to verify $Rg1$. Once a soft error (ϟ) is detected in $Rg2$, VeriPipe consults *Recovery PC* to resume the program execution from $Rg0$'s end.

4.3 VeriPipe Compiler

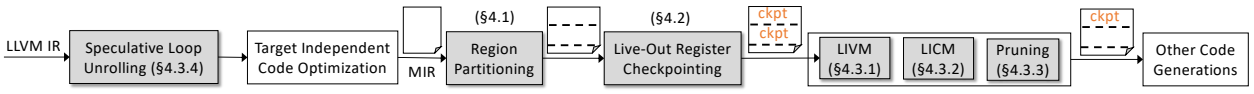


Figure 4.6. VeriPipe compiler workflow; shaded passes are newly proposed

This section illustrates how VeriPipe compiler forms a series of verifiable/recoverable regions as shown in Figure 4.6.

4.3.1 Region Partitioning

VeriPipe—akin to prior work [214]—leverages the gated store queue (GSQ) as a redo buffer to hold the data being stored for verification. To circumvent GSQ overflow, which would otherwise cause incorrect error recovery, VeriPipe compiler partitions input program into a series of regions at LLVM IR level with half of the GSQ size in mind. Thus, the GSQ never overflows when a region is being executed while a prior region is on the *Verify*. VeriPipe compiler first partitions program at callsites and loop headers. Specifically, it inserts a region boundary at all entry and exit points of functions. To avoid GSQ overflow during the execution of loops, VeriPipe compiler also inserts a region boundary in the loop headers, i.e., each loop iteration forms a region. Starting with these initial boundaries, VeriPipe compiler counts the stores while traversing the control flow graph (CFG) of the input function; it picks the maximum of store counts from multiple paths at joint points. When the count reaches the threshold—i.e., half of GSQ size, a region boundary is inserted and serves as a recovery point in case the following region gets interrupted by soft errors.

4.3.2 Live-Out Register Checkpointing

However, the GSQ only verifies memory data, leaving register values unverified. To address this issue, VeriPipe turns register verification into memory verification by checkpointing registers to the memory at LLVM Machine IR level; such register checkpointing is scheduled after instruction selection and register allocation. First, VeriPipe compiler identifies *live-out* registers in each region using liveness analysis [136]; these registers are used as inputs of subsequent regions. Later, VeriPipe compiler checkpoints *live-out* registers of each region to the memory by inserting stores after their most recent definitions in the region; VeriPipe still ensures that none of regions has more stores than the partitioning threshold. In particular, a region's checkpoints will be used to recover a subsequent region in case of soft error detected. As such, soft errors occurred in a region do not compromise the region's error recovery since its recovery uses the checkpoints in the prior regions that must be already verified before proceeding to the error-interrupted region.

4.3.3 Checkpoint Elimination

VeriPipe's register checkpointing inserts checkpoint stores that incur more run-time overhead. VeriPipe exploits four existing compiler optimizations to eliminate unnecessary checkpoints while still maintaining the soft error resilience guarantee.

Loop Induction Variable Merging (LIVM): Existing compilers use loop strength reduction (LSR) [136] to replace an expensive expression of induced induction variables, such as multiplication of computing array element's address, by a cheap addition of basic induction variables and constants. Figure 4.7 (b) shows that the calculation of an array element's address is replaced by the addition ($r1 = r1 + 4$) of a basic induction variable $r1$. However, LSR leads to more checkpoints as (1) it introduces more basic induction variables involving loop-carried dependence, e.g., $r0$ and $r1$ in Figure 4.7(b); (2) they are live-out to the next loop iteration (i.e., region) and thus must be checkpointed, e.g., $5c$ and $6c$ in the figure.

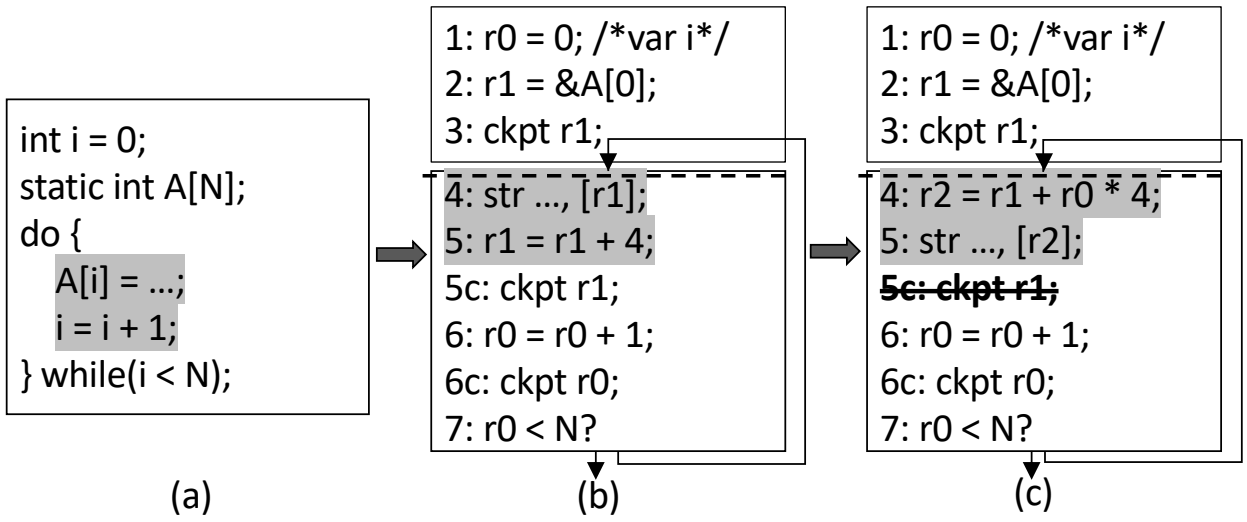


Figure 4.7. (a) original C code, (b) assembly code with LSR enabled, and (c) eliminating checkpoint $5c$ by LIVM

To eliminate the loop-carried dependencies for certain registers and their corresponding checkpoints, VeriPipe implements the same loop induction variable merging (LIVM) of prior work [214]. LIVM merges induced induction variables into the expressions of basic induction variables, resulting in only one checkpoint per induction variable chain. For example, Figure 4.7 (c) shows that $r2$ is now computed using basic induction variable $r0$ and constants.

This eliminates the checkpoint 5c. Consequently, LIVM brings a significant performance improvement for loop-intensive applications.

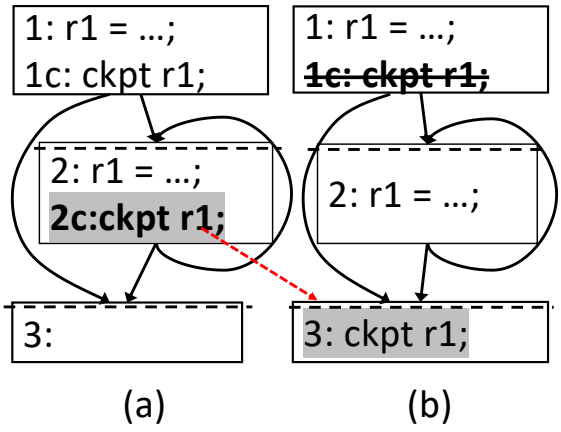


Figure 4.8. (a) original code, (b) moving checkpoint 2c out of loop with LICM

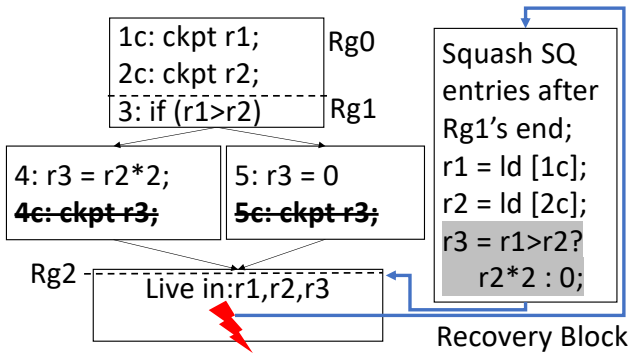


Figure 4.9. Eliminate checkpoint 4c and 5c with optimal pruning; recovery process on the right

Loop-Invariant Checkpoint Motion (LICM): For certain remaining checkpoints inside loops, VeriPipe employs the same loop-invariant checkpoint motion (LICM) as in Turnpike [214]. LICM can safely move checkpoints from within loops to their exit blocks, provided that the checkpointed registers are loop-invariant, i.e., no write-after-read (WAR)-dependence on them in the loops. Figure 4.8 shows that checkpoint 2c is moved out of the loop to the bottom basic block because $r1$ is loop-invariant. Moreover, with 2c moved to the bottom basic block, 1c in the top basic block becomes redundant and thus can be eliminated, enabling synergy further.

Optimal Checkpoint Pruning: To further reduce the run-time overhead caused by checkpoints, VeriPipe exploits the optimal checkpoint pruning—invented by [132]—to eliminate redundant checkpoints since they can be reconstructed using constants and/or other remaining checkpoints at recovery time. For example, Figure 4.9 shows that checkpoint 4c and 5c are eliminated. In the wake of soft error interruption, VeriPipe runtime recomputes register $r3$'s value using the checkpoint 1c/2c and immediate values in the recovery block and resumes the program execution from the beginning of $Rg2$. Consequently, VeriPipe shifts

checkpoint overhead from run time to the recovery time, significantly lowering the run-time overhead without jeopardizing the soft error resilience guarantee.

Speculative Loop Unrolling: Recall that VeriPipe compiler inserts a region boundary in all loop headers to avert GSQ overflow during loop execution. However, this often generates a lot of short regions given that small loops are prevalent in the evaluated benchmarks; Figure 4.10 shows that 50% of the loops in the evaluated applications have less than 30 instructions. Given that registers tend to be short-lived [224] and thus long regions likely have fewer live-out registers to be checkpointed, VeriPipe's region partitioning generates superfluous checkpoints for the short regions.

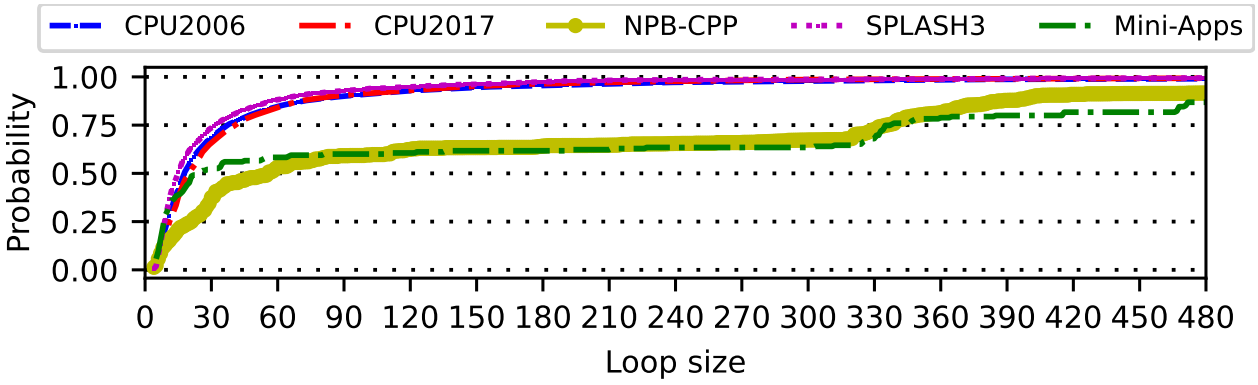


Figure 4.10. CDF of instruction count in loops

To address the above issue, VeriPipe applies the *speculative loop unrolling*—invented by Capri [60]—to enlarge loops no matter if their iteration counts are static-unknown; conventional loop unrolling only unrolls the loops with constant iteration counts[3]. While duplicating a loop body for a certain number of times, VeriPipe compiler inserts the code to check for proper loop termination after each duplicated loop body. To compute a proper unrolling factor, VeriPipe takes an optimistic approach that assumes each instruction finishes in one cycle on out-of-order cores with commit width `CW`. Thus, the unrolling factor is computed as $\frac{WCDL \times CW}{\text{Loop Size}}$ to balance code size with the size of unrolled loops. In particular, VeriPipe carefully tunes speculative loop unrolling to incur as little negative impact on the performance as possible.

[3]↑GCC can unroll some loops with static-unknown iteration counts if they are computed as expressions, while VeriPipe's unrolling is generic to any loops.

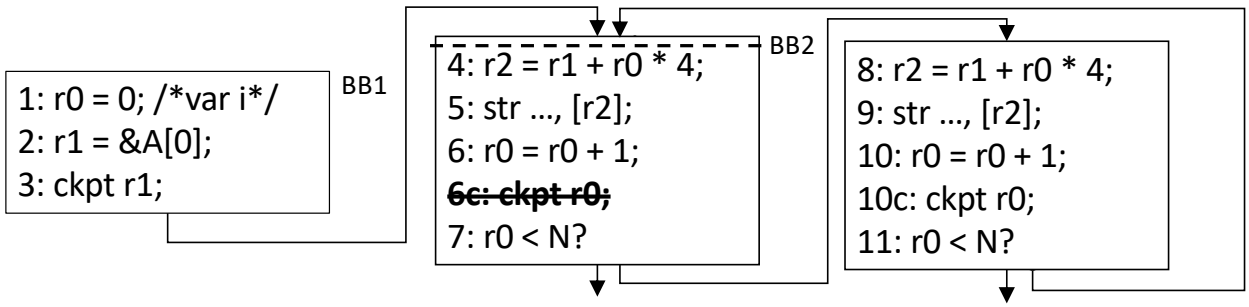


Figure 4.11. Reduce the dynamic instances of $r0$'s checkpoint by a factor of 1/2 via speculative loop unrolling

Once regions are enlarged, certain registers become no longer live-out anymore, rendering their checkpoints redundant. For example, suppose WCDL is 10 cycles while CW is 1 on an out-of-order core, Figure 4.11 shows that VeriPipe unrolls the loop in Figure 4.7(c) by a factor of 2 [225]. That way, checkpoint 6c becomes unnecessary and can be eliminated since $r0$ defined at line 6 is not live-out anymore. Consequently, VeriPipe reduces the number of dynamically executed checkpoints by 2x.

4.4 VeriPipe Hardware

Recall that VeriPipe compiler inserts a region boundary at all entry and exit points of functions, this generates a lot of short regions for small functions if they are recursive and called inside loops. Even worse, in certain evaluated applications, e.g., `povray` and `leela`, many small recursive functions cannot be transformed to be non-recursive by tail call elimination [136] for inlining. Here, the problem is that the presence of short regions causes incorrect region verification. This is because at the end of a short region, WCDL cycles have not passed since the end of the prior region that is on the *Verify*. Therefore, moving the prior region to the *Commit* could release potentially corrupted data to the L1 data cache, resulting in detected-but-uncorrectable errors (DUEs).

Naive Dynamic Enforcement: To address the above issue, one naive way is artificially extending the execution time of the short region to be WCDL cycles. That is, the core pipeline stalls at the end of each short region until the countdown timer hits zero, i.e.,

Algorithm 1 Verification Controller

Require: Countdown Timer T
Require: Instruction I
  if $T == 0$ **then**
    for each $Str \in$ GSQ[$start\_index$, $GSQ\ Pointer$] **do**
      merge Str to L1 data cache
    end for
    if I is a region boundary instruction **then** ▷ Performing the following 4 steps simultaneously at circuit level
      $T \leftarrow WCDL$
      $Recovery\ PC \leftarrow Region\ Register$
      $Region\ Register \leftarrow I's\ PC$
      $GSQ\ Pointer \leftarrow GSQ\ tail$
    end if
  else if I is a region boundary instruction **then**
    Treat I as a noop
  end if

WCDL cycles have passed since the end of the prior region that is on the *Verify*. However, the naive approach incurs a significant run-time overhead due to frequent core pipeline stalls occurred at the end of short regions, which becomes even worse for longer WCDLs; Section 4.7.2 shows that this strategy incurs an average of 8% and up to 50% run-time overhead.

4.4.1 Region Stitching

We find out it is unnecessary to stall the core pipeline at the end of a short region as long as the stores of the prior region are held in the GSQ for WCDL cycles. In light of this observation, VeriPipe proposes a simple hardware technique called *region stitching* that dynamically combines a short region with the following regions while holding the stores of the prior region in the GSQ. VeriPipe continues this process until the countdown timer hits zero at a region boundary, i.e., the stitched region is now long enough to cover WCDL. We can even relax this for higher performance. When the timer becomes zero, the prior region is immediately verified to be error-free without waiting for the next region boundary. This early releases the region's stores from the GSQ to the L1 data cache. *The beauty of*

region stitching is that it does not incur any storage overhead and scales up to arbitrarily long WCDL.

Algorithm 1 describes VeriPipe's actions upon committing an instruction, which accepts two inputs: the countdown timer T and the instruction I. If T hits zero (at line 1), VeriPipe early merges the stores older than *GSQ Pointer* to the L1 data cache since they are already verified to be error-free. This relieves the pressure on the GSQ while still maintaining soft error resilience guarantee. If I is a region boundary instruction, VeriPipe updates its three registers accordingly and resets the timer as shown in the algorithm from line 6 to 9. When the timer is not zero and the input instruction is a region boundary, VeriPipe treats the instruction as a noop, doing nothing special. In particular, region stitching still verifies inserted checkpoints, though it might render some checkpoints not live-out and thus unnecessary.

Most important, region stitching still works correctly for synchronization primitives, e.g., atomic operations and memory fences, which force all prior stores to be merged into the L1 data cache before committing them. This is because VeriPipe treats the primitives as region boundaries during region formation (see Section 4.3.1). Therefore, stores prior to the synchronization primitives are held in the GSQ for verification until the regions ending at the primitives reach the *Commit*. After that, the stores of the regions are released to the L1 data cache, i.e., they become visible to other cores, allowing the ROB to commit the primitives. Consequently, other cores competing for the primitives can start their execution without observing any unverified data.

4.5 Recovery Protocol

To resume program from soft errors, VeriPipe performs three actions in a row: (1) discarding the stores younger than *GSQ Pointer* from the GSQ, (2) restoring live-in registers of the oldest unverified region in a compiler-generated per-region recovery block (see Figure 4.9), and (3) resuming the program execution from the region's beginning. Figure 4.12 shows how VeriPipe recovers the program with region stitching enabled. Assume $Rg0$-$Rg1$ and $Rg5$-$Rg6$ are long enough to cover WCDL, while $Rg2 - Rg4$ are not.

To start with, $Rg0$ is on the *Commit* (i.e., it is error-free), and $Rg1$ is on the *Verify*, as shown in Figure 4.12 (a). Upon reaching the end of $Rg2$—on the *Execute*, VeriPipe combines $Rg2$, $Rg3$, and $Rg4$ into a single region referred to as $Rg234$ hereafter to ensure that the total execution time of $Rg234$ meets or exceeds the required WCDL cycles. When a soft error is detected in either $Rg2$ or $Rg3$, it is safe to resume the program from $Rg1$'s beginning pointed to by *Recovery PC*. This is because $Rg1$'s live-in registers are already verified to be error-free in prior regions; all stores in $Rg1 - Rg4$ are squashed from the GSQ and thus do not affect the memory states.

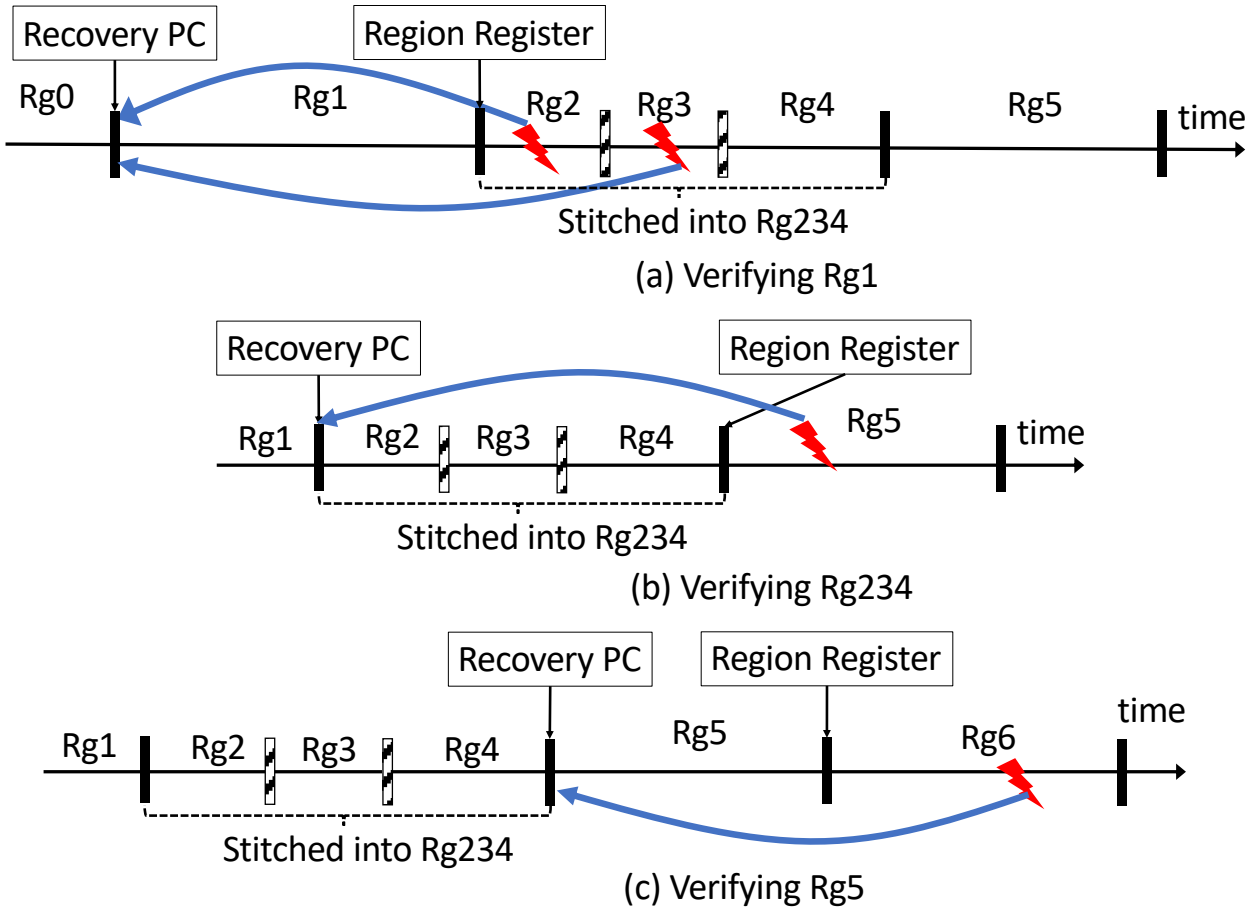


Figure 4.12. VeriPipe recovery example with region stitching

When reaching the end of $Rg234$, $Rg1$ moves to the *Commit* and is verified to be error-free, $Rg234$ enters into the *Verify*, and $Rg5$ starts with the *Execute*. Here, *Recovery PC* is updated with $Rg1$'s end, while *Region Register* points to $Rg234$'s end, as shown in Figure

4.12 (b). That way, upon a soft error detected in $Rg5$, VeriPipe guarantees the program to be recoverable from the beginning of $Rg234$. The reason is three-fold: (1) VeriPipe only uses the live-in registers of $Rg234$ to recover the program; (2) all of them are live-out from the prior verified regions and must already be error-free; (3) region stitching only makes certain checkpoints in $Rg2 - Rg4$ redundant, which do not affect $Rg234$'s live-in registers.

Eventually, $Rg234$ is verified and transits to the *Commit* at the end of $Rg5$. $Rg5$ then moves to the *Verify*, and $Rg6$ starts with the *Execute*; VeriPipe updates its three registers and the countdown timer accordingly. Upon a soft error detected in $Rg6$, as shown in Figure 4.12 (c), VeriPipe runtime can successfully resume the program execution from $Rg5$'s beginning—it is also the end of the verified region $Rg234$. The reason is twofold: (1) $Rg5$'s live-in registers are defined in prior regions and thus already be verified to be error-free; (2) region stitching still reserves the checkpoints in $Rg2$-$Rg4$ for recovering $Rg5$.

4.6 Discussion

4.6.1 Fault Model

VeriPipe assumes that the memory system (i.e., caches and DRAM) is already protected with error-correcting code (ECC) as in commodity processors [226]. Also, VeriPipe assumes that its proposed hardware structures, e.g., three registers and one timer, and store queue are hardened against soft errors as in prior designs [84, 214].

4.6.2 Why No Fault Injection?

As stated in [2, 227], soft errors are predominantly generated by energetic particle strikes that always generate a sound wave. Due to the physical phenomena, the sound wave must be detected by deployed acoustic sensors. Consequently, soft errors are sure to be detected [84] within a bounded latency no matter how many soft errors occur simultaneously, leading to never missed soft errors. Thus, the 100% guaranteed detection of particle-induced soft errors obviates the need for fault injections. Thanks to the near-zero overhead, VeriPipe can be integrated with other techniques [28] to achieve a full protection against all kinds of soft errors.

4.6.3 False Positive Rate

As prior work [84] shows, with calibration, acoustic sensors can avoid the detection of those particle strikes which do not generate bit flips, thus reducing the chance of reporting such weak strikes to zero. Nevertheless, false-positive case still occurs since not every bit flip causes a program failure, e.g., incorrect program output, program crash, and program hang, because of the so-called bit-masking effect [228]. If acoustic sensors do not trigger the detection of weak particle strikes, the false positive rate becomes the same as bit masking rate. According to prior studies [229, 230], the bit masking rate of soft errors is ≈90% for CPU applications, and Gupta et al. [230] found the post-masking failure rate is typically 0.9 error per day. Given all this, the pre-masking error rate per day is $\frac{0.9}{1-0.9} = 9.0$. Therefore, acoustic sensors are expected to report $9.0 \times 0.9 = 8.1$ errors per day. The implication is that VeriPipe runtime re-executes a region to correct a soft error occurred therein every ≈3 hours, in which case the overhead is negligible considering that the average region execution time is 47.63 ns (see Section 4.7.5).

4.6.4 Error Recovery for Multi-Cores

To ensure program recovery for multi-cores, VeriPipe assumes data-race-freedom (DRF) program which is guaranteed by C/C++ 11 standard [231] onwards. As with prior proposals [40, 214], VeriPipe treats synchronization primitives, e.g., atomic operations and memory fences, as region boundaries such that critical sections form regions. The implication is three-fold: (1) the stores of critical sections are released to the memory and thus visible to other cores only after their verification; (2) upon detecting soft errors, there must be at most one core in a critical section since other cores have not obtained the lock of the section; (3) in the case of soft error detected, the cores can be independently rollbacked to their latest verified points without the need of tracking inter-thread dependence.

4.6.5 Exception and Interrupt

Upon detecting a soft error while receiving an exception/interrupt signal, VeriPipe resumes the program execution from the end of the latest verified region and continues the execution till the program point where the exception took place, After that, VeriPipe invokes the corresponding exception handler to deal with the specified exception as a regular processor does. Notably, VeriPipe has a minimal impact on the performance of exception handling since the chance of encountering both soft error and exception at the same time is practically negligible. Even if this case occurs, VeriPipe delays the exception handling by only 47.63 *ns* on average.

4.7 Evaluation

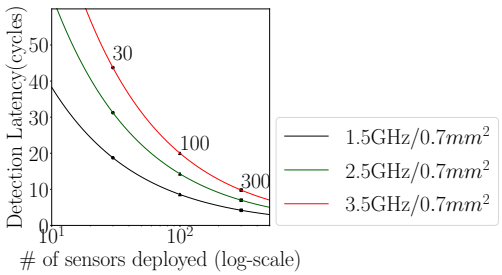


Figure 4.13. Detection latency across the number of sensors deployed

We implement our compiler optimizations atop Clang/LLVM 13 compiler [139] with about 2300 lines of code. All evaluated C/C++ applications are compiled with -O3 flag and statically linked. We implement our hardware design using gem5 [204] simulator to model an eight-core *out-of-order* ARMv8 Cortex-A77 processor [85] clocked at 2.42GHz. Each core has 256 reorder buffer (ROB) entries, 85 load queue (LQ) entries, 90 store queue (SQ) entries, 160 instruction queue (IQ) entries, and 256 physical registers. Also, each core is equipped with a 64KB 4-way private L1 data cache with 4-cycle hit latency and a 512KB 8-way private L2 cache with 9-cycle hit latency. All the eight cores share a 4MB 16-way L3 cache with 31-cycle hit latency. The main memory is configured to 16GB DDR4 2400 8x8.

We treat the original program running on the original hardware platform without soft error resilience as our baseline.

We run multi-threaded benchmarks, e.g., SPLASH3 [217] and NPB-CPP [216], on gem5 full system (FS) mode, while simulating SPEC2006/2017 [140, 141]/Mini-Apps [218, 219] on system call emulation (SE) mode. We synchronize the number of simulated instructions by measuring the number of function calls in the baseline which is a constant across different binary version generated by various compiler optimizations. As with prior techniques [40, 214, 232], all SPEC/Mini-Apps applications are chosen to be fast forwarded 5 billion instructions with an atomic CPU, and then are simulated for 1 billion instructions in O3 CPU model. The FS simulation boots an Ubuntu 14.04 with Linux kernel 4.18.0 and runs SPLASH3/NPB-CPP with eight cores by default.

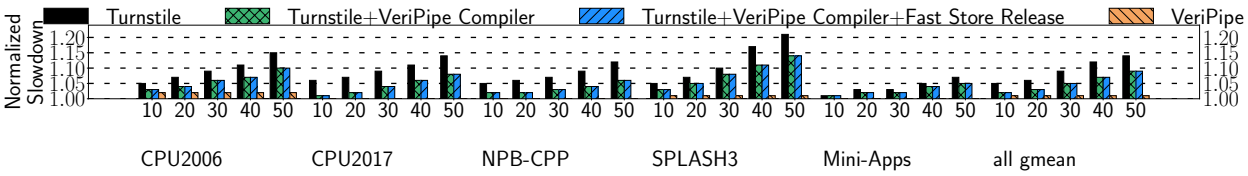


Figure 4.14. Run-time overhead comparison between VeriPipe and prior approaches with varying WCDL (default to 30 cycle); lower is better

As WCDL is affected by the number of sensors deployed, we calculate the number of desired sensors for the given WCDL cycles as with prior work [40, 214]. Figure 4.13 shows that deploying 30-300 sensors achieves 50-10 cycles of WCDL on an ARM Cortex-A77 core—0.7 mm^2 core size excluding caches with TSMC $7nm$ technology—with varying clock frequency. With this in mind, we conservatively set the default WCDL to 30 cycles.

4.7.1 Run-time Overhead Analysis

To demonstrate the high performance of VeriPipe, we compare VeriPipe to the state-of-the-art work **Turnstile** [40] across a variety of WCDLs. We also apply VeriPipe compiler optimizations to Turnstile, forming a scheme called **Turnstile+VeriPipe Compiler**. We further implement the fast store release of Turnpike [214]—which proposes a hardware-based

fast store release to bypass store verification and thus relieves the pressure on SQ, represented as **Turnstile+VeriPipe Compiler+Fast Store Release**.

As shown in Figure 4.14, VeriPipe is greatly superior to all prior techniques across all WCDLs from 10 to 50 cycles. VeriPipe incurs an average of only 1% run-time overhead for all the WCDLs, while Turnstile incurs an average of 5% to 14% and up to a 62% run-time overhead for the varying WCDLs. Here, **Turnstile+VeriPipe Compiler** still results in an unacceptable overhead for certain applications, e.g., 61% for `povray`, 28% for `fft`, and 33% for `lu-contig`, though our compiler optimizations can improve the performance of Turnstile owing to eliminating redundant checkpoints. The reason is twofold: (1) Turnstile cannot tolerate small regions due to limited region boundary buffer (RBB)—20 entries in our configuration; and (2) VeriPipe compiler fails to enlarge these small regions (see Section 4.4 for the discussion in details). As a result, Turnstile cannot scale up to longer WCDLs no matter if enabling VeriPipe compiler optimizations. Noteworthily, **Turnstile+VeriPipe Compiler+Fast Store Release** still does not help in improving the performance of **Turnstile+VeriPipe Compiler** at all, despite bypassing the verification for certain stores. This is because (1) out-of-order cores are equipped with 10x larger store queue (SQ) than in-order cores, and thus (2) holding stores in the SQ for verification has no extra pressure on the SQ as confirmed in Section 4.7.6.

4.7.2 Impact of VeriPipe's Optimizations

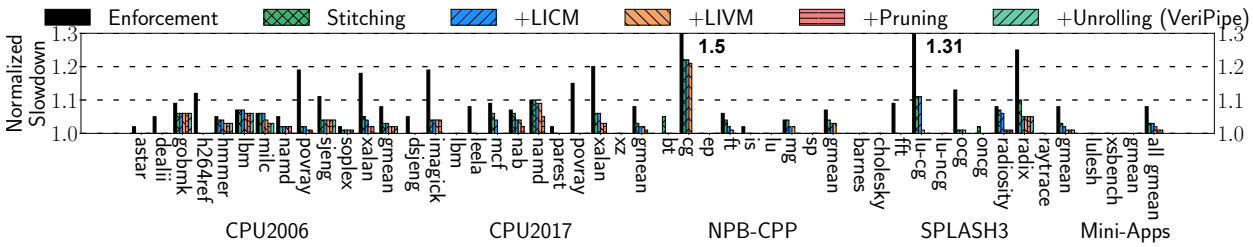


Figure 4.15. Impact of each VeriPipe optimization; lower is better

To investigate the effect of each VeriPipe optimization, we conduct a series of experiments with the optimizations enabled incrementally and present the results in Figure 4.15.

Enforcement: shows the run-time overhead of enabling the naive dynamic enforcement. The figure shows that this naive strategy causes a significant run-time overhead, e.g., 8% on average and up to 50%, due to pipeline stalls incurred at the end of each short region.

Stitching: stands for the run-time overhead of enabling region stitching. As the figure shows, region stitching significantly reduces the run-time overhead compared to the naive enforcement. On average, region stitching incurs 3% run-time overhead. In particular, region stitching lowers the overhead of many applications, e.g., `h264ref`, `povray`, `leela`, `fft`, and `ocean-contig`, to near-zero.

+LICM: shows the run-time overhead after incrementally enabling the LICM. It further lowers the run-time overheads of certain applications, e.g., 1% reduction for `xalancbmk`, 2% reduction for `mcf` and `nab`, and 5% reduction for `radix`.

+LIVM: depicts that the LIVM lowers the average run-time overhead to only 2%. In particular, the LIVM lowers the overheads of certain applications to near zero, e.g., `mcf`, `lu-config`, and `radiosity`, thanks to its ability to move checkpoints out of loops.

+Pruning: indicates the run-time overhead of enabling the optimal pruning further. The figure shows that the pruning reduces the average overhead to only 1% as it eliminates redundant checkpoint stores. Note that the pruning lowers the overheads of many applications to near zero, e.g., `cg`, `mg`, `lu-config`, and `ocean-contig`.

+Unrolling (VeriPipe): stands for the overall run-time overhead VeriPipe incurs with all optimizations enabled. Eventually, VeriPipe incurs only 1% run-time overhead on average for total 43 applications. The figure shows that the unrolling can further reduce the overheads of some applications, e.g., 2% reduction for `namd`, as it can avoid certain checkpoints of enlarged regions.

4.7.3 Effect of Region Stitching

To investigate the effectiveness of region stitching in eliminating region boundaries, we compute the ratio of the regions removed by the region stitching to total amount of regions. Figure 4.16 shows that the region stitching eliminates 49% of regions. This explains why the

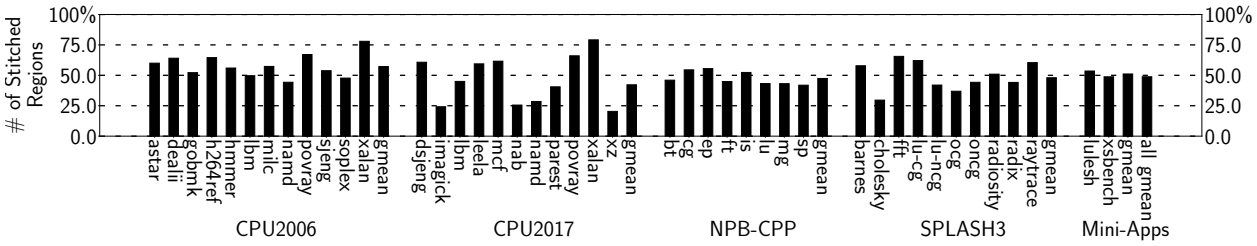


Figure 4.16. The ratio of the region eliminated by the region stitching to total regions; higher is better

region stitching is so good at obviating pipeline stalls occurred at the end of short regions and lowering the run-time overhead.

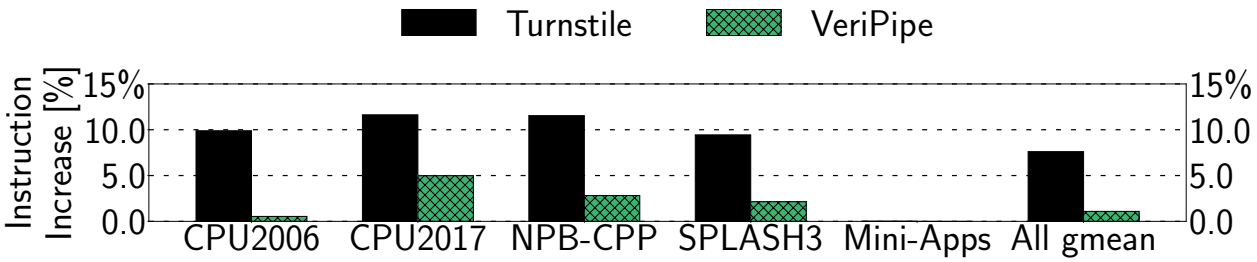


Figure 4.17. Normalized dynamic instruction increases of Turnstile and VeriPipe to the baseline; lower is better

4.7.4 Dynamic Instruction Increase

To inspect how effective VeriPipe's compiler optimizations are in eliminating redundant checkpoints, we collect the number of committed instructions of Turnstile and VeriPipe. Figure 4.17 shows that VeriPipe leads to an average of only 1.09% increase in committed instructions, while Turnstile incurs an average of 7.61% increase. Consequently, VeriPipe incurs minimal pressure on the instruction cache of modern server-class processors where the pressure has been becoming a concern [233].

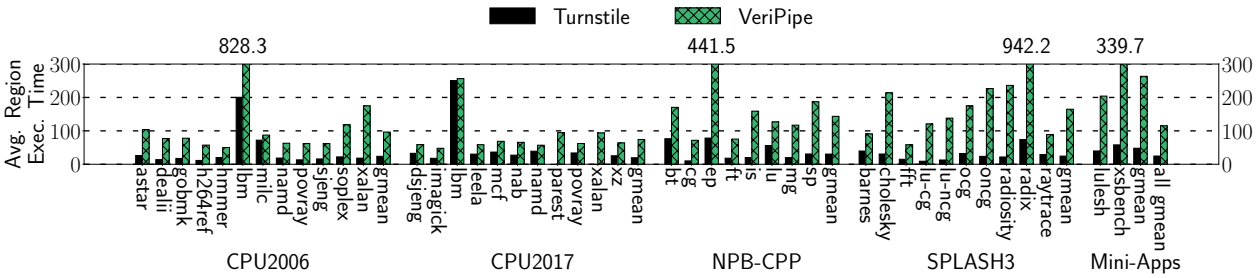


Figure 4.18. Average region execution time in cycles; higher is better for less CPU pipeline stalls at a region boundary

4.7.5 Region Characteristics

To investigate why the state-of-the-art work Turnstile causes significant pipeline stalls at the end of regions, while VeriPipe incurs near-zero pipeline stalls at region end. We demonstrate the average region execution time of Turnstile and VeriPipe in Figure 4.18. We compute the region execution time by subtracting the commit time of the region's first instruction from that of the last instruction. Then, we average the execution time of all regions for each application. The figure shows that Turnstile's average region execution time is only 24.63 cycles—implying that Turnstile wastes many CPU cycles at the end of short regions, while VeriPipe's is 115.26 cycles thanks to the synergistic compiler/architecture co-design. Notably, owing to the region stitching, we can enlarge VeriPipe's region size further for longer WCDLs with no overhead incurred.

4.7.6 Sensitivity Analysis

Sensitivity to WCDL: To clearly show how WCDL affects VeriPipe's run-time performance, we test VeriPipe for varying WCDLs from 10 to 50 cycles as shown in Figure 4.19. The trend is that VeriPipe incurs the same run-time overhead for all evaluated applications no matter how long the WCDL is owing to the simple yet effective region stitching. This implies that VeriPipe can significantly reduce the number of deployed acoustic sensors—lowering hardware complexity further—with no performance impact.

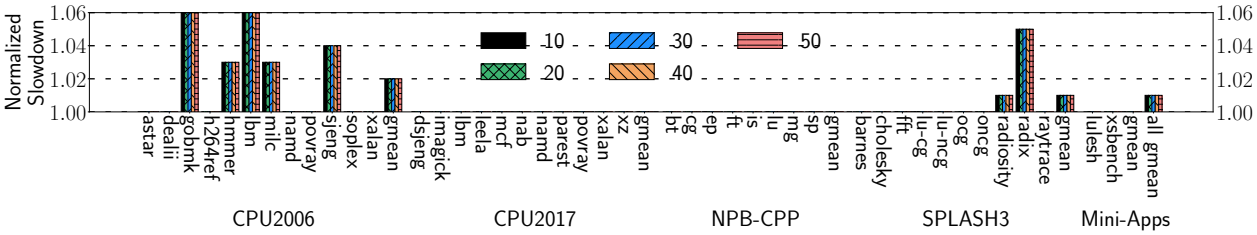


Figure 4.19. Normalized slowdown of VeriPipe with varying WCDL (default to 30); lower is better

Sensitivity to Store Queue Size: You might think that buffering the data being stored in the store queue (SQ) for verification imposes extra pressure on the SQ. To see how this affects the run-time performance of VeriPipe, we vary the SQ size from 56 up to 110, which represent the SQ sizes of four typical high-performance out-of-order cores, e.g., Marvell ThunderX3 [234], ARM Cortex-A76 [235], ARM Cortex-A77 [85], and Apple M1 [236]. Figure 4.20 shows that VeriPipe leads to no increase in the run-time overhead. This is because VeriPipe puts minimal pressure on the SQ owing to its compiler optimizations, allowing VeriPipe to be integrated into varying computing devices ranging from mobile platforms to datacenters.

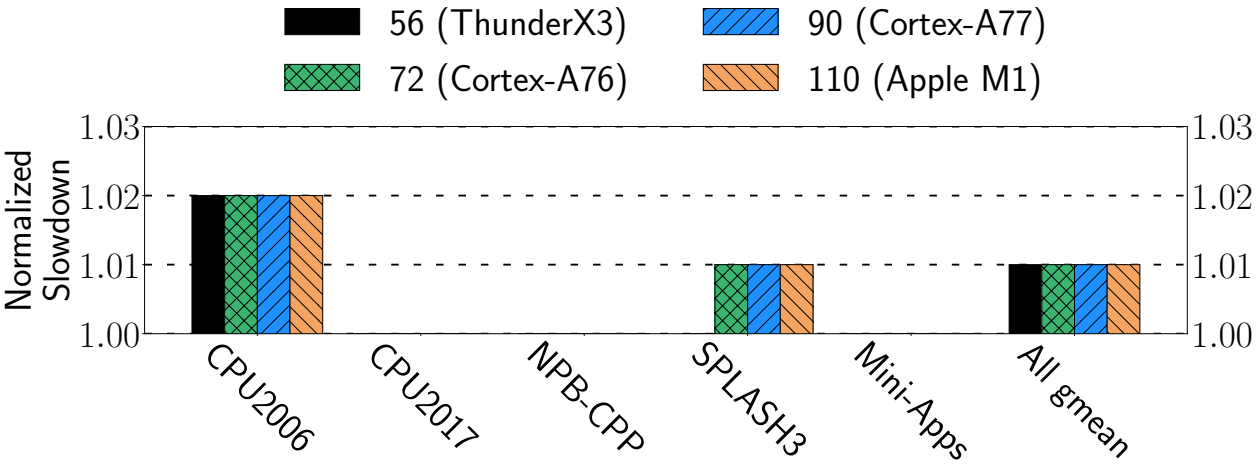


Figure 4.20. Normalized slowdown of VeriPipe to the baseline with varying SQ size from 56 up to 110; lower is better

Sensitivity to Thread Count: As with prior techniques [40, 214, 237], VeriPipe treats synchronization primitives as region boundaries for correct multi-cores recovery. This might

delay the inter-thread synchronization due to adding verification latency to the execution delay of the synchronization primitives. To investigate such an impact, we vary the number of threads for NPB and SPLASH3 from 8 up to 64. As shown in Figure 4.21, VeriPipe practically has negligible (1%) impact on the performance of these program for all configurations. The reason is twofold: (1) critical sections account for a small portion of the program execution time; (2) VeriPipe incurs minimal stall cycles at the end of critical sections (regions) thanks to long enough regions.

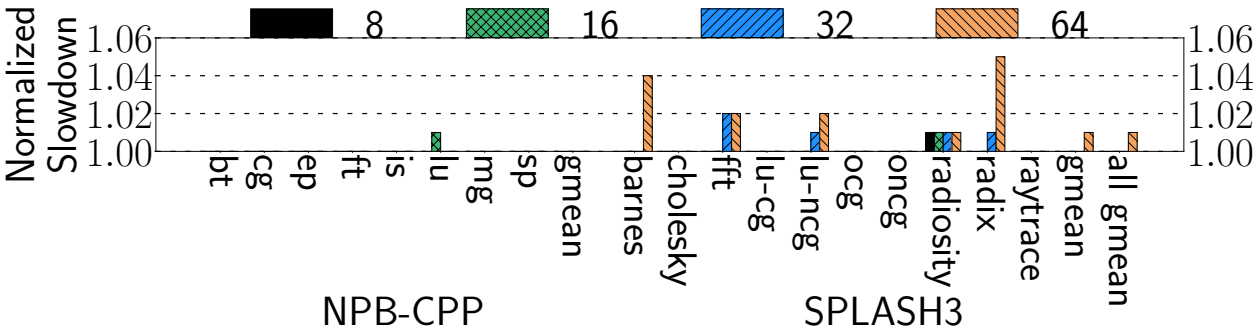


Figure 4.21. Normalized slowdown of VeriPipe with varying thread count from 8 to 64; lower is better

4.7.7 Power and Area Overheads

VeriPipe proposes only two 64-bit registers (*Recovery PC* and *Region Register*), a 7-bit register (*GSQ Pointer*), a 5-bit countdown timer, and a simple control logic comprised of a few comparators. We implement the hardware components of Turnstile/VeriPipe using Chisel [238] and compile the code into RTL with TSMC 28nm—the open-source RTL compiler we get only supports 28nm. Table 4.1 shows that VeriPipe incurs 8.8% area, 8.7% power consumption, and 26.9% access latency of Turnstile.

Due to the lack of open-source RTL compiler that supports TSMC 7nm node, we scale the synthesis results of the 28nm down by 6.6x [239, 240] to approximate the results of TSMC 7nm technology. It turns out that VeriPipe occupies only 0.018% area of a 0.7mm^2 ARM Cortex-A77 core (excluding caches).

Table 4.1. Hardware cost comparison between Turnstile and VeriPipe with TSMC 28nm technology

	Area (μm^2)	Power (mW)	Max. Access latency (ns)
VeriPipe	849.1	2.4	0.07
Turnstile	9667.3	27.5	0.26
VeriPipe / Turnstile	8.8%	8.7%	26.9%

4.8 Other Related Work

Many prior techniques [28–34] propose to leverage instruction-level/thread-level/process-level duplication to detect the occurrences of soft errors, ending up with high run-time overhead. Although hardware-based techniques [154, 241, 242] can lower run-time overhead, they come with expensive hardware costs. Other schemes [38, 39] detect the error occurrence by checking for the symptoms that soft errors generate, while dual/triple modular redundancy (DMR/TMR) schemes [36, 37] check for faulty consequences. However, they all suffer from lower detection coverage. Prior recovery schemes [59, 116, 134, 237, 243–251] cause high run-time overhead regardless of their recovery granularity. Recently, Kim et al. proposed Penny [132] to provide high-performance soft error resilience for GPUs. Penny makes use of parity code for immediate soft error detection and idempotent processing [42, 57, 116, 133] for error recovery. Flame [215] leverages acoustic sensors and idempotent processing for GPU soft error resilience. It exploits the massive multi-threading of GPUs to hide the verification latency of warps, achieving lightweight resilience.

4.9 Summary

This chapter introduces VeriPipe, a near-zero-overhead resilience scheme that protects out-of-order cores against soft errors with acoustic-sensor-based detection. VeriPipe compiler partitions input program into a series of recoverable regions, while VeriPipe hardware verifies whether they are error-free at run time. For the verification purpose, VeriPipe delays the writeback of any data stored during region execution until the region is verified to be error-free. If an error is detected, VeriPipe corrects it by resuming the program from the end

of the latest verified (error-free) region. The experiments with 43 applications of SPEC 2006/2017/NPB-CPP/SPLASH3/Mini-Apps demonstrate that VeriPipe incurs only a 1% run-time overhead, on average. In particular, VeriPipe achieves such high-performance soft error resilience at minimal hardware complexity (3 registers and 1 countdown timer), unlike the state-of-the-art work that requires intrusive microarchitectural modifications and yet causes a significant run-time overhead. We believe that VeriPipe lays the foundation for the commercialization of acoustic-sensor-based soft error protection.

5. PPA: PERSISTENT PROCESSOR ARCHITECTURE

Nonvolatile memory (NVM) technologies such as ReRAM [252, 253], 3D XPoint [254], PCM [255–258], and STT-MRAM [259–263] have emerged as alternatives to DRAM. Thanks to their byte-addressability, high areal density, and in-memory persistence, they are to be used as nonvolatile main memory (NVMM)—also known as persistent memory (PMEM). That is, they can transparently replace DRAM to accommodate persistent applications with large memory footprint and obviate the need for serializing data in a block device to survive power failure.

However, it is not easy to make this obvious use case (i.e., transparent NVMM) in reality. For example, while Intel Optane persistent memory (PMEM) [107, 264–267] provides the transparent use of PMEM called *memory mode* where DRAM is used as the last-level cache atop PMEM, *the Optane manual states that the PMEM works as volatile memory [108]. The Optane persistent memory is not persistent at all*; this is mainly due to the difficulty of maintaining crash consistency in the memory mode[1]. As a result, under the memory mode, users have no choice but to risk the loss of all PMEM data in case of power failure.

Although PMEM offers *app-direct mode* where DRAM is used as main memory and PMEM serves as persistent heap [108], it pawns off the hard work of persistent programming on users, trading the transparency for in-memory persistence. In this partial-system persistence (PSP) model [18–25], users must delineate a part of code that requires persistence, rewrite the data structures used therein with crash consistency and memory persistency [268] in mind, and often devise application-specific recovery code tailored to the data structures [269–274]. Besides, PSP requires dedicated PMEM allocation interfaces such as `pmalloc` [275], rendering already error-prone persistent programming more complex [276–282]. While using transactions [19, 55, 58, 283, 284] or failure-atomic sections [18, 57, 285, 286] mitigates the programming complexity, the resulting persistent program is slower than the original one due to the undo/redo logging involving persistence barrier (`clwb` and `sfence` for x86).

[1]↑Since PMEM here is transparently used as main memory without any code change, it is solely the architecture's responsibility to flush data through the deep cache hierarchy (L1∼DRAM caches) and keep PMEM states consistent across power failure for correct recovery.

Given the limitations of PSP and the demand for transparent use of PMEM without sacrificing the in-memory persistence and crash consistency, there is an increasing interest in whole-system persistence (WSP) [60, 110] which covers all sorts of applications—rather than being limited to a small set of PSP application domains such as in-memory index structures/databases and key-value stores. That is, *WSP is agnostic to program semantics yet capable of recovering any kind of program from power failure no matter when it occurs!*

One naive approach to WSP is flushing all volatile states (register files, SRAM caches, and DRAM cache) to PMEM when power is about to be cut off. For example, Narayanan et al. [110] propose to use residual energy in uninterruptable power supply (UPS) and persist all volatile data before impending power failure, which requires a considerable amount of energy to be secured for flushing. In a similar vein, Intel's extended asynchronous DRAM refreshing (eADR) flushes the entire cache contents to PMEM upon power failure using a backup battery. However, eADR also leads to significant energy cost requiring a bulky supercapacitor of 3400 mm^3 [287]; this situation gets even worse for a deeper cache hierarchy that is driven by ever-increasing working sets of data-intensive applications [288, 289]. Apart from the inability to persist other volatile states such as registers, eADR cannot guarantee crash consistency for PMEM's memory mode—as it is unaffordable to reserve a sufficient amount of energy for flushing the data of large DRAM cache to PMEM; typical servers in data centers are equipped with more than 1TB DRAM. Given all this, it has been practically impossible to achieve WSP on the cheap.

To this end, this chapter presents *Persistent Processor architecture* (PPA), the first of its kind to realize transparent, lightweight, and performant WSP without recompilation for all program embracing legacy software whose source code is unavailable. We found that crash inconsistency is caused by unpersisted stores left behind power failure and can be corrected by replaying (persisting) them in the wake of the power failure. Suppose the program commits 3 stores ($strA$; $strB$; $strC$) in a row, and due to cache replacement, the youngest $strC$ is persisted in PMEM before the older ones. Although this violates the program semantics if a power outage occurs while others are cached, it is possible to fix the inconsistency by replaying $strA$ and $strB$—unpersisted before the outage—when power comes back. We can even relax this for simple hardware implementation, i.e., rather than

tracking the (un)persistence of each individual store, PPA instead replays all 3 committed stores and resumes the interrupted program following the last committed instruction in the wake of the outage.

To achieve that, it is essential to preserve the registers of stores (for replay) and other committed instructions (for resumption of the interrupted program) across power failure. The implication is two-fold: (1) PPA should prevent store registers from being overwritten; this is so-called store-integrity [245]. (2) Both store registers and other committed instruction registers must be able to survive power outage, i.e., PPA should save the registers on the outage—using a tiny capacitor whose energy is six orders of magnitude smaller than what eADR requires—for the replay and the resumption in the wake of the outage.

In particular, PPA realizes the *store integrity* in the core microarchitecture at a low cost. The key insight is that the values of store registers are retained in the corresponding physical registers[2] until they are deallocated. For example, once the architectural register $r0$ of a store is renamed to a physical register $p0$, PPA can retrieve $r0$ by reading the value from $p0$ unless it is remapped and overwritten by another instruction. To preserve the physical registers to which architectural registers of stores are renamed, PPA proposes to delay the deallocation of the physical registers—though the reorder buffer (ROB) already commits the store instructions[3]. Recall that out-of-order cores have a lot more physical registers than architectural ones to minimize the stalls caused by the lack of physical registers [290]; a physical register file (PRF) tends to be underutilized most of the time since only a part of instructions in ROB (30% in our experiments), e.g., loads and ALU operations, define new registers. Prior work also observes this phenomenon, which leads to the advent of simultaneous multi-threading (SMT) [291–296], PRF bank switching [297], and physical register inlining [298]. The takeaway is that due to PRF underutilization, PPA can delay the deallocation of store registers with minimal run-time overhead.

Such register-renaming-based store integrity is a building block of PPA enabling *region-level persistence*, where store integrity is ensured within each region (epoch) [299] for crash

[2]↑In an out-of-order processor, architectural registers are renamed to physical registers via register renaming (see Section 5.1.1 for details).

[3]↑More precisely in the context of a unified PRF (Section 5.1.1), PPA does not deallocate the physical registers of stores even after ROB commits them redefining their architectural registers.

consistency as well as *lightweight yet performant WSP*. PPA dynamically delineates the regions, performing region-level persistence and physical register reclamation across their boundaries; whenever PRF runs out, PPA starts a new region (epoch) with a persist barrier, which ensures the committed stores of the prior region have already been written to PMEM and reclaims those physical registers mapped by the stores. To persist the stores of each region efficiently, PPA uses asynchronous writeback overlapping them with the execution of other instructions in the region as prior work [121, 300–304]. It turns out that the region size is long enough to fully hide the store persistence latency, thanks to the large PRF of modern out-of-order cores. If any region is interrupted by a power outage, PPA checkpoints minimal architectural states, e.g., a part of PRF and hardware structures related to register renaming [305]. In the wake of the outage, PPA restores those checkpointed states, replays the committed stores of the interrupted region, and resumes the program from the last commit point before the outage—rather than rolling back to the beginning of the interrupted region—for correct and efficient recovery.

To evaluate PPA, we test it with 41 applications from SPEC CPU2006/2017 [140, 141], SPLASH3 [217], STAMP [306], WHISPER [307], and DOE Mini-apps [218, 219]. The experimental results show that PPA incurs only an average of 2% run-time overhead compared to the baseline (running original applications on PMEM's memory mode lacking in-memory persistence and crash consistency support). In summary, PPA makes the following contributions:

- PPA is the first lightweight yet performant whole-system persistence that introduces minor modifications on the hardware, e.g., 2 registers and 1 queue, and only needs a tiny capacitor of 21.7 μJ, unlike eADR that requires a supercapacitor of 550mJ.
- PPA outperforms the complex state-of-the-art compiler and architecture co-design approach [60] in terms of all aspects, such as run-time performance, energy requirement, and hardware cost.
- PPA treats the underlying cache hierarchy as a black box, thus being suitable for current/future caches with an arbitrary depth of the hierarchy, e.g., CXL (Compute Express Link) based far persistent memory [308–312].

- PPA only incurs an average of 2% run-time overhead and 0.005% areal cost, which we believe paves the way to practical whole-system persistence for all, driving the revival of persistent memory production with its cost-effectiveness.

5.1 Background and Challenges

5.1.1 Register Renaming

Register renaming [305, 313–315], serving as a basis for PPA's store-integrity region formation, provides a way to eliminate false register dependence and thus enables more instruction-level parallelism (ILP). To efficiently rename architectural registers, out-of-order processors are equipped with a unified PRF as in Alpha 21264 [316], MIPS 10K [317], ARM Cortex A-series out-of-order cores [318], RISC-V SonicBOOM [319], and modern Intel processors from Pentium 4 onwards [320]. For renaming an instruction, the processor picks a register from a Free List (tracking free physical registers) and maintains such a mapping from architectural register to physical one in a register alias table (RAT), i.e., any data access to the architectural register is referred to the corresponding physical register by consulting the RAT. Once ROB retires the instruction, the processor puts the mapping to a commit rename table (CRT) for facilitating exception handling and debugging.

In particular, the physical register can only be reclaimed to the Free List when a later instruction redefining the associated architectural register gets retired from ROB—because the physical register value is no longer used thereafter.

5.1.2 PSP vs WSP

PSP has been a *de facto* standard for server-class systems backed with Intel Optane persistent memory (PMEM) to ensure the crash consistency of their user applications. However, this paper argues that PSP is inferior to WSP for 3 reasons: high performance overhead, programming/maintenance burden, and the risk of losing all system-level states upon power failure.

First, the *app-direct mode* of PMEM cannot take advantage of the deep cache hierarchy despite the ever-increasing data footprint of PSP applications. Our experiment (Section

5.6.2) indicates that due to the inability to leverage DRAM as a cache, even an ideal PSP design is significantly (up to 2.4x and 1.39x on average) slower than the *memory mode* of PMEM for memory-intensive applications. Second, PSP is not transparent and requires programmers either to redesign their data structures with persistence and recoverability in mind—incurring severe bugs during development [276–282] and maintenance costs in the future [303, 321–323]—or to leverage transactions for mitigating the programming burden[4]. Third, PSP can only recover the states of user applications and hence puts operating systems at the risk of losing their entire states upon power failure, while WSP like PPA can ensure that the entire system states are consistent across power failure; see Section 5.4 for details.

Not only does WSP eliminate PSP programming and maintenance costs, but it also makes persistent applications faster with the DRAM cache. Of course, for those using PMEMs memory mode to leverage the deep cache hierarchy, WSP offers them persistence and crash consistency without hurting the transparency and performance. This is particularly beneficial for HPC applications (e.g., Mini-apps) whose states must be saved to storage on a regular basis. We believe that lightweight persistence/recoverability, e.g., PPA, can enable performant application-level resilience—related to one of the nations exascale challenges [324–326]—by obviating the need for expensive periodic global checkpointing to storage.

5.1.3 Region-Level Persistence for WSP

Prior techniques [59, 245, 246] recently investigate region-level persistence to provide crash consistency in energy harvesting systems (EHSs) [95, 134, 250, 327] where WSP is the norm. These techniques partition the program into a series of regions (akin to recoverable epochs) where their boundaries serve as recovery points. Either compiler [59, 245, 246] or hardware (this work) is responsible for the region formation and the persistence of each region. In particular, each region should ensure that all its stores are persisted before the next region starts so that the program can be recovered by restarting the power-interrupted region upon power back.

[4]↑Either way, the resulting performance overhead is so significant that PSP cannot be used for those who expect similar performance to that of running their applications in PMEM's *memory mode*.

However, such a region-level persistence scheme incurs a non-negligible performance overhead, since the program must wait at each region boundary for the preceding region to persist its stores, i.e., pausing until they are all written back to nonvolatile memory (NVM). While the prior work leverages ILP to overlap the persistence latency with the execution of other instructions, they still cause significant performance degradation—especially in the presence of a more deep cache hierarchy—because their regions are too short to fully hide the long latency with ILP.

5.1.4 Store Integrity for Performant WSP

The key observation PPA builds upon is that we can safely recover the system states by *replaying stores that are potentially unpersisted before power outage*. Although this principle has been investigated and adapted by many prior approaches as a concept of atomic stores with logging them all [58, 59, 283, 300, 302], the prior schemes suffer from the problem of doubling NVM stores—known as *write amplification*.

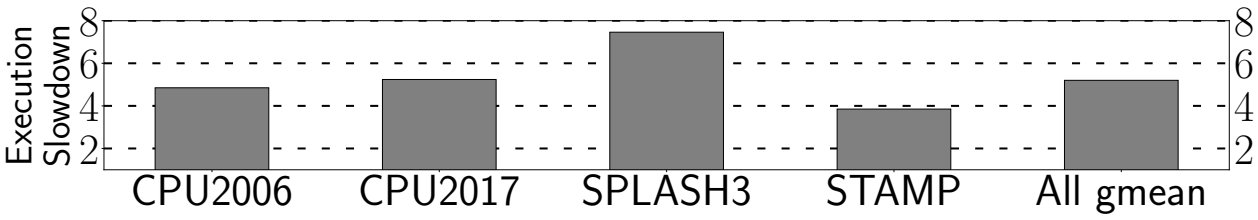


Figure 5.1. ReplayCache's slowdown to the baseline (running original applications on PMEM's memory mode)

To achieve high-performance WSP, we make another observation that crash inconsistency is essentially caused by the mismatch between the program order of committed stores and the order in which their cache blocks are written back to NVM. To be specific, a younger store might be evicted (persisted) to NVM while the older ones are cached; if power failure happens before their persistence, NVM status becomes inconsistent across the failure on which the data of the older stores are lost since they have not been persisted. This finding inspires us to recover the inconsistent NVM status by rewriting **only** those potentially unpersisted stores to NVM in the wake of the power failure—unlike traditional undo loggings that checkpoint

all stores. The upshot is that no matter which random order of persisted stores is across power failure, it is always possible to correctly recover by replaying all committed stores left behind the failure and resuming from the last commit point. Zeng et al. show that store replaying needs compilers to prevent the store registers from being clobbered by following redefinitions, which requires a special register allocator; they call this *store integrity* in their energy harvesting work, ReplayCache [245], and use compiler-based region-level persistence to divide the program to a series of regions where store integrity is enforced to guarantee crash consistency.

Unfortunately, ReplayCache incurs too much performance overhead (5x average slowdown as shown in Figure 5.1) when used to achieve WSP for server-class cores; see Table 5.2. The reason is 2-fold: (1) ReplayCache's regions are so short (average 12 instructions in regions) that they cannot accomplish enough ILP to hide the region-level persistence latency through multi-level caches. That is mainly due to the inherent issues of ReplayCache's compiler analyses, e.g., function calls/loops, scarce architectural registers, and energy-aware region splitting for avoiding *stagnation* [59, 251] in EHSs. Hence, the short region leads to frequent pipeline stalls at each region boundary serving as a persist barrier; (2) ReplayCache inserts `clwb` after each store to write it back to NVM, which doubles the instruction count and places high pressure[5] on store queue whose overflow stalls the pipeline as well. Unlike ReplayCache, PPA achieves performant WSP for server-class cores causing only a 2% overhead (Section 5.6.1).

5.2 Design

PPA aims to achieve a lightweight WSP that works for a deep cache hierarchy, where DRAM cache is used as in PMEM's memory mode, without sacrificing the transparency (i.e., keeping the entire software stack as is and obviating the need for recompilation) and the performance. PPA adopts store integrity for crash consistency, but its novel hardware design for the integrity enforcement makes it possible to realize a performant WSP at a low cost. In particular, PPA leverages ample physical registers in out-of-order cores to preserve

[5]↑The `clwb` instruction occupies a store queue entry.

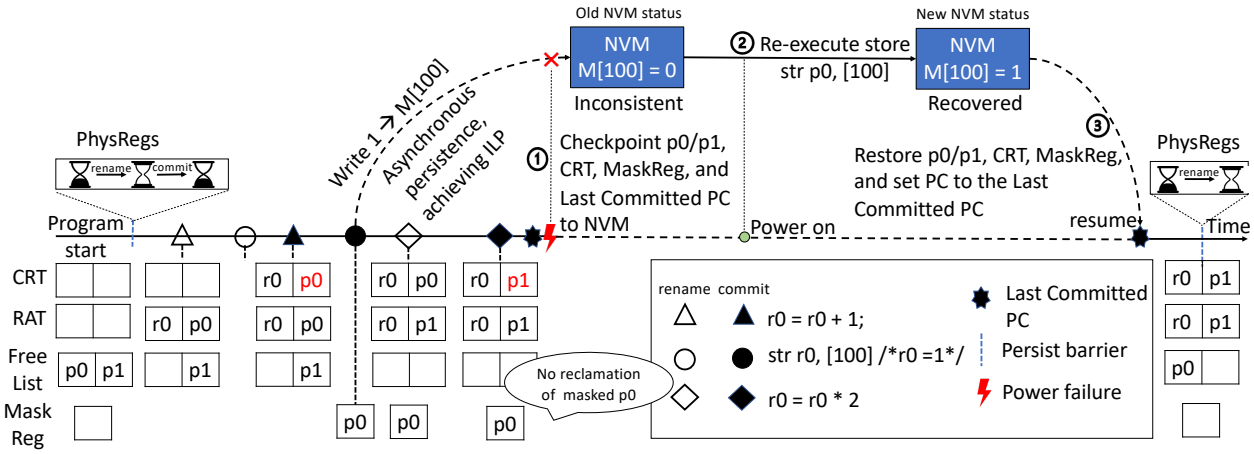


Figure 5.2. PPA overview; for store integrity, $p0$ is not recycled even after the multiplication commits

store registers; it dynamically delineates the region (epoch) boundary whenever physical registers run out. In this way, sufficiently long store-integrity regions serve as the basis for failure recovery, thus effectively hiding the store persistence latency.

Figure 5.2 depicts how PPA realizes WSP based on register renaming of a modern out-of-order core[6]. In the figure, commit rename table (CRT), register alias table (RAT), and Free List are existing microarchitectural components. CRT keeps the mapping from an architecture register to a physical register for committed instructions, while RAT records that for in-flight instructions. The free list maintains free registers for later renaming use. PPA proposes *MaskReg*, a bit vector, to record which physical register is used by prior committed stores and therefore should not be remapped (overwritten) by the following redefinitions.

In Figure 5.2, upon renaming a destination architectural register $r0$ (i.e., $\triangle$ $r0 = r0 + 1$), the processor removes a physical register $p0$ from the free list and puts the mapping from $r0$ to a physical register $p0$ into RAT as usual. Thus, for renaming the following store (◯), i.e., $str\ r0, [100]$, the reference to $r0$ is replaced by $p0$. Once the addition instruction commits (▲), making the defined value of $r0$ architecturally visible, the processor puts the mapping $r0 \rightarrow p0$ in CRT as usual. In particular, on the commit of the store (◯), PPA starts to track $p0$ in MaskReg, watching it for store integrity. When the following redefinition of $r0$ is

[6]↑We assume a server-class Intel Skylake core [86] though PPA can be generalized to other out-of-order cores.

renamed (◊), the multiplication instruction obtains $p1$—not $p0$ since it is already masked—from the free list with RAT updated accordingly. Additional pipeline details are deferred to Section 5.2.3.

5.2.1 Dynamic Region Formation

Similar to prior techniques [59, 245], PPA also provides region-level persistence. However, what makes PPA stand out from them is its ability to build regions dynamically without user intervention, recompilation, and significant performance loss. PPA instead leverages an existing microarchitectural feature to deliver the region formation with the store integrity enforced at a low cost. In particular, PPA considers the number of free physical registers to decide when to place a region boundary (persist barrier). As shown in Figure 5.2, PPA places the boundary (barrier) when no free physical register is available at the renaming stage of the out-of-order pipeline (⧖). Once PPA ensures at each region boundary that the committed stores of the finished region are all persisted, it reclaims their physical registers with MaskReg cleared—before starting the next region, as shown at the left bottom of the figure.

5.2.2 HW-Based Asynchronous Store Persistence

Although prior software-logging-based PSP techniques guarantee consistent NVM status across power failure, they incur significant performance overhead because of a persist barrier (e.g., `clwb` and `sfence` in x86). In contrast, PPA does not block the pipeline execution while stores are being persisted to NVM. That is, once the data being stored is merged into the L1 data cache (◯ in Figure 5.2), the L1 data cache controller immediately asynchronously writes back the resulting dirty cacheline to NVM in the background, keeping the pipeline busy with other instruction executions in the meantime.

To ensure all stores prior to the end of a region are already persisted in NVM before committing following instructions, *PPA treats every region boundary (the last instruction of each region) as a special persist barrier*. Therefore, the core pipeline waits until the acknowledgment of persisting the region's all prior stores in NVM is received by the core

before entering the next region. While stalling the pipeline can lead to a slowdown due to the wasted cycle time, our experimental results show that our hardware-based store persistence has a minimal impact on the pipeline performance due to long enough regions (see Section 5.6.5) and thus resulting in negligible stall cycles at the end of regions (see Section 5.6.3).

5.2.3 Dynamic Enforcement of Stores Integrity

Figure 5.2 shows how PPA ensures store integrity on the fly during the pipeline execution. Upon retiring *str* $r0, [100]$ (◯ in the figure) whose $r0$ was renamed to $p0$, PPA masks $p0$ in MaskReg to notify it is occupied by the store, which makes the target register of the following multiplication instruction renamed to $p1$ (◊) instead of $p0$. Unlike conventional cores, upon retiring the multiplication (⧫ $r0 = r0 * 2$) with updating CRT with $r0 \rightarrow p1$, PPA does not reclaim the physical register $p0$ which is associated with $r0$'s prior definition $r0 = r0+1$—though its value can no longer be used due to the retirement of the multiplication overwriting $r0$. That is because $p0$ is masked as a committed store register in MaskReg, and it should be preserved in case of power failure so that the store can be replayed in the wake of the failure. In this way, PPA not only guarantees store integrity in each region but also achieves performant WSP with a much longer region size than the compiler-based prior work [245], thus hiding the store persistence latency.

5.2.4 Checkpoint and Recovery Protocol

To achieve correct program execution across power outage, all the store registers preserved by our register renaming trick must survive power failure. For this reason, PPA should maintain necessary microarchitecture status such as CRT across the outage. Also, in the wake of power failure, PPA should be able to resume the program right after the last commit point behind the outage.

In light of this, PPA exploits just-in-time (JIT) checkpointing to save minimal architectural states—e.g., physical register $p0$, CRT, and the last committed PC as shown in Figure 5.2 (①)—to a designated checkpoint storage in NVM, when power is about to be cut off. Owing to its simplicity, PPA only requires a tiny capacitor to secure energy for

JIT checkpointing, while Narayanan's [110] and eADR's demand a significantly large bulky Li-thin battery or supercapacitor [287] (Section 5.6.8). When the power comes back, PPA first replays all committed stores behind the failure, e.g., *str p*0, [100] in Figure 5.2 (②), and restores other checkpointed states such as CRT (③). Then, PPA resumes the interrupted region from the latest uncommitted instruction following the *last committed PC* to continue program execution. More details are deferred to Section 5.3.5 and Section 5.3.6.

5.3 Implementation

Figure 5.3 shows PPA's microarchitecture with its 3 newly added components; *Last Committed Program Counter (LCPC)*, *Store Operands Mask Register (MaskReg)*, and *Committed Store Queue (CSQ)*. The *LCPC* register keeps the PC of the last committed instruction so that a power-interrupted program can resume thereafter in the wake of power failure. Note that PPA does not save or recover architectural status related to speculation, such as in-flight instructions in ROB. The *MaskReg* comprises as many bits as the PRF size. Each set bit of *MaskReg* indicates that the corresponding physical register has been used as an operand of any committed store in the current region and thus prevents those physical registers from being updated by the following instructions of the region. Finally, the *CSQ* is a circular FIFO queue for tracking committed stores per region. When a store retires from ROB, a pair of (1) the index of the source physical register and (2) the destination physical address of the store is inserted into the rear position of *CSQ*.

Actions of PPA across an Outage: Upon a power outage, PPA has 5 components (shaded in Figure 5.3) JIT-checkpointed in NVM: CSQ, LCPC, CRT, MaskReg[7], and the physical registers tracked by CSQ/CRT. When power comes back, PPA (1) restores the checkpointed registers, MaskReg, CRT, LCPC, and CSQ from NVM, (2) scans CSQ entries from front to rear re-executing the stores committed before the outage[8], (3) populates RAT with the restored CRT, and (4) resumes the power-interrupted program right after LCPC.

[7]↑MaskReg should be JIT-checkpointed as well to prevent store registers therein from being recycled even after power comes back.

[8]↑Even if some stores might have already been persisted, there is no harm to execute them again as each store is idempotent [42, 114, 130, 132, 133, 215, 244].

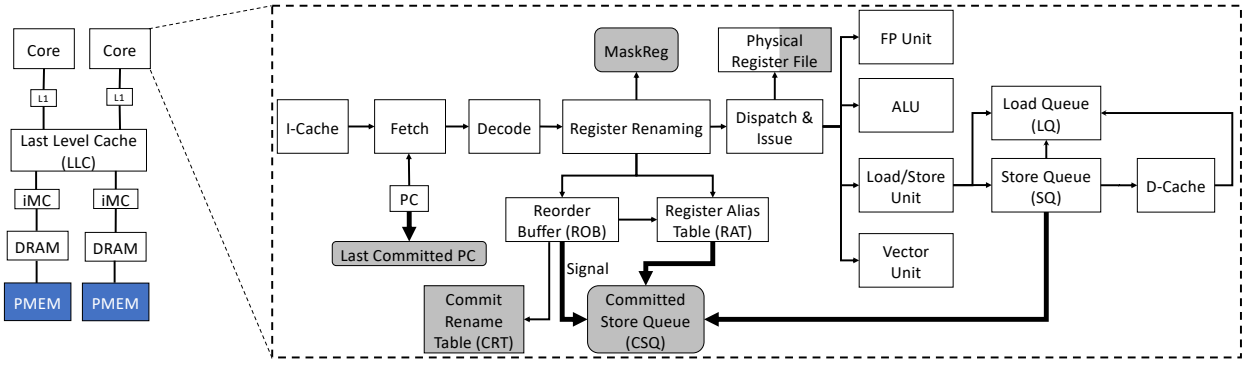


Figure 5.3. PPA with Intel's memory mode; rounded rectangles corresponds to new components, while thick lines to new signal or data paths; shaded parts are JIT-checkpointed upon an outage (PPA checkpoints only those registers masked by MaskReg/CRT)

Note that once a region is persisted at the boundary, i.e., the pipeline receives an acknowledgment that all the committed stores of the region have been persisted to NVM, PPA clears both the CSQ and the MaskReg—reclaiming the store registers masked therein—before starting the next region. Thanks to the long region size (Section 5.3.1) and the asynchronous writebacks (Section 5.3.3), PPA effectively hides the store persistence latency at each region end.

5.3.1 Enforcing Store Integrity Efficiently

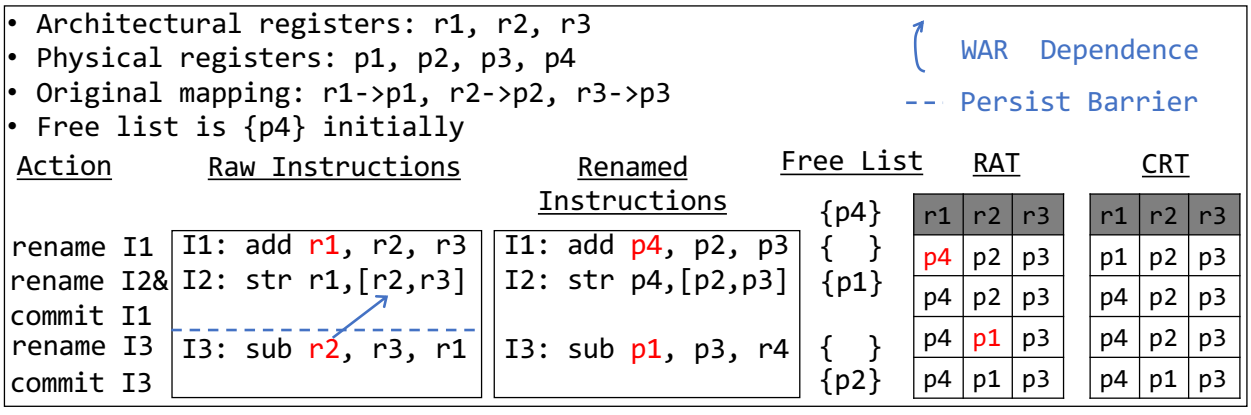


Figure 5.4. Impact of register renaming on the region length

At first glance, forming store-integrity-preserving regions seems easy, i.e., placing a region boundary right before the redefinition of store registers to preserve their values within a region. For example, placing a region boundary (persist barrier) after store $I2$ in Figure 5.4 ensures store integrity and post-crash consistency but yields short regions because of a write-after-read (WAR) dependence on store register $r2$. A sophisticated compiler approach might form relatively longer regions by renaming the redefinitions of previous store registers—unless architectural registers run out—as in ReplayCache [245]. However, the prior approach [245] to store integrity still generates short regions due to a limited number of architectural registers, e.g., 16 general-purpose registers in x86. The crux of the problem is that ReplayCache pays for persist barrier overheads so often at each end of such short regions.

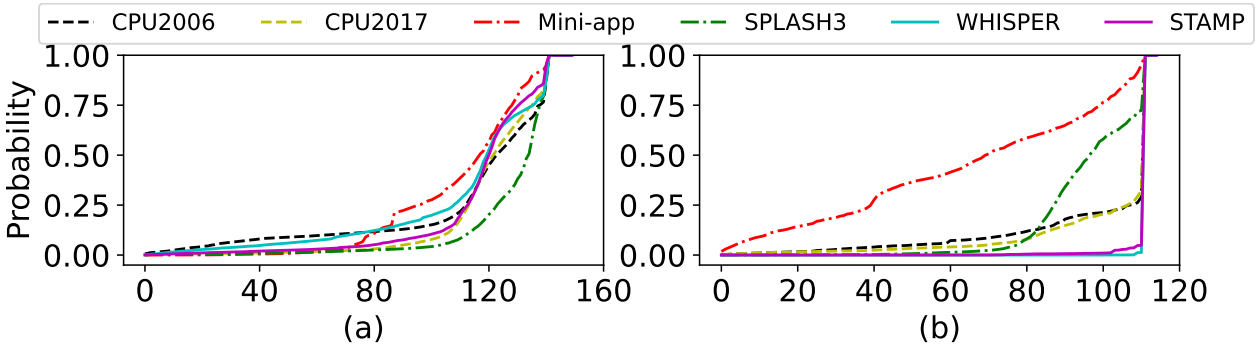


Figure 5.5. (a): CDF of free integer registers; (b) CDF of free floating-point registers

Fortunately, the out-of-order cores already have the ability to eliminate WAR dependence with register renaming [305], e.g., renaming $r2$ to $p1$ for the subtraction $I3$ in Figure 5.4. That way the pipeline can execute $I3$ without redefining a register $r2$, thus obviating the need for the persist barrier after the store $I2$—unlike the prior approach [245] that needs the barrier to persist the store before $I3$—forming longer regions than it can.

Note that the number of physical registers is much larger than that of architectural registers and that they are often idle. Figure 5.5 shows that CDFs of free integer and floating-point registers—sampled every cycle[9]—respectively. For example, 75% of the program exe-

[9]↑We measure the number of free physical registers every cycle at the renaming stage of an OoO core (Table 5.2).

cution cycles, the baseline core has 138/110 integer/floating-point registers not utilized for CPU2006. As such, PPA's region size can be sufficiently long to hide the persistence latency of stores in each region without incurring slowdown a lot.

5.3.2 Forming Longer Regions at a Low Cost

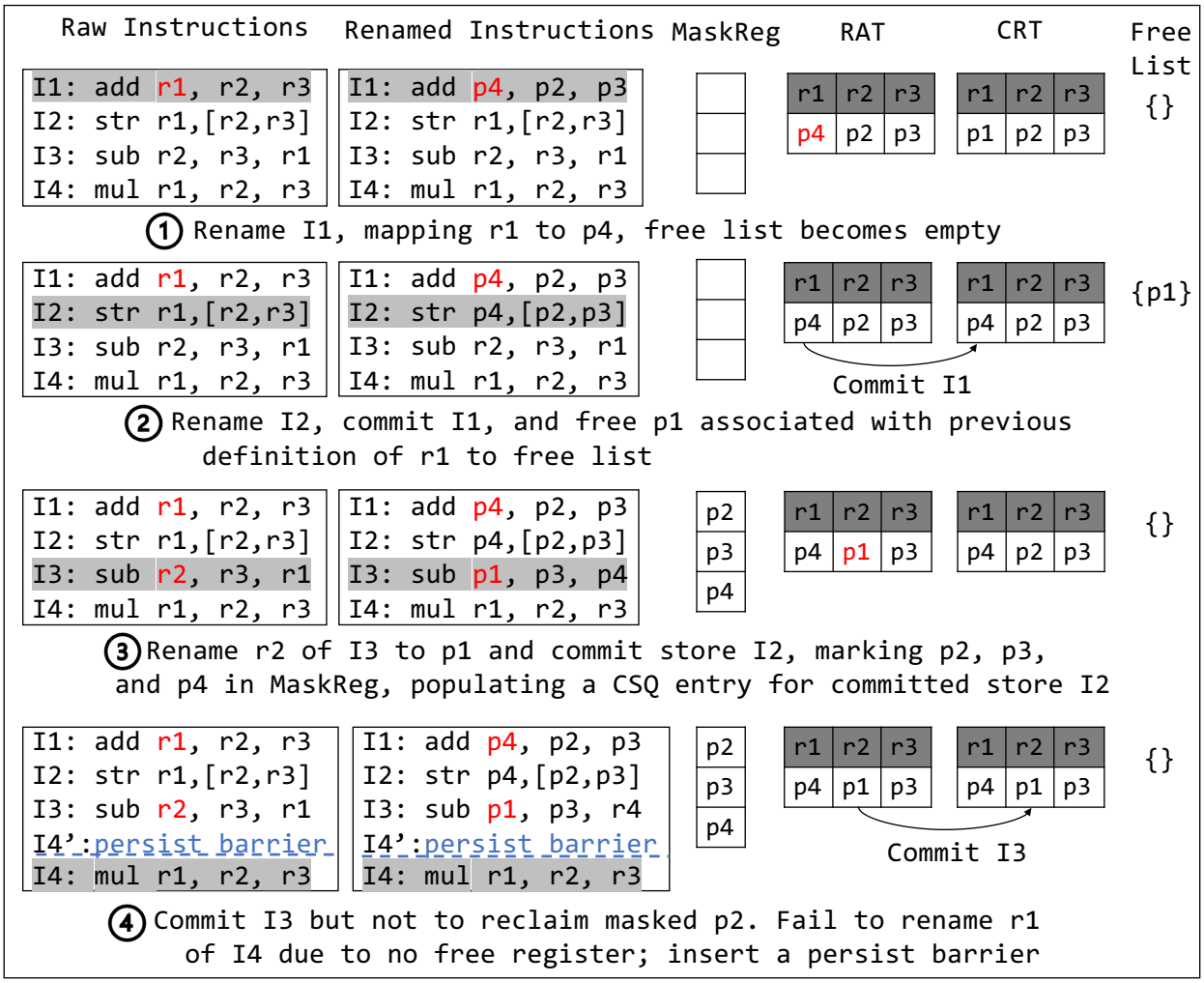


Figure 5.6. Dynamic region partitioning by physical register file size; the free list shows its status after the action

With the above observation in mind, PPA proposes a minimal change to the instruction pipeline so that it enforces store integrity during the execution of each region. That is, PPA dynamically partitions program to a series of regions by placing a region boundary, i.e., a

persist barrier, upon a pipeline stall at the renaming stage; if a register-defining instruction cannot be renamed due to the lack of a free physical register in free list, PPA injects a persist barrier **right before** the instruction (see Figure 5.6 for details). Once the pipeline retires the persist barrier at the boundary of a region, i.e., all its stores are already sure to have persisted, PPA acknowledges the renaming stage to reclaim the physical registers masked by MaskReg, clears it, and resumes the pipeline to start the next region.

Figure 5.6 shows a step-by-step example of how to perform dynamic region formation while preserving registers of stores. We assume a 4-bit *MaskReg* for total 4 physical registers $p1 - p4$. Initially, *MaskReg* is empty, and $p1 - p3$ are occupied by previous definitions of register $r1 - r3$, and a free list contains only $p4$. When renaming $r1$ of the addition instruction $I1$ at step ①, PPA maps $r1$ to the only free physical register $p4$ and updates the RAT and the free list accordingly. Then, at step ②, when renaming a store instruction $I2$, the references to architectural registers are replaced by physical registers as usual. At the same time, the pipeline retires $I1$ instruction, deallocating $p1$ associated with $r1$'s previous definition (not shown in the figure) and updating the CRT with $r1 \rightarrow p4$. At step ③, the pipeline renames $r2$ of $I3$ to $p1$, with RAT and the free list correspondingly updated, and commits the store $I2$ setting the bits of $p2 - p4$ in *MaskReg* to 1[10] and populates a CSQ entry in its back position for the committed store $I2$.

In particular, at step ④ where the pipeline commits $I3$ and renames $I4$, PPA takes different actions from the traditional out-of-order pipeline that allows $p2$ to be remapped. PPA does not deallocate physical register $p2$ associated with $r2$'s previous definition (not shown in the figure), though $I3$ commits redefining $r2$. This is because $p2$ is masked in $MaskReg$ as a store operand. However, at this moment, there is no physical register in the free list, which makes PPA fail to rename the register $r1$ of $I4$. Thus, PPA injects a persist barrier here as a region boundary right before instruction $I4$. Once the barrier gets retired, PPA reclaims all masked physical registers $p2-p4$ to the free list, clears *MaskReg*, and starts a new region allowing them to be reused therein.

[10]↑While *MaskReg* could record all operand registers of each store, we opt to keep only a data register as an optimization. See Section 5.3.6 for more details.

Full CSQ as an Implicit Region Boundary: If CSQ becomes full, PPA cannot accommodate stores anymore, thus being unable to replay them for power failure recovery. Thus, PPA treats this event as a virtual region boundary where it waits for all prior stores to be persisted. Once the core receives the acknowledgment of persisting all prior stores, PPA starts a new region with CSQ and MaskReg cleared. Our experiment (Section 5.6.6) shows that a 40-entry CSQ rarely overflows, thus incurring a minimal performance impact.

5.3.3 Region-Level Asynchronous Persistence

In addition to preserving store registers for their integrity in each region, PPA also ensures that its stores are persisted to NVM before moving on to the next region. Instead of leveraging cacheline writeback instruction (`clwb` in x86) that has a lot of drawbacks, as shown in Table 5.1, e.g., occupying a store queue entry for each store, requiring inter-core snooping, and not being able to flush data from core to NVM main memory through a DRAM cache above, PPA leverages the asynchronous store writeback which effectively takes it off the critical path [121, 300–304]. That is, when the data being stored is merged into L1 data cache after cache coherence transactions are completed, an asynchronous store persistence operation is generated in the write buffer (WB)[11] of L1 data cache and then issued by its controller. The implication is 2-fold: (1) the store persistence happens in the background while the core continues the execution of following instructions, achieving ILP; (2) once a store persistence operation is issued, all other cores already have up-to-date memory data.

Table 5.1. Comparison between PPA and CLWB

	Store Queue Occupied	Single Store Tracking	Snooping	Reaching NVM
CLWB in x86	✓	✓	✓	✗
PPA	✗	✗	✗	✓

Unlike `clwb`, i.e., a cacheline writeback instruction that occupies a store buffer entry, PPA instead uses a counter register in the L1 data cache controller to record the number of stores being persisted, rather than tracking each individual store with `clwb`; the counter increases

[11]↑WB already exists in Intel processors sitting between L1D and L2 cache for buffering dirty cacheline eviction.

for each store performed and decreases every time the controller receives the acknowledgment of the writeback completion. In particular, to lower write traffic towards NVM, PPA performs *persist coalescing* [303] on the WB for the data being persisted. That is, a younger store being persisted is merged with the old *unpersisted* one of the same address sitting in the WB. This is correct because persist barriers ensure that the WB's data—to be persisted—are from the same region, and the stores of the following regions are not performed yet.

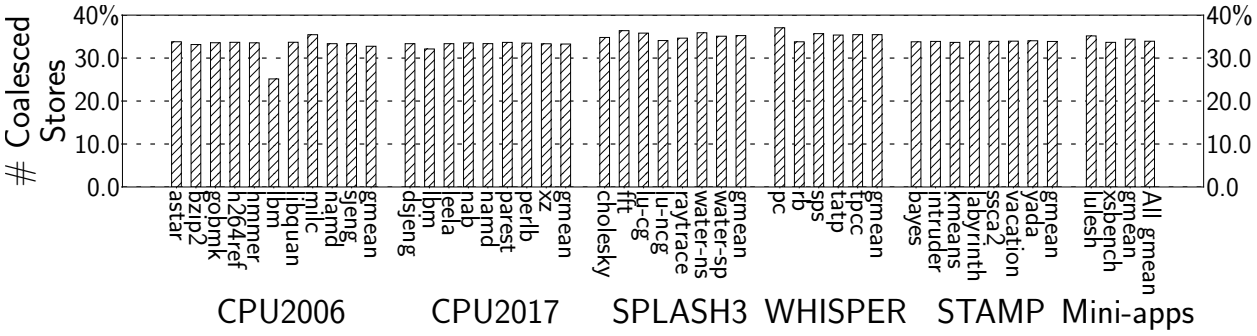


Figure 5.7. Ratio of coalesced stores to total stores

To see how many stores could be coalesced with prior pending stores—i.e., their persist requests have not been initiated yet—in the WB, we collect the ratio of coalesced stores to the number of total committed stores. As shown in Figure 5.7, PPA coalesces 34% of all committed stores on average, resulting in less data to be sent to NVM and therefore achieving high performance.

When the counter hits zero, the controller tells the core that all prior stores in a region are persisted to NVM, allowing both CSQ and *MaskReg* to be cleared. In this way, PPA determines if the pipeline needs to stall at the region boundary by simply comparing the counter with zero. Although such a stall might slow down the pipeline by waiting for the counter to be zero at each region boundary, it turns out that the performance impact is not significant. The reason is that the region-level persistence latency is fully overlapped with the execution of other instructions in the long regions dynamically formed by PPA (Section 5.6.3). Moreover, PPA's asynchronous store writeback does not generate coherence traffic—since each core is responsible for its own writeback—thus reducing the persistence latency further.

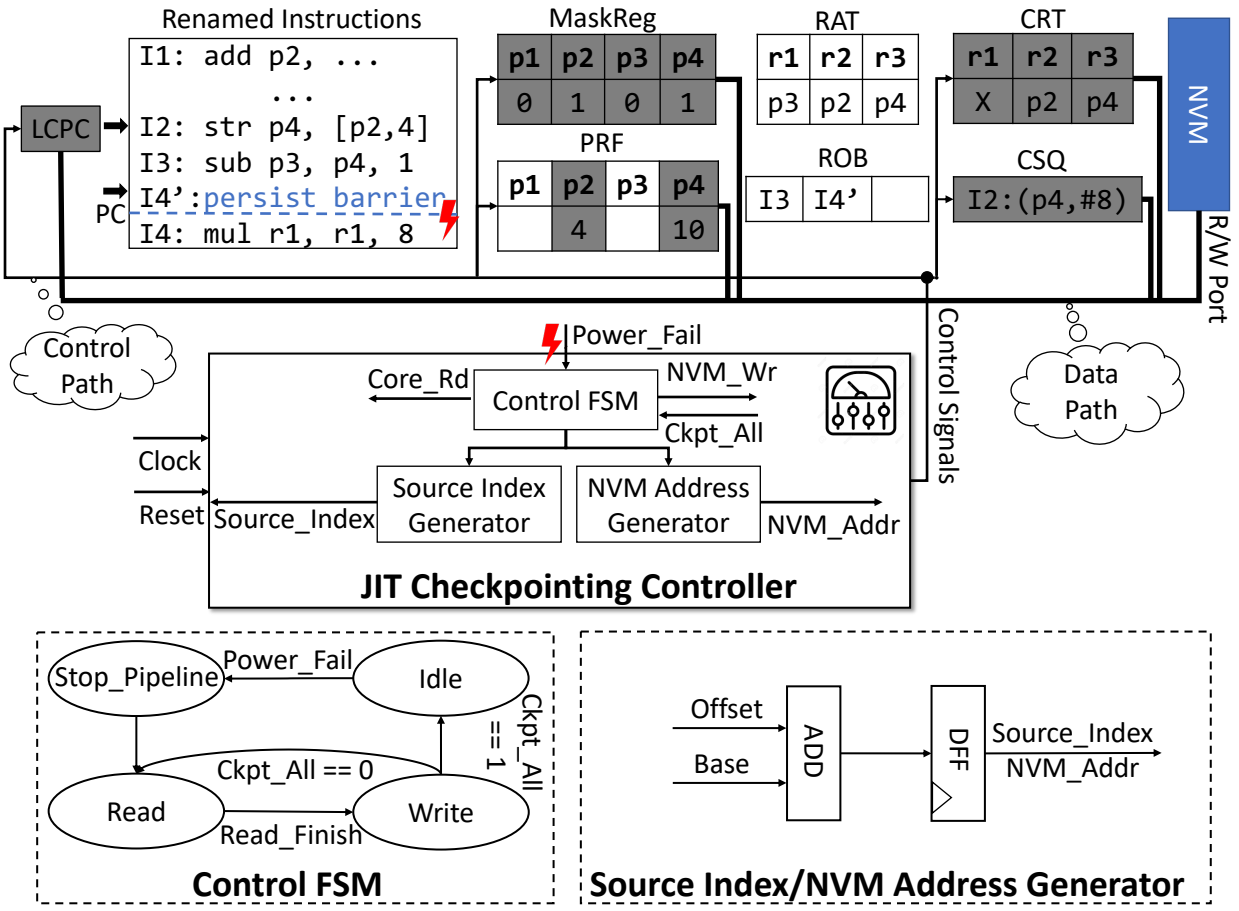


Figure 5.8. JIT Checkpointing logic; gray parts are checkpointed before impending power failure

5.3.4 Lightweight Hardware for Recovery

To achieve highly energy-efficient checkpointing and recovery, PPA needs to checkpoint only essential architectural statuses, e.g., a part of physical registers used by committed stores or linked with committed instructions in the interrupted region, committed stores of the region, CRT, and program counter (PC) of the latest committed instruction, upon power failure. With checkpointing such minimal states, we can still restore consistent memory status by re-executing those stores and then resume the program execution following the latest committed instruction.

To facilitate this, PPA proposes a simple yet hardware-efficient FIFO queue called committed store queue (CSQ) and Last Committed PC (LCPC). Each CSQ entry keeps the source physical register index and the destination (physical) address of committed stores in program order, and LCPC gets updated with the PC value after committing an instruction. Note that CSQ and LCPC do not affect the existing pipeline's timing logic at all because they are out of the critical path. More importantly, CSQ is organized with a read/write port eliminating an expensive CAM structure, making a large CSQ realistic; we only need 40 CSQ entries at most as shown in Section 5.6.6 though. During normal program execution, the port is used to populate a CSQ entry in its rear position and to checkpoint the entire CSQ to NVM upon power failure. Finally, PPA clears CSQ at each region boundary as with MaskReg emptied, i.e., when all committed stores in the finished region become persistent in NVM, before moving on to the next region.

5.3.5 Just-In-Time (JIT) Checkpointing on Power Failure

To ensure correct program recovery across power failure, PPA should checkpoint necessary states when power is about to be cut off. Figure 5.8 shows how such just-in-time checkpointing works with its circuitry implementation; upon the delivery of `Power_Fail` signal, PPA saves the contents of its 5 structures to NVM, i.e., MaskReg, commit rename table (CRT), committed store queue (CSQ), a part of PRF, and last committed PC (LCPC). Note that PPA only checkpoints those physical registers marked by CRT or CSQ entries in that neither free registers ($p1$ in the figure) nor uncommitted registers ($p3$ defined by $I3$) affect correct program recovery. Similarly, PPA does not have to checkpoint any other status of in-flight instructions, e.g., their RAT and ROB entries. This is because PPA can resume the execution of power-interrupted program from the latest uncommitted instruction following LCPC, when power comes back.

As with prior work on JIT checkpointing [95, 98, 189, 205, 328–330] developed for energy-harvesting systems [245, 249, 327, 331] to realize power failure recovery, PPA implements a controller that governs checkpointing and recovery[12] operations, according to each signal

[12]↑Recovery operations are not shown in Figure 5.8 as they are the opposite of checkpointing operations.

delivered on power failure and its wake-up. As shown in the middle of Figure 5.8, the controller consists of 3 components: (1) *Control Finite State Automaton (FSM)*, (2) *Source Index Generator (SIG)*, and (3) *NVM Address Generator (NAG)*. *FSM* is responsible for generating control signals to checkpoint PPA's 5 structures, i.e., MaskReg, CRT, CSQ, PRF, and LCPC, into their storage in NVM. During the checkpointing process, *FSM* triggers *SIG* and *NAG* that share the same logic—shown in the bottom right of the figure—for the sum of the inputs `Base` and `Offset` to determine (1) what to be checkpointed and (2) where to save in the NVM, respectively.

It is worth noting that PPA activates its checkpointing controller only on power failure, and therefore it is out of the critical path most of the time as long as power is on, i.e., PPA does not have to optimize the controller's circuitry for latency. This allows PPA to keep the controllers hardware design simple by sequentially checkpointing PPA's 5 structures[13] one entry at a time. To illustrate, as shown at the bottom left of Figure 5.8, *FSM* is triggered upon `Power_Fail` to transit from *Idle* stage to *Stop_Pipeline* stage, where PPA stops the core pipeline to preserve the contents of the 5 structures. Then, *FSM* moves to *Read* stage, raising the read signal `Core_Rd` on the control path so that the entry indexed by *SIG* can be read in each of 5 structures across which `Base` and `Offset` are properly updated. Upon the delivery of `Read_Finish` signal, *FSM* enters `Write` stage enabling the write signal `NVM_Wr` to write the data to the NVM address generated by *NAG*. Once the writing is done, *FSM* either goes back to *Read* stage or exits to *Idle* provided if `Ckpt_All` is activated, i.e., all 5 structures are completely checkpointed.

To realize the above sequential checkpointing while maintaining a low hardware complexity, PPA exploits the existing non-temporal path [332] in x86 processors to deliver data to NVM—other than introducing a new data path. This indicates that PPA checkpoints its 5 structures at an 8-byte granularity as with their entry size (Section 5.6.7). Likewise, *FSM* reads PRF and CRT at an 8-byte granularity, which seems possible given that they are implemented with SRAM [317, 319, 333]. The takeaway is that the aforementioned JIT-checkpointing logic is lightweight, i.e., a few hundred logic gates, keeping the overall hardware cost of PPA minimal (Section 5.6.7).

[13]↑The order in which the structures are visited does not affect the correctness of checkpointing and recovery.

5.3.6 Power Failure Recovery Protocol

To achieve correct program recovery, in the wake of power failure, PPA restores MaskReg, CRT, and checkpointed physical registers by reloading their data from NVM as an opposite operation of the JIT checkpointing. PPA then re-executes those potentially unpersisted stores by reading the CSQ entries checkpointed in NVM. To be specific, for each CSQ entry, PPA gets both the data value by retrieving the restored PRF with an index of the checkpointed physical register and the destination address. That way, PPA writes the data value to the target address. Finally, PPA resets the PC to the instruction following the LCPC to continue the program execution.

5.4 Interaction with OS

This section describes how PPA interplays with the rest of the computing stack, such as the operating systems (OS), to enable system-level crash consistency.

Handling I/O Operations: To the best of our knowledge, supporting irrevocable operations such as I/O remains an open problem. PPA can be extended to have a battery-backed buffer for crash-consistent I/O operations. In this way, PPA considers any store to the buffer as persisted.

Context Switching: PPA treats context switching as is without any special consideration. In particular, PPA does not differentiate between kernel code and user program thanks to the benefits of WSP. While keeping context switching as is, PPA still guarantees correct process (de)scheduling and resumption. That is because PPA ensures that the architectural states, e.g., stores and architectural registers, of a descheduled process are crash-consistent by following PPA s JIT checkpoint and recovery protocol. That being said, PPA might have an indirect impact on performance, provided that a region boundary is introduced during context switching. In reality, such a case rarely occurs because PPA forms reasonably long regions (see Section 5.6.5), keeping the frequency of encountering region boundaries low. Even if the case occurs, i.e., PRF runs out in the middle of the context switching, and the resulting region boundary incurs the region-level persistence overhead, PPA can still minimize the stall cycles at the boundary leveraging the asynchronous store persistence, e.g., only

a few stall cycles occur on average (see Section 5.6.3). It turns out that they are negligible compared to typical context switching overhead (e.g., 5-20 μs) [334–336]. Consequently, the context-switching performance would practically be the same with PPA.

Interrupt Handling and System Calls: There is no special treatment of PPA for Interrupt handling[14] and system calls—that rely on trap instructions (`syscall` in x86_64)—for the same reason above. That is, PPA guarantees that any architectural state is consistent across power failure. As such, PPA can resume interrupt handlers and system calls exactly from the power failure point without rollback. For an interrupt handler that encounters power failure in the middle of the execution, PPA can recover all committed but unpersisted stores and architectural registers and resume the handler from the last commit point in the wake of the failure.

5.5 Discussion

Recovery for Multi-Cores: To guarantee correct recovery for multi-threaded applications on multi-core processors, we assume data-race-free (DRF) applications as required from the C/C++11 onward. DRF implies that conflicting accesses should be explicitly ordered by a synchronization primitive, e.g., serializing them in a lock-protected critical section or leveraging an RMW (read-modify-write) instruction. PPA treats all synchronization primitives, including atomics and fences, as a region boundary so that their actions comply with PPA's original recovery protocol in case of power failure; for each synchronization primitive running on a core, it cannot be committed until all stores of its region are sure to have been persisted to NVM with the CSQ of the core emptied. For example, the stored data before a lock release can exist in the CSQ of at most one core. The implication is two-fold: (1) there cannot be multiple pending stores to the same address in the CSQs of different cores due to the absence of data races; (2) thus, we may replay stores in the cores CSQs in an arbitrary order, which still achieves correct recovery—because each core's CSQ entries are disjoint with any other core's CSQ entries. That is, PPA can restore consistent NVM states

[14] We use the term interrupt to describe software exception and hardware interrupt.

of DRF applications—though it lets each core perform the recovery protocol (Section 5.3.6) individually—without maintaining the recovery order among the cores.

Memory Consistency Model: Although PPA is evaluated with X86 ISA (total store ordering), it works well for other consistency models, e.g., relaxed memory ordering (RMO) in ARM and RISC-V, because PPA leaves load/store unit (LSQ) as is by proposing a tiny CSQ. One might think of gating those retired stores in store buffer (SB) without merging them to L1 cache as an alternative. However, it complicates the hardware design and limits the performance optimizations of RMO for 3 reasons: (1) region-level persistence prohibits inter-region store coalescing and out-of-order store writeback from SB to L1 data cache; (2) it is hard to enlarge the SB size for hiding long memory latency. That is because SB's CAM searching structure is expensive, and it must provide data within L1-hit time, which would otherwise complicate the scheduling loads with variable latency; (3) data being stored exists in both SB and PRF, wasting the energy to checkpoint the same data twice.

In-Order Cores and ROB-Style Register Renaming: Our design can be easily extended to provide WSP for both cores by accommodating data values (rather than indexes to PRF as in the current PPA) and destination addresses of committed stores in the CSQ as usual. Across power failure, the CSQ entries can be checkpointed and thus restored to recover inconsistent NVM status via replaying.

Multiple Memory Controller (MC) Support: PPA naturally supports multiple memory controllers without any hassle. This is because PPA only moves on to the next region once all stores of the prior region are persisted in NVM with the help of region-level persistence (Section 5.3.3); this makes it impossible to persist a younger store (in program order) destined to a near MC before the older one to a far MC, if the two stores are separated in different regions. Even if the stores exist in the same region and its power failure exposes the possible ordering violation, PPA replays them all together with other stores of the power-interrupted region in the wake of the failure. Consequently, either way PPA prevents crash inconsistency from occurring in the presence of multiple MCs.

5.6 Evaluation

All programs are compiled with -O3 flag and are statically linked. We use the Clang/LLVM 13.0.1 compiler [139, 337] to build the baseline binaries with default compilation flags. We implement the same ReplayCache region formation in the same compiler to build store-integrity binaries with disabling ReplayCache's energy-aware region splitting to enlarge the region size as much as possible.

Table 5.2. Microarchitectural Parameters

Component	Configuration
Full System Mode	Ubuntu 18.04 and Linux Kernel 5.4.46
Processor	8-core 4-width x86_64 OoO processor at 2GHz. Unified PRF, ROB/IQ/SQ/LQ/Integer PRF/Floating-Point PRF: 224/97/56/72/180/168
L1I	private 32KB, 8-way, 64B block, 3 cycles
L1D	private 64KB, 8-way, 64B block, 4 cycles, write back
L2	shared 16MB, 16-way, 64B block, inclusive, 44 cycles, write back
DRAM Cache (LLC)	shared direct-mapped, 4GB, DDR4 2400 8x8
PMEM	32GB, Read = 175ns/Write = 90ns, 16-entry WPQ [120, 338], 2.3GB/s write bandwidth [338]
CSQ	40-entry FIFO queue

We use the cycle-accurate simulator gem5 [204] to model an 8-core (one thread per hardware core) x86_64 Skylake-X processor with *two integrated memory controllers*, each of which manages a DRAM as an off-chip direct-mapped cache as with PMEM's memory mode. Table 5.2 shows the details of the microarchitectural parameters.

To measure the impact of PPA on varying programs, we choose 6 benchmark suites, e.g., CPU2006/2017 [140, 141, 339, 340], SPLASH3 [217], STAMP [306], WHISPER [307], and Mini-apps [218, 219], which represent different application domains from CPU performance benchmarks, shared-memory multi-core systems, transactional applications, key-value stores, to memory-intensive programs.

We simulate the entire SPLASH3/STAMP/WHISPER program in the full system (FS) mode of gem5 with 8 cores by default. To stress the memory system and demonstrate the benefits of enabling DRAM as a cache, we use reference inputs to simulate SPEC CPU appli-

Table 5.3. Data inputs for DOE Mini-apps and WHISPER apps

Application	Short Description	Simulation Data Input	Memory Footprint
LULESH [218]	High instruction and memory-level parallelism.	-s 100	664MB
XSBench [219]	Stress memory system with little computations.	-s small	241MB
PC [118]	Update in hash-table.	8 100000	196MB
RB [118]	Insert/delete nodes in a red-black tree.	8 100000	166MB
SPS [118]	Swap random entries of an array.	8 200000	264MB
TATP [118]	`update_location` transaction.	8 100000	287MB
TPCC [118]	`add_new_order` transaction.	8 100000	110MB
r20w80 [341]	Memcached with 20% reads and 80% writes	-m 1000 -t 8	189MB
r50w50 [341]	Memcached with 50% reads and 50% writes	-m 1000 -t 8	189MB

cations and the data inputs specified in Table 5.3 for Mini-apps and WHISPER. Additionally, we modify the source code of WHISPER applications to increase the key/value sizes, keeping their data footprint large enough; see Table 5.3. Similarly, we follow the prior work [307] using Memcached 1.6.18 [341] as a server and `memaslap` from libMemcached 1.0.18 [342] as a client to initiate 8 threads sending 10000 requests to the server. For each memaslap request, we test two ratios of read-to-write operations: 20/80 and 50/50 for write-intensive and read-intensive. In particular, we set the key and value sizes of Memcached to 64 bytes and 1KB, respectively. We follow the same way as prior work [40, 114, 116, 119, 214, 343, 344] to fast forward the first 5 billion instructions and then simulate the next 1 billion instructions with a detailed CPU model.

5.6.1 Run-time Overhead Analysis

As a comparison, Figure 5.9 presents run-time overheads of PPA and the state-of-the-art WSP—Capri [60] which incurs high hardware costs due to the separate FIFO persist path between the core and NVM and the complex undo+redo logging structures; see Table 5.6 for the comparison. To be practical, we set the persist path bandwidth of Capri to 4GB/s

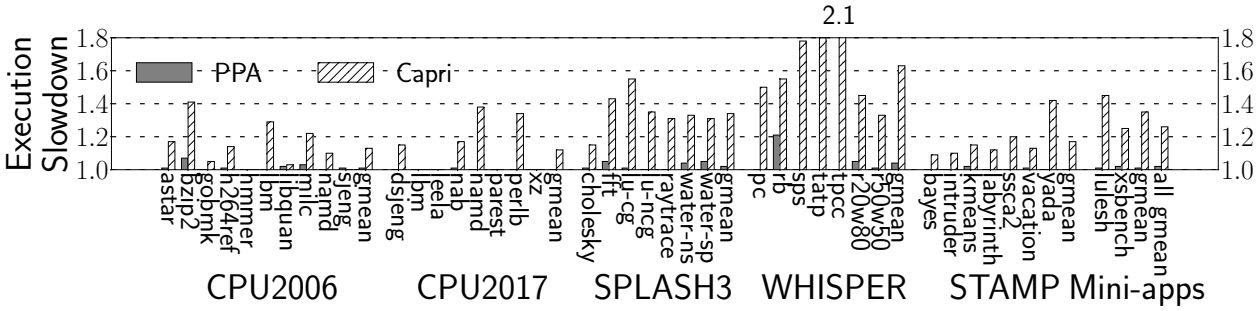


Figure 5.9. Normalized slowdown of PPA and Capri to the baseline (original binaries running on PMEM's memory mode); lower is better; 40 CSQ entries

instead of its original unrealistic 32GB/s[15]. PPA incurs an average of 2% overhead, while Capri incurs a 26% overhead due to its 11x shorter regions than that of PPA; see Section 5.6.5. Note that PPA only incurs a slightly high overhead for `rb` of WHISPER due to the relatively higher write traffic towards NVM, as confirmed in Figure 5.16 and Figure 5.19.

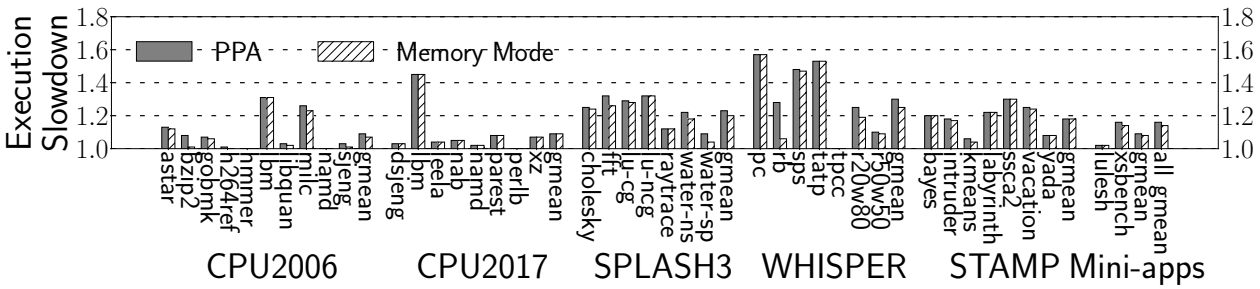


Figure 5.10. Normalized slowdown to a DRAM-only system with 32GB volatile memory; lower is better

We also compare PPA and PMEM's memory mode to the DRAM-only system with a 32GB DRAM. Figure 5.10 depicts that PPA and the memory mode are 16% and 14% slower than the system only with a 32GB DDR4 DRAM, respectively. The results are encouraging in that PPA's cost of making the DRAM-only system persistent is comparable to the run-time overhead of PMEM's memory mode that does not offer persistence. In particular, `lbm` and `pc` incur e.g., 44% and 58% overheads, respectively. That is because they have poor

[15]↑We get Capri's source code and figure out its default persist path bandwidth is 32GB/s.

locality and thus the DRAM cache only increases the critical path of their memory accesses with a lot of misses.

5.6.2 Comparison to Partial-System Persistence

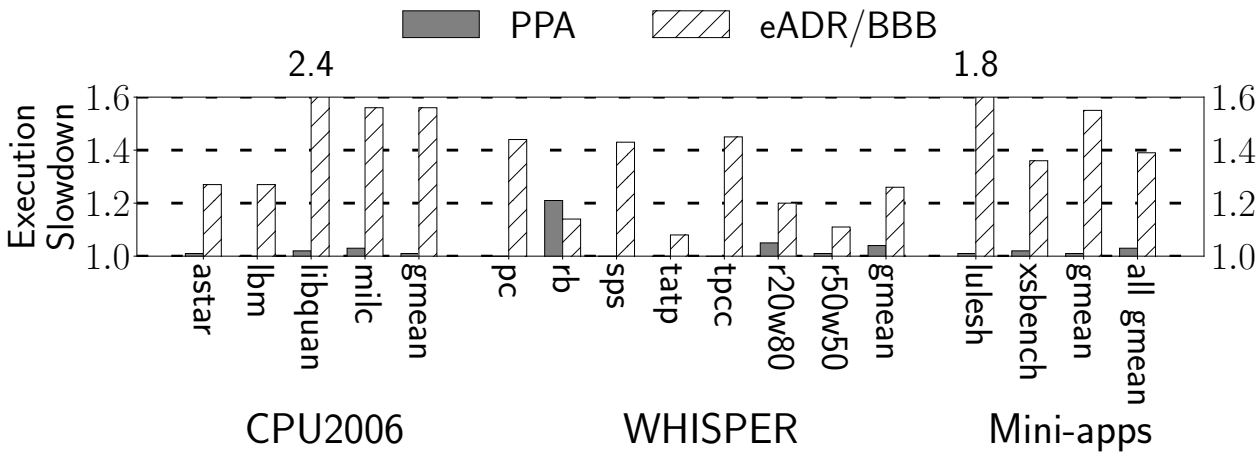


Figure 5.11. Normalized slowdown of PPA and eADR/BBB (ideal PSP) to the baseline (running original program on PMEM's memory mode); lower is better

To demonstrate the benefits of enabling DRAM as a cache for the applications with high L2 miss rates (ranging from 18% to 100%), we compare PPA to an optimized version of BBB [287] whose performance is close to that of eADR, representing the upper-bound performance of a PSP scheme. Figure 5.11 shows that PPA incurs only an average of 3% run-time overhead for these programs, while BBB/eADR slows down the programs by 1.39x on average and up to 2.4x for `libquantum`. Notably, PPA underperforms BBB/eADR slightly for `rb`. The reason is two-fold: (1) PPA leads to higher contention in WPQ (Section 5.6.6) due to the store persistence; (2) `rb` exhibits high locality (4% L2 miss rate) and thus has less write traffic towards NVM for the baseline.

5.6.3 Analysis of Stall Cycles at Region End

Figure 5.12 shows the average ratio of the stall cycles occurred at the end of each region to the execution cycles of that region. Thanks to the sufficiently long region size (i.e., high

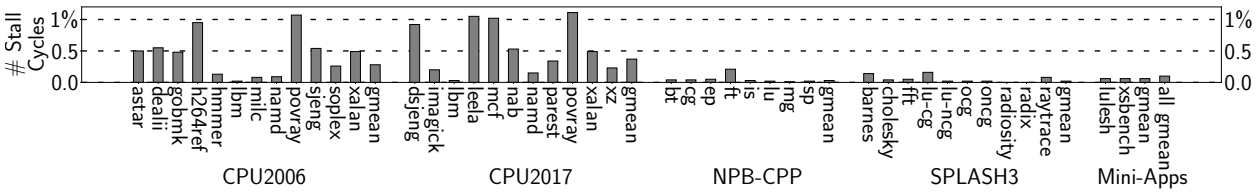


Figure 5.12. Stall cycles at the end of regions as a percentage of their execution time; lower is better

ILP for hiding store persistence latency), PPA only increases the stall cycle ratio of the baseline (PMEM's memory mode) by 0.21% on average, showcasing why PPA incurs a low run-time overhead, i.e., 2% on average. Figure 5.12 also shows why PPA incurs a relatively higher overhead for `water-ns` and `water-sp`; the reason is that, as shown in the figure, these two applications have more stall cycles, i.e., 6.1% and 8.1%, respectively due to their shorter regions and more stores therein (see Figure 5.14).

5.6.4 Impact on PRF Pressure

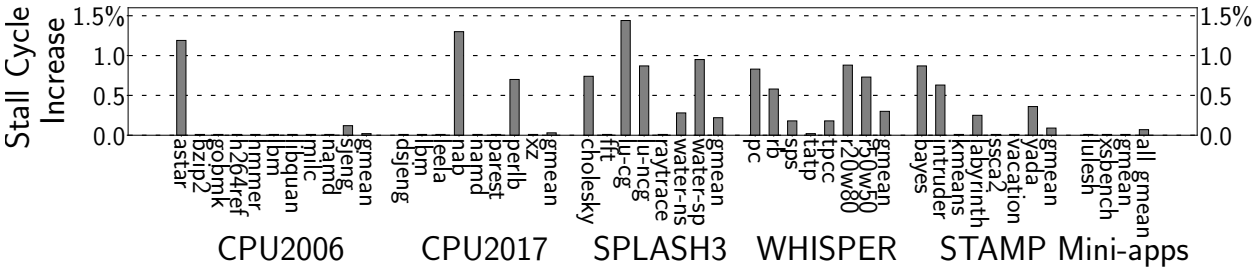


Figure 5.13. Increase in stall cycles at the renaming stage when the core is out of physical registers; lower is better

For both the baseline (PMEM's memory mode) and PPA, we measure the number of stall cycles due to the lack of physical registers in the renaming stage of the simulated core. Figure 5.13 highlights that PPA incurs negligible extra stall cycles (0.07%) on average compared to the baseline. The reason is two-fold: (1) The core pipeline stall caused by running out of free registers rarely occurs due to the sufficient amount of free registers (see Figure 5.5). (2)

Although the stall happens, PPA tends to spend minimal cycles at the end of regions (see Figure 5.12) and thus quickly deallocates their reserved registers for later use.

5.6.5 Dynamic Region Characteristics

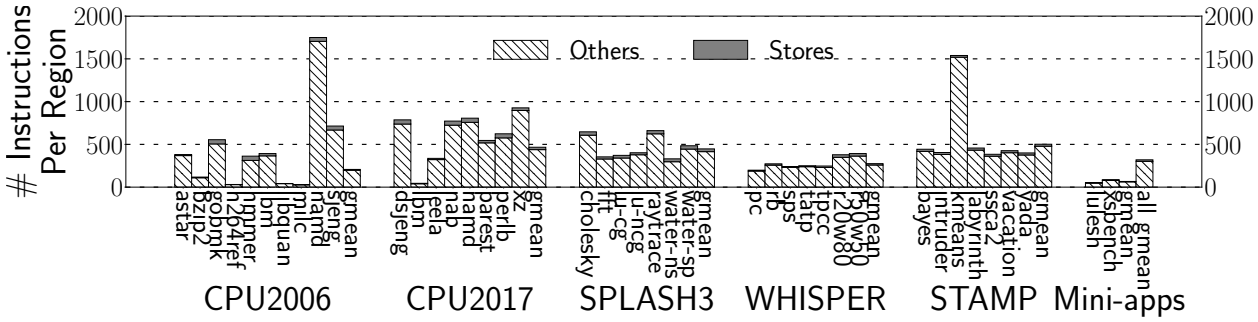


Figure 5.14. Average number of stores and others in regions

To demonstrate why PPA incurs such a low run-time overhead, we measure the number of stores and others in each region. As shown in Figure 5.14, each region has 301 other and 18 store instructions on average thanks to the abundant free registers, while Capri's average region size is only 29. As a result, PPA has enough room to keep the pipeline busy while asynchronously persisting the data being stored to NVM without waiting at each region boundary. Note that some applications, e.g., `bzip2` and `libquantum`, have smaller region sizes due to their heavy register usage.

5.6.6 Sensitivity Analysis

Sensitivity to Deeper Cache Hierarchy: To evaluate the sensitivity to deeper cache hierarchy, i.e., 3-level SRAM caches atop DRAM cache, we add a shared 16MB 16-way set-associative L3 cache of 44-cycle hit latency to both PPA and the baseline (PMEM's memory mode). We also alter the existing L2 cache in Figure 5.2 to a private L2 with 14-cycle hit latency and 1MB. Figure 5.15 shows that PPA incurs a negligible overhead (1%) even when the L3 cache is used atop DRAM cache thanks to PPA's sufficiently long region size (see Section 5.6.5) that can cover the extended store persistence latency through the hierarchy.

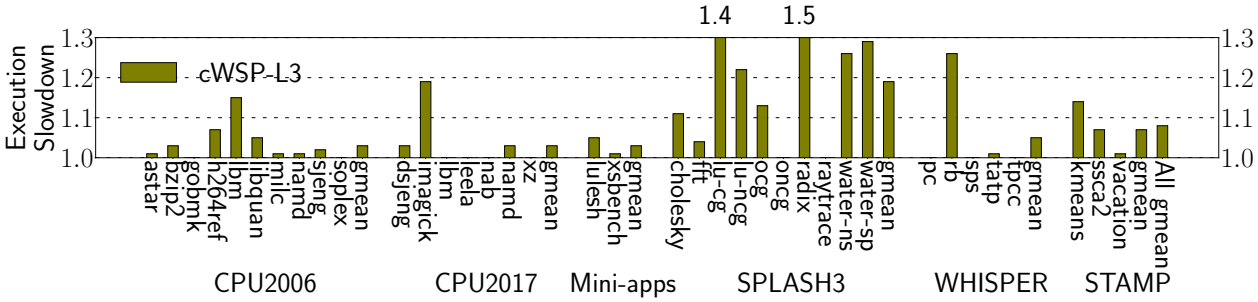


Figure 5.15. Normalized slowdown of PPA to the baseline when using L3 cache atop DRAM cache; lower is better

Sensitivity to WPQ Size: To see the impact of the NVM write pending queue (WPQ) on the performance of PPA, we vary the WPQ size from 8 to 24 for memory-intensive applications of CPU2006/Mini-apps and multi-threaded applications. As shown in Figure 5.16, PPA still incurs a low overhead (8%) though the WPQ size decreases to 8. This is because many applications exhibit high L2 write miss rates indicating already high pressure on the WPQ for the baseline. As such, the negative effect of extra write traffic caused by PPA's store writeback is amortized. Note that PPA incurs a higher overhead for some applications, e.g., `rb`, `water-ns`, and `water-sp`., as setting WPQ size to 8. The reason is two-fold: (1) they have low L2 miss rates indicating low run-time execution time for the baseline; (2) the store writeback leads to high pressure on WPQ due to more generated write traffic to it. Fortunately, the extra write traffic can be absorbed by enlarging the WPQ size to the default (16).

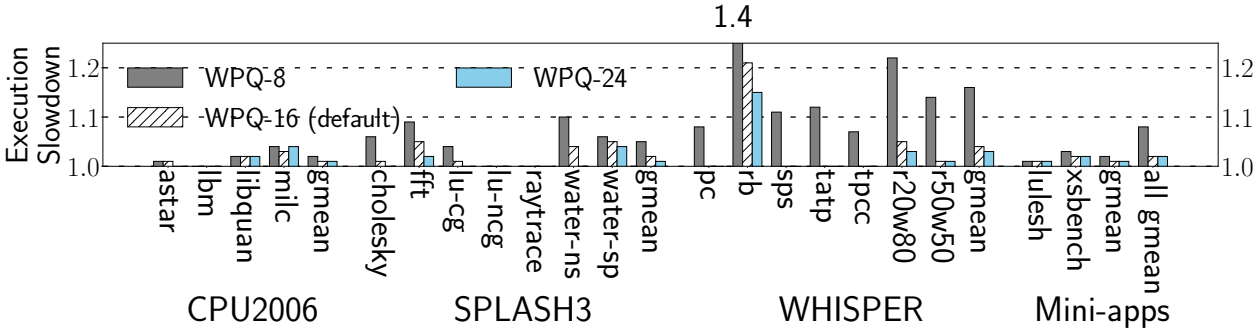


Figure 5.16. PPA's normalized slowdown varying WPQ size from 8 to 24; lower is better

Sensitivity to PRF Size: To show how PRF size affects PPA's performance, we vary the PRF size from 80/80 to 280/224 (integer/floating-point PR count). As shown in Figure 5.17, PPA incurs less overhead with a larger PRF. Note that even with the smallest PRF size of 80/80, PPA still forms sufficiently long regions and thus incurs an average of only 12% overhead owing to the underutilization of the PRF size. Interestingly, the benefit of the large PRF diminishes once its size increases beyond the default. This is because the default PRF setting already has enough amount of free registers to form long regions covering the persistence latency. Notably, with PRF size 80/80, PPA incurs about 30% run-time overhead for some programs, e.g.,`hmmer`, `lbm`, `lu-cg`, and `tpcc`, since (1) PPA requires at least 65/68 integer/floating-point registers for their normal execution, and (2) the programs have intensive memory writes, ending up with putting high pressure on the PRF.

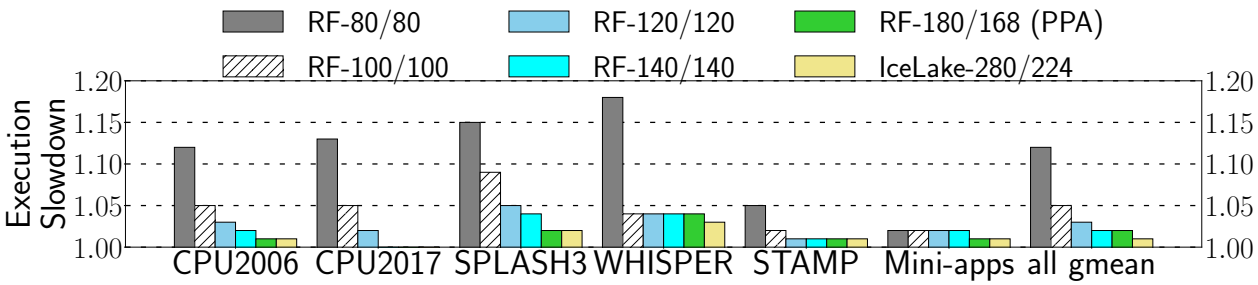


Figure 5.17. PPA's normalized slowdown varying RF sizes; lower is better

Sensitivity to CSQ Size: To investigate the proper size of the CSQ, we vary the CSQ size from 10 to 50. As shown in Figure 5.18, the CSQ size has a minimal impact on PPA's performance since there are an average of only 18 stores in each region (see Figure 5.14). In light of this, we set the CSQ size to 40 by default such that the core pipeline encounters as less pipeline stalls as possible caused by the CSQ overflow; it is cheap to enlarge the size of the CSQ to 40 because of its simple structure.

Sensitivity to PMEM Write Bandwidth: To show how PMEM write bandwidth affects PPA's performance, we vary the NVM write bandwidth from 1GB/s to 6GB/s for memory-intensive CPU2006/Mini-apps, SPLASH3, and WHISPER benchmarks. To be practical, PPA sets the default bandwidth to 2.3GB/s according to the empirical Intel PMEM analysis [338]. As shown in Figure 5.19, PPA still incurs an average of only 7% overhead even

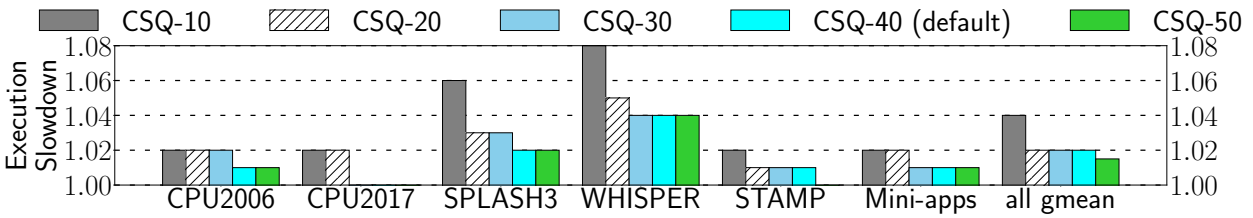


Figure 5.18. PPA's normalized slowdown with varying CSQ size; lower is better

for 1GB/s write bandwidth. Once the write bandwidth goes up beyond the default, PPA keeps its performance overhead as low as 2% thanks to the long regions hiding the potential pipeline stalls upon full WPQ. It is worth noting that PPA incurs a relatively higher overhead for SPLASH and WHISPER programs with 1GB/s bandwidth. This is because different threads of these multi-threaded applications always compete for the shared WPQ and the lower bandwidth exacerbates the competition. Note that some applications, e.g., `water-ns`, `water-sp`, and `rb`, are more sensitive to the write bandwidth due to their inherent less memory writeback traffic (i.e., they exhibit high locality).

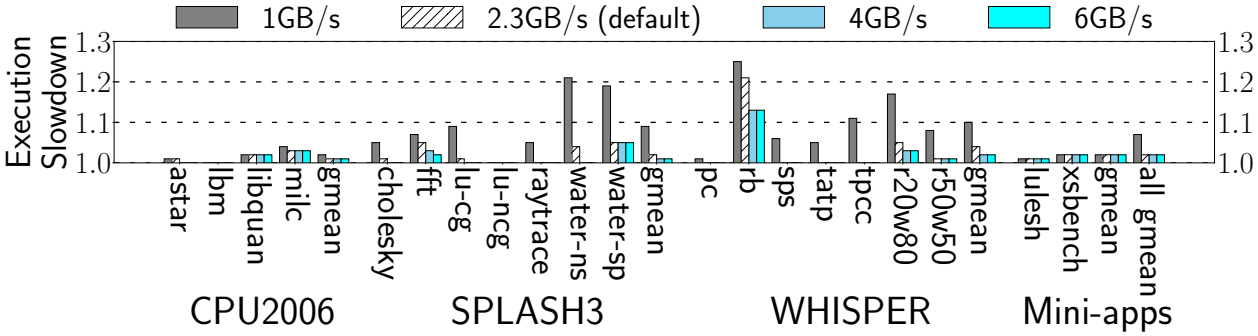


Figure 5.19. Normalized slowdown of PPA with varying NVM write bandwidth; lower is better

Sensitivity to Thread Count: To study the impact of PPA on cache coherence, we vary the thread count and scale up the NVM WPQ/shared L2 size proportionally. Figure 5.20 shows that the resulting performance impact is quite small; PPA still maintains high performance, i.e., an average of 2%–6% overheads for 8–64 threads. PPA incurs slightly

higher overheads for `water-ns`, `water-sp`, and Memcached (`r20w80`) with more threads due to the increasing stall cycles taken for thread synchronization.

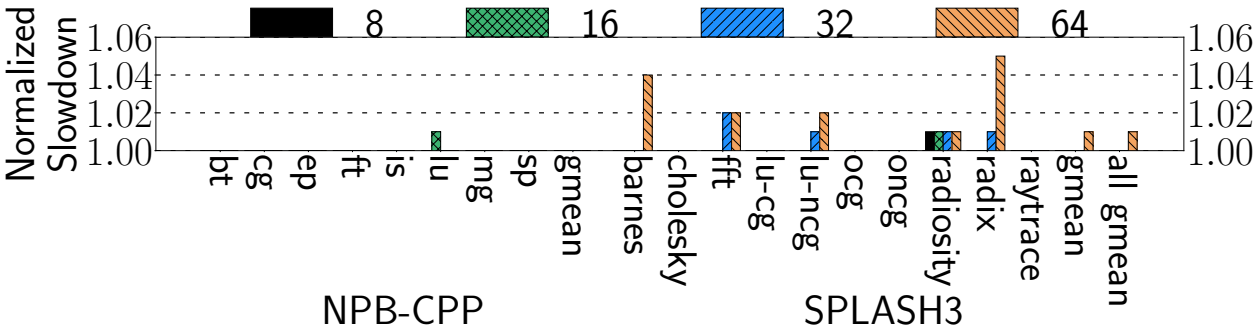


Figure 5.20. Normalized slowdown of PPA with varying thread count from 8 to 64 for multi-threaded apps; lower is better

5.6.7 Hardware Cost Analysis

PPA introduces a 64-bit LCPC register, a 348-bit vector register MaskReg due to the PRF size (348), and a 40-entry CSQ. Each CSQ entry records a pair of 9 ($\lceil log_2^{348} \rceil$)-bit index to a physical register and a 48-bit physical address. To facilitate JIT checkpointing, we round the size of PPA's proposed structures to the nearest multiple of 8 bytes such that their entry size is 8 bytes. We then use these numbers to calculate their hardware overheads (see Table 5.4).

Table 5.4. PPA's hardware overheads

	Area (μm^2)	Access Latency (ns)	Dynamic Access (pJ)
64-bit LCPC	12.20	0.057	0.00034
384-bit MaskReg	74.03	0.067	0.00029
40-entry CSQ	547.84	0.07	0.00025

We use CACTI 7.0 [345] to estimate the hardware cost of PPA's proposed hardware structures with a 22 nm process technology node. Table 5.4 showcases PPA's low hardware costs in terms of chip area, access latency, and power consumption. In summary, PPA's proposed hardware structures only occupy 0.005% chip area of an Intel Skylake core (11.85 mm^2 after excluding its shared L2 cache); the core area size is calculated with McPAT [346].

5.6.8 Energy and Latency for JIT Checkpointing

Upon impending power loss, PPA checkpoints CSQ, LCPC, CRT, MaskReg, and a part of PRF marked by entries of CSQ or CRT in NVM. We assume 16 architectural integer registers and 32 architectural floating-point registers. Therefore, we need to checkpoint at most 88 physical registers (40 in CSQ and 48 in CRT).

Energy Consumption: We assume that the checkpointed hardware structures are based on SRAM. To estimate the energy consumption, we leverage prior work [287, 347, 348]. They measure the energy cost per memory operation by using an external power meter while executing carefully designed microbenchmarks. These microbenchmarks are used to observe the energy consumption of only data movement between core and memory and minimize the impact of other architectural optimizations and non-memory operations. It turns out that 11.839 nJ/byte is necessary for accessing data in SRAM cells and moving it from core to NVM. Therefore, we need to secure 21.7 μJ to JIT checkpoint 1838 bytes of data considering the worst case that each physical register has 128-bit data. However, the ideal PSP scheme BBB [287] and Intel's eADR require a supercapacitor of 775 μJ and 550 mJ, which are 36.5 and 25943x larger than ours, respectively.

Table 5.5. Comparison of Energy requirement for JIT flushing

	PPA (WSP)	Capri [60] (WSP)	LightPC [349] (PSP)
Energy Consumption	21.7μJ	0.6mJ	189mJ
Volume (SuperCap/Li-thin)	0.06 mm^3/0.0006 mm^3	1.57 mm^3/0.016 mm^3	527.8 mm^3/5.3 mm^3
Ratio to Core Size (SuperCap/Li-thin)	0.005/ 5×10^{-5}	0.14/0.0014	44.5/0.45

We leverage the prior work [287] to calculate the required size of supercapacitor [350] and Li-thin battery [351]. These two battery techniques have an energy density of $10^{-4}Wh/cm^3$ and $10^{-2}Wh/cm^3$, respectively. Table 5.5 shows that PPA needs a 0.06 mm^3 supercapacitor or a 0.0006 mm^3 Li-thin battery, which occupies 0.5%/0.0005% of an Intel server core (11.85 mm^2), respectively.

Checkpointing Time: PPA's JIT-checkpointing controller can persist 8 bytes of data per cycle thanks to its simple structure (see Section 5.3.5). According to our RTL synthesis results with TSMC 22 nm technology, the controller only requires 144 D flip-flops with 88 two-input logic gates. Therefore, the controller takes 114.9 ns to read 1838 bytes of data. Along with the write bandwidth (2.3GB/s) of PMEM [352] in our simulations, PPA needs only 0.91 μs to flush the 1838 bytes data to PMEM upon power failure.

Comparison of Energy Consumption: We calculate the energy consumption of a single core equipped with WSP Capri or PSP LightPC [349] to highlight the low energy requirement of PPA. Upon power failure, Capri flushes data in its battery-backed redo buffers (54KB per core) to NVM with 11.839 nJ per byte [287], thus costing 0.6mJ per core. Likewise, LightPC flushes volatile data of *only* user processes in architectural registers (4224 bytes of 16 GPRs and 32 XMM registers), L1D cache (64KB), and L2 cache (16MB) all the way to NVM, leading to a high energy consumption of 189 mJ; LightPC uses PCM as main memory.

5.7 Other Related Work

Many prior PSP schemes [24, 25, 57, 119, 248, 286, 287, 302, 331, 349, 353–363] have offered user program persistency with crash consistency guaranteed. However, they require substantial programming burden in that users have to understand the underlying memory persistency model [118] and carefully write the code with crash consistency in mind. Moreover, the schemes often cause high run-time overhead (software approaches [58]) or significant logic complexity (hardware approaches [364]).

To this end, Narayanan et al. [110] propose the first WSP that flushes all volatile data, e.g., architectural registers/caches/DRAM contents, to NAND flash storage upon an impending power outage. Unfortunately, the just-in-time (JIT) checkpointing of all the data requires a considerable amount of energy to be secured always, which is in need of an expensive uninterruptible power supply (UPS). To lower the energy consumption, Capri [60] proposes a crash consistency mechanism based on hardware-managed redo buffers that only require a capacitor for their JIT checkpointing. In particular, Capri compiler partitions the input program into a series of recoverable regions with so that their stores never overflow the

buffer. During the region execution, Capri persists the data being stored in the region by moving them from the redo buffer to NVM through the non-temporal path [332], bypassing the cache hierarchy completely. However, Capri still suffers expensive chip area/energy overheads due to per-core capacitor-backed redo buffer (each requiring 54 KB). On the other hand, ReplayCache [245], another WSP scheme for EHS, incurs high run-time overhead with the frequent pipeline stalls at the end of compiler-formed store-integrity regions.

In summary, the overheads of the prior WSP schemes are so significant that they cannot enable a lightweight yet performant WSP. With the store integrity implemented using the simple register renaming trick, PPA achieves high-performance WSP for all at a negligible hardware cost. As shown in Table 5.6, PPA outperforms all prior WSP schemes in terms of all comparison criteria.

Table 5.6. Comparison of PPA to prior WSP approaches

	WSP [110]	Capri [60]	ReplayCache [245]	PPA
Hardware Complexity	Extremely High	High	No	Low
Energy Requirement	Extremely High	High	Low	Low
Recompilation	No	Yes	Yes	No
Transparency	Yes	Yes	Yes	Yes
Enable DRAM Cache	Yes	Yes	No	Yes
Enable Multi-MCs	Yes	No	Yes	Yes

5.8 Summary

This chapter details PPA, the first microarchitectural approach to WSP. As a basis for crash consistency and lightweight WSP, PPA realizes so-called store integrity in the out-of-order core pipeline. That is, PPA prevents store registers from being overwritten and dynamically partitions program to a series of regions whose boundary is delineated when the physical register file runs out. Upon impending power failure, PPA checkpoints the minimal architectural states including the preserved store registers using a tiny capacitor. When power comes back, PPA restores the checkpointed states, replays (persists) the stores

of the power-interrupted region, and resumes the program following the latest committed instruction before the failure. Experimental results with 41 applications highlight the benefits of PPA causing only a 2% average run-time overhead and 0.005% chip areal cost. We believe that PPA lays the foundation for WSP and pave the way to realizing it for all.

6. CWSP: COMPILER-DIRECTED WHOLE-SYSTEM PERSISTENCE

Nonvolatile memory (NVM) [252–256, 259–261, 331, 365] technologies have been deemed an alternative to DRAM thanks to their irresistible features, e.g., nonvolatility, byte-addressability, lower cost per bit, and near-zero standby power. They are now commercialized by many vendors, e.g., Intel Optane persistent memory (PMEM) [108], Everspin STT-MRAM [366], and Fujitsu ReRAM [367]. Considering this, many cloud service providers and national labs—such as Microsoft Azure [102] and Argonne National Lab's Aurora [368]—already equip their server fleets with PMEM as a key to offering data-intensive workloads sufficient memory [104, 369–373]. However, indiscriminately replacing DRAM with PMEM incurs significant performance loss in that PMEM is slower than DRAM. According to Peng et al. [103], PMEM leads to 2-18x slowdown compared to DRAM for their graph benchmark applications.

Thanks to the emerging cache-coherent CXL (Compute eXpress Link) [374] technology, which offers high-bandwidth and low-latency interconnect based on PCIe interface, it is now practically possible to mitigate the performance issue of NVM. This is because CXL can enable a deeper and wider memory hierarchy at low cost. For example, local DRAM can serve as a last-level cache (LLC) positioned between the conventional L3 cache and the CXL-enabled low-tier PMEM, which is akin to Intel PMEM's *memory mode* [108] where DRAM acts as an LLC atop PMEM main memory. Adopting such deep cache hierarchy effectively lowers the chance of accessing slow NVM, making its performance drawbacks more tolerable. The upshot is that users can benefit from NVM's enticing features, such as nonvolatility, with a minimal impact on run-time performance.

Figure 6.1 illustrates the normalized execution time of using CXL PMEM compared to that of CXL DRAM for memory-intensive applications with 4 different cache hierarchies[1]: (1) 2-level caches comprised of 64KB 8-way L1 data cache with 4-cycle hit latency and 1MB 8-way L2 with 14-cycle hit latency; (2) 3-level caches by adding a 16MB 16-way L3 with 44-cycle hit latency; (3) 4-level caches by adding a 128MB 16-way L4 with 82-cycle hit latency [375]; (4) 5-level caches by adding a 4GB direct-mapped DRAM cache. The clear trend in

[1]↑Other architectural parameters are listed in Section 6.8.

the figure is that the performance loss gradually drops from 2.14x to only 1.34x along with the deeper hierarchy [375, 376]. This trend implies that the performance loss of NVM would be ignorable for the future deeper memory hierarchy enabled by CXL. More importantly, big data applications still benefit from CXL-enabled NVM owing to its high density, thereby maintaining their performance.

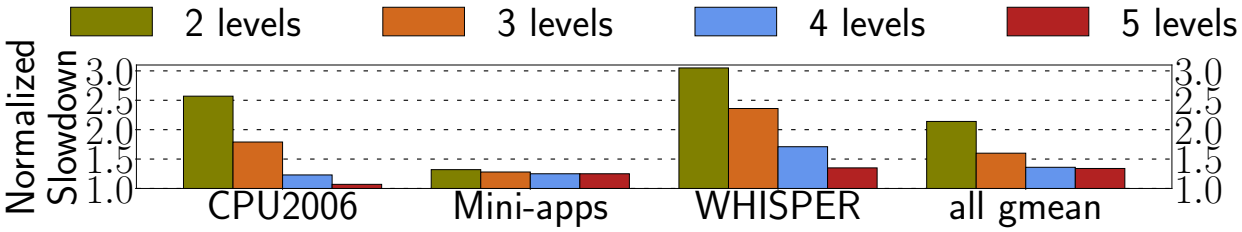


Figure 6.1. Normalized slowdown of CXL PMEM main memory to CXL DRAM main memory with varying levels of caches

However, the naive use of NVM can lead to a crash inconsistency. To illustrate, suppose users try to insert a new node to the beginning of a doubly-linked list, which is done by (1) setting the new nodes next pointer to the address of the old head node and (2) resetting the old head nodes previous pointer to the address of the new node. Now, assume a scenario where the second store persists in NVM with the cacheline evicted from LLC while the first store is cached. If power is suddenly cut off here, the data stored in the cache is lost. This causes the new nodes next pointer to become a dangling pointer, leading to inconsistent NVM states.

Considering this, Intel proposes eADR [377] to preserve the contents of volatile caches across power failure by just-in-time (JIT) checkpointing them to NVM right before the failure. Unfortunately, eADR requires a constant and high energy supply for the JIT checkpointing process. This becomes unsustainable, especially with the growing LLC size, e.g., a 384MB L3 in the AMD EPYC 9654P CPU [378]. Even worse, eADR falls short in covering other volatile components, e.g., registers and DRAM—at terabyte scale [379], which is the reason why eADR cannot guarantee crash consistency at all.

Thus, to make program persistent with NVM, users should resort to partial-system persistence (PSP) where NVM gets a separate address space next to DRAM's main memory

space as with Intel PMEM's *app-direct mode* [20–25, 59, 117–122]. However, *PSP faces 4 challenges: (1) run-time or hardware cost, (2) difficulty of enabling DRAM cache, (3) programming burden, and (4) vulnerability to bugs.* First, software-based PSP schemes [24, 57, 58] require the insertion of persist barriers—e.g., `clwb` and `sfence` in x86—to flush the data stored to NVM address space, while hardware-based schemes [118, 120, 248, 307] accelerate the store persistence using costly architectural support. Second, since DRAM serves as main memory, both PSP schemes cannot afford to exploit DRAM as the last-level cache (LLC)[2], thereby losing the performance benefit of the DRAM cache. Third, users are burdened with rewriting data structures with memory persistency [113] in mind, often leading to designing custom recovery logic for their crash consistency. That is why persistent programming is generally hard and error-prone [279, 303, 321–323]. Last, PSP requires a special memory allocator such as `pmalloc` [275], introducing the potential risk of persistent memory leaks. This renders already error-prone persistent programming even more complex and buggy [265, 277, 278, 280–282, 380, 381].

Given PSP's deficiencies, the interest in whole-system persistence (WSP) [60, 110, 232] is growing in both academia and industry. Since NVM serves as main memory in WSP allowing DRAM to be repurposed as LLC without hassle, WSP can enable deeper cache hierarchy and achieve high performance. Moreover, WSP transparently ensures the store persistence and the crash consistency for all kinds of program. However, WSP faces skepticism due to its complicated hardware design and substantial energy requirements [110] for flushing all the volatile states to NVM before impending power failure. The state-of-the-art WSP solution, Capri [60], addresses the skepticism to some extent but is still deemed impractical for several reasons: (1) significant storage overhead of 54KB per core for Capri's hardware buffers; (2) high amount of energy for JIT-checkpointing the buffers without power interruption; (3) complex hardware loggings and their demand for extremely high bandwidth of persist data path; and (4) inability to efficiently guarantee crash consistency for multiple memory controllers (MCs), and so on.

[2]↑Unless additional hardware support is devised to use a portion of DRAM as cache leaving the rest for main memory.

To this end, this chapter presents cWSP, a synergistic compiler/architecture co-design to achieve lightweight yet performant WSP. The key idea is that cWSP can recover potentially inconsistent NVM states by re-executing a small portion of code. As such, cWSP compiler partitions any program including operating systems (OS) and runtime libraries—as long as they can be translated into LLVM bitcode [139]—into a series of idempotent regions (epochs) [116]. Since they are designed to be free of memory antidependence, they can be re-executed multiple times yet still generate the same correct output. This allows cWSP to resume program from the beginning of the power-interrupted region, i.e., the end of the most recently persisted region. The takeaway is that cWSP obviates Capri's expensive hardware buffers and their JIT checkpointing while maintaining high performance.

Another prior work iDO [57] also leverages idempotent processing for power failure recovery. However, iDO works for only user applications and slows them down significantly; the reason is twofold: (1) iDO compiler generates superfluous memory writes to NVM which has limited write bandwidth and high write latency; (2) iDO causes the core pipeline to stall at the end of each region because 2 persist barriers are inserted before and after the region boundary. In contrast, cWSP addresses these issues with 2 pillars: (1) its compiler optimization eliminates unnecessary memory writes; (2) its hardware enables asynchronous store persistence, allowing the core pipeline to execute other instructions while persisting previous stores. In particular, cWSP persists the 8-byte data being stored to NVM through a FIFO persist path—built on Intel's existing non-temporal data path [112] with its write-combining buffer (WCB) disabled—immediately after the store instruction is committed. In this way, cWSP enables fast store persistence and lowers the bandwidth requirement for the persist path by 8x compared to all prior work relying on 64-byte data persistence [25, 118, 120, 248, 287, 382].

Last but not least, cWSP introduces a novel concept of *memory controller speculation*, aiming to efficiently ensure crash consistency in the presence of multiple memory controllers (MCs) that have non-uniform memory access (NUMA) time. This pivotal feature distinguishes cWSP from prior schemes [60, 118, 248, 307, 354, 382, 383]. They conservatively wait at each region (epoch) boundary for previous stores to persist in case the NUMA leads to a reordering of stores across regions and the resulting crash inconsistency on power failure

in-between. In contrast, cWSP assumes power failure is unlikely between regions and *speculatively* persists the stores of the subsequent regions with no stall at their boundaries. As a safe net, the MCs undo-log the stores upon their arrival to handle potential misspeculation (i.e., power failure occurred). Upon misspeculation, cWSP reverts the speculative NVM updates with undo logs to maintain consistent NVM states across the power failure.

The experimental results with 37 applications from SPEC CPU2006/2017 [140, 141], Mini-apps [218, 219], SPLASH3 [217], WHISPER [307], and STAMP [384] demonstrate that cWSP incurs an average of only 6% run-time overhead and a storage overhead of 176 bytes, making cWSP highly suitable for implementation on silicon, whereas the state-of-the-art WSP work incurs a 27% run-time overhead despite its significant hardware overheads. In summary, cWSP makes the following contributions:

- cWSP is the first approach to a lightweight yet performant WSP—only a storage cost of 176 bytes (346x reduction of the state-of-the-art work's 54KB)—while supporting multiple MCs efficiently.
- cWSP eliminates the expensive yet power-hungry JIT checkpointing in prior approaches thanks to the intelligent compiler/architecture co-design.
- cWSP works well even for future CXL-based deeper and wider memory hierarchy—Section 6.8.3 shows that cWSP incurs only a 4% run-time overhead for memory-intensive applications running on CXL-enabled NVM.
- cWSP provides a complete compiler toolchain—based on Clang/LLVM 13.0—that can rebuild the entire Linux software stack with crash consistency ensured.

6.1 Background and Challenges

6.1.1 Persist Path and Stale Read Issue

Prior PSP schemes [25, 118, 120, 248, 287, 307] utilize the existing non-temporal path [112] as a dedicated persist path of NVM stores—and thus drop their dirty cacheline evictions from LLC—to deliver the data to NVM in order, achieving *strict persistency [113]*. Here,

data merged on L1 data cache are also placed on the persist path so that it directly transfers the data to NVM, bypassing the lower-level caches and thus avoiding costly persist barriers. However, these schemes come with some challenges. First, they demand a *high-bandwidth* persist path, which is not always feasible, as they persist a 64-byte cacheline to NVM for every 8-byte store merged into L1 data cache. Moreover, they delay the persistence of a store until it is merged into L1 data cache, significantly impacting the performance, especially given the high L1 data cache miss rate—22% for `470.lbm` of CPU2006 in our simulation.

More importantly, the use of the persist path without caution can cause a *stale read issue* [248], leading to wrong program output due to the lack of ordering guarantees between the persist path and the regular (cache) data path. Figure 6.2 (a) and (b) show how the issue occurs; in this example, `str 100,[A]`; `str 200,[A]`; and load `ldr r0,[A]` all access the same memory location A. Here, the two stores are merged into the same cacheline—which is later silently dropped from LLC as in prior schemes [118, 120, 248, 287, 307]—while the two memory updates are sent to NVM through the persist path. Suppose the cacheline is dropped (①) from LLC before `str 100,[A]` and `str 200,[A]` persist because of congestion in the persist path. If the core pipeline encounters an LLC miss for the load when only the first store has persisted (②), then the load ends up reading an outdated value from NVM—instead of the up-to-date value of 200.

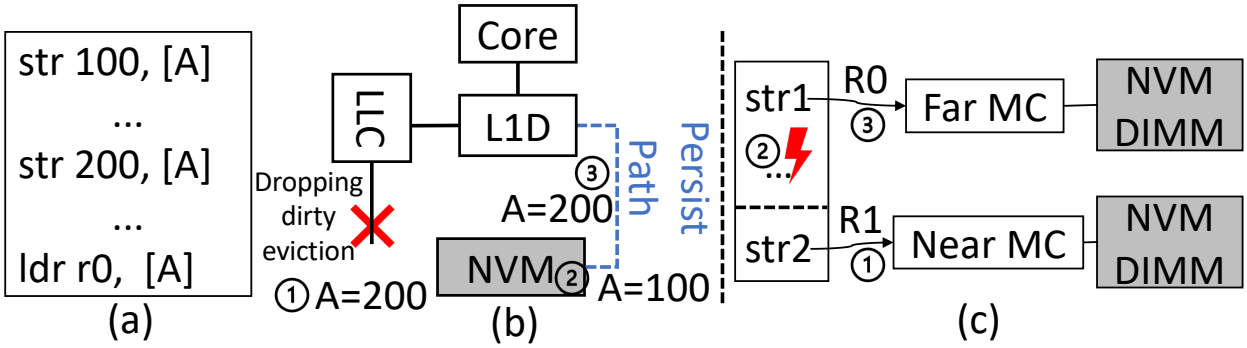


Figure 6.2. (a) Original assembly code; (b) stale read issue occurred; (c) crash inconsistency for multiple MCs

To address the issue, prior work such as BBB [287] and DPO [118] include front-end persist buffers (PBs)—containing the stores for the example shown in Figure 6.2 (a)—in the

cache coherence domain. That way, the load in Figure 6.2 (a) reads the up-to-date data from the PBs. However, the prior work significantly complicates the already complex cache coherence protocol. On the other hand, other prior work like HOPS [307] delays the loads missing in LLC—in case the up-to-date data are pending on the persist path—until they persist in NVM. For this purpose, HOPS requires a bloom filter near memory controllers to check if the data are pending. The implication is that every NVM store must pay for long latency to access the backend bloom filter, which might hurt the performance. Either way, these prior schemes come with notable overheads, rendering them unsuitable for lightweight yet performant WSP targeting the CXL-enabled deeper memory hierarchy.

6.1.2 Multiple Memory Controllers and Crash Inconsistency

The presence of multiple memory controllers (MCs) in a server system [385] poses a daunting challenge in maintaining the FIFO ordering of the persist path, which is the basis for crash consistency, though they enable large memory space. As shown in Figure 6.2 (c), due to non-uniform memory access across MCs, a younger store—e.g., `str2` in the 2nd region R1—could persist in NVM (①) before an older one (③) if they are destined to different MCs. This can cause inconsistent NVM states when power failure occurs in between (②). To address this issue, many prior proposals [60, 118, 248, 307, 382, 383] simply wait at region boundaries for prior stores to be persisted, thus degrading the performance.

6.1.3 Region-Level WSP with Persist Path

Capri [60], the state-of-the-art WSP scheme, addresses the challenges of persisting stores with compiler/architecture co-design for a wide range of applications[3]. Capri relies on a hardware-managed redo buffer [301, 304, 386] as the basis for crash consistency. For this reason, Capri compiler partitions input program into a series of recoverable regions with the buffer size in mind, preventing the buffer overflow during region execution and ensuring correct power failure recovery.

[3]↑We get Capri compiler's source code from authors and figure out that it cannot compile runtime libraries though it covers the OS and user code.

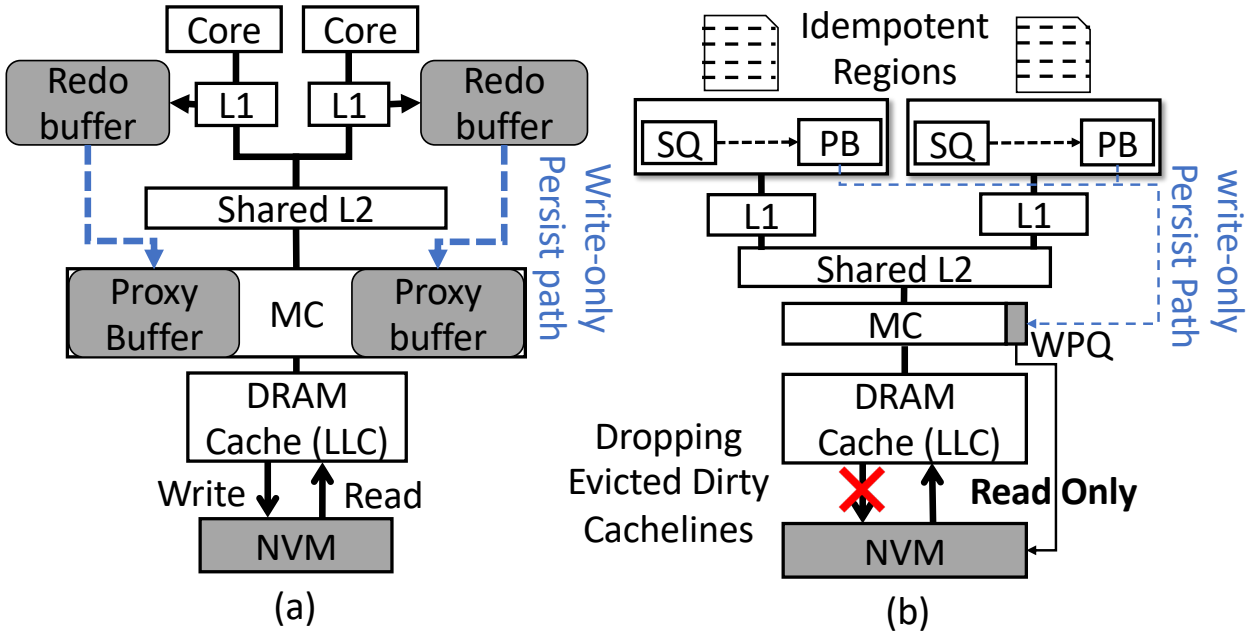


Figure 6.3. (a) Capri architecture for PMEM memory mode; (b) cWSP architecture for PMEM memory mode; shaded boxes are in persistence domain; round boxes are newly proposed by either architecture; the thin persist path of cWSP indicates its lower bandwidth requirement

Figure 6.3 (a) shows Capri's high-level architectural diagram. During region execution, Capri copies the dirty cachelines touched by the region's stores to the redo buffer—next to the L1 data cache as shown in the figure. Each cycle, Capri attempts to transfer the data in the redo buffer to NVM through the persist path. For failure-atomic region persistence, Capri employs a 2-phase approach. It moves the redo buffer entries to a battery-backed proxy buffer—managed by the memory controller—and then from there to NVM media. The 2-phase persistence ensures that either the proxy buffer or NVM remains intact across power failure.

However, such a redo buffer approach forces the CPU pipeline to stop at each region end until all its buffered stores are moved to the persistent domain (proxy buffer) before preceding to the next region, causing significant slowdown. To solve this issue, Capri lets the redo buffer battery-backed as well. This allows the next region to immediately start in that the prior one has buffered its stores already in the persistent domain (redo buffer).

6.1.4 Limitations of Prior Work

Capri faces 5 issues, rendering its practical use impossible. First, its hardware buffers incur a high storage cost, totaling (N + 1) × M × 18KB, where N corresponds to the memory controller count, M to the core count. For example, Capri results in a storage overhead of 88MB for AMD 128-core EPYC 9754 processors with 12 MCs [385]. Second, Capri requires a considerable amount of energy at all times for JIT checkpointing, leading to battery maintenance burden and environmental impact [387–389], while power failure is scarce in server fleets. Third, Capri relies on an over-complex redo+undo logging to recover potentially inconsistent NVM status by using undo or redo logs depending on where the 2-phase persistence is power-interrupted. This complex hardware logging scheme ends up amplifying NVM writes by 8x and demands an extravagant bandwidth for the persist path.

Forth, Capri incurs extra hardware cost for resolving the stale read issue. That is, Capri delays DRAM cache eviction to scan the proxy buffer and invalidate the matched proxy buffer entry of the same address. Even if no matching entry is found, which is a common case, Capri cannot release the DRAM cache eviction. That is because the data being matched might be pending on the persist path; therefore, Capri should wait for the worst-case data delivery latency in case the data is to be found within the latency [60]. Finally, Capri causes a high run-time overhead for server-class cores with many MCs [385] due to frequent persistence stalls at the end of short regions—29 instructions in regions on average.

6.2 Design

Figure 6.3 (b) shows the architectural diagram of cWSP. In particular, cWSP's persist path connects each core to MC unlike Capri's starting from L1 data cache. As will be shown in Section 6.8.3, cWSP works well for CXL-based NVM, in which case the persist path ends at CXL Home Agent [374], though cWSP assumes the less complex Intel PMEM memory mode by default for a fair comparison to the state-of-the-art work Capri.

6.2.1 Region-Level Crash Consistency for All

To achieve crash consistency for the entire Linux software stack, cWSP compiler is capable of partitioning any C/C++ program including the OS kernel into a series of idempotent regions [42, 57, 116, 130, 133, 215], that are free of memory antidependence also known as write-after-read dependence, serving as the basis for *recovery-via-resumption*. Similarly, cWSP also ensures crash consistency for C/C++ libraries and the Linux kernel by partitioning their functions such as `malloc` and `sbrk`; see Section 6.3 for details.

6.2.2 Asynchronous Store Persistence

Unlike all prior work [24, 57, 60, 248, 287, 307, 354], cWSP *for the first time* decouples store persistence from cache access. That is, cWSP persists the data being stored as soon as the store is committed. To achieve this, cWSP repurposes Intel's write-combining buffer (WCB) as a volatile persist buffer (PB) that connects from store queue (SQ) to the memory controller (MC) as shown in Figure 6.3 (b). Each time a store is committed, its data is copied to the PB and then transferred to the MC along the persist path in the background. The implication is twofold: (1) cWSP persists stores at 8-byte granularity and thus brings an eightfold reduction in the persist path bandwidth, compared to the prior work based on 64-byte cacheline granularity; (2) cWSP exerts practically no pressure on the SQ, which would otherwise slow down the core pipeline execution. Further details are deferred to Section 6.4.1.

6.2.3 Memory Controller Speculation

To ensure high-performance crash consistency even in the presence of multiple memory controllers (MCs), cWSP proposes *memory controller speculation*. While the stores of a region are on their way to NVM locations, cWSP *speculatively* persists the following regions' stores with the data logged in NVM, despite non-uniform memory access across MCs, assuming power failure is unlikely in the meantime. cWSP leverages undo logging to enable in-place updates and avoid costly read redirection. If misspeculation (i.e., power failure) oc-

curs, cWSP reverts the speculative NVM updates using the undo logs, thereby maintaining consistent NVM states across power failure; details are provided in Section 6.4.2.

6.2.4 Power Failure Recovery Protocol

Since the memory controller speculation of cWSP allows multiple regions to be persisted concurrently, care should be taken to ensure correct power failure recovery. It is possible that these regions have at least some of their stores persisted before power failure. This paper calls such regions *unpersisted*. On the other hand, a region is called *persisted* only after its stores are all persisted.

In the wake of power failure, cWSP resumes the interrupted program in 3 steps with identifying a boundary between persisted and unpersisted regions: (1) reverting speculative NVM updates using undo logs; (2) preparing the inputs to the oldest *unpersisted* region, the entry of which serves as the recovery point; and (3) restarting the region from the beginning; details are found in Section 6.6.

6.3 cWSP Compiler and Runtime

6.3.1 Cutting Memory Antidependence

To partition program into a series of idempotent regions, it is a critical step to ensure the absence of memory antidependencies within each region. For this purpose, cWSP uses the same idempotent processing algorithm developed by De Kruijf et al. [116]. First, cWSP compiler treats function callsites and synchronization points—such as atomic operations and memory fences—as initial region boundaries. cWSP also inserts a region boundary at the header of each loop, forming a region per iteration; of course, extra boundaries are inserted in the loop body to split other memory antidependence therein. Second, cWSP compiler computes a set of cutting points for antidependence pairs of memory using LLVM's *alias analysis*. Later, cWSP compiler uses a hitting set algorithm to find out the best partitioning strategy. As Figure 6.4 (a) shows, a region boundary separates `r2 = ldr [r0]` and `str r1,[r0]` and keeps them in two separate regions.

6.3.2 Checkpointing Live-Out Registers

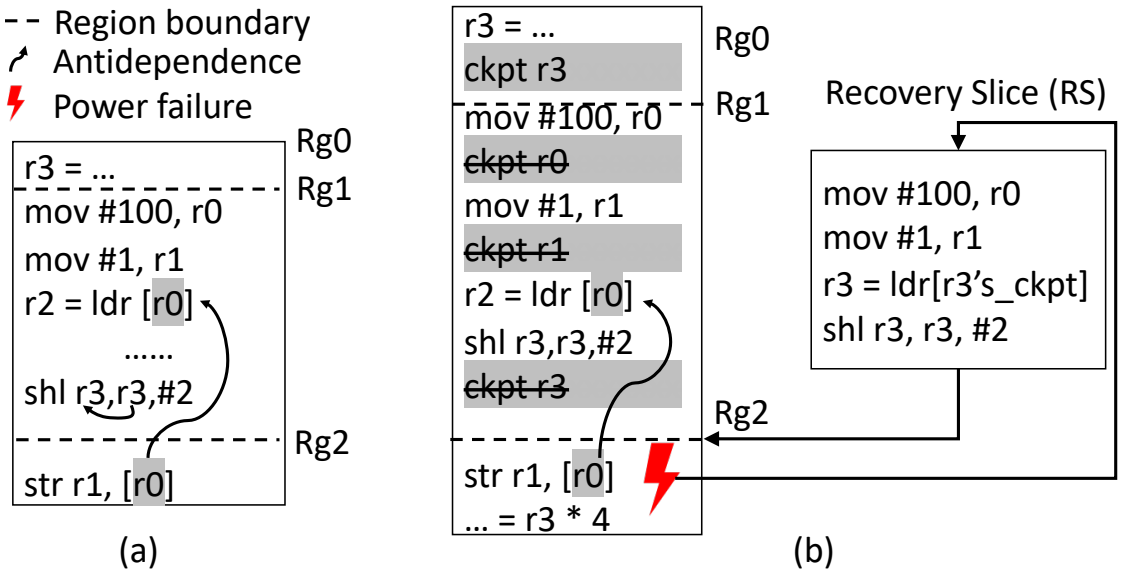


Figure 6.4. (a) Cut memory antidependence; (b) inserts the checkpoints for live-out registers and then prunes all 3 checkpoints in region Rg1; note that region Rg0/R1 are already persisted before power failure (⚡), whereas Rg2 is not

However, solely preventing memory antidependence within regions is insufficient to achieve WSP, as volatile registers lose their data upon power failure. To address this issue, cWSP compiler checkpoints (saves) registers to a designated storage in NVM, indexed by architectural registers and managed by cWSP hardware. cWSP compiler first calculates a set of *live-out* registers for each region using LLVM's *liveness analysis* and then checkpoints their values to NVM. Figure 6.4 (b) shows that `ckpt r3` is inserted in region Rg0 since `r3` is *live-out*—i.e., it is used by some later region(s).

6.3.3 Pruning Register Checkpoints

To mitigate the potential increase in write pressure on the persist path caused by inserted checkpoints (essentially store instructions), cWSP leverages the optimal checkpoint pruning algorithm of Penny [132]. We found out that this optimal checkpoint pruning, originally designed for soft error resilience, can efficiently eliminate redundant checkpoints without

compromising the crash consistency guarantee. The intuition behind the checkpoint pruning is that many checkpoints are unnecessary if they can be reconstructed using immediate values and/or the remaining checkpoints at recovery time. For example, all 3 checkpoints in region Rg1 are eliminated as shown in Figure 6.4 (b), improving the performance greatly (see Section 6.8.2). Across power failure (⚡) occurred in region Rg2, cWSP's recovery runtime first executes Rg2's recovery slice (RS)—on the right of Figure 6.4 (b)—to reconstruct the values of region Rg2's 3 live-in registers. As shown in the RB, $r0$ and $r1$ are reconstructed from 100 and 1, respectively, while $r3$ is done by (1) loading the value checkpointed in region Rg0 and (2) applying the shift instruction over the value. With these input registers restored, cWSP then resumes the interrupted program from the beginning of the region Rg2.

6.3.4 cWSP Runtime and Linux Kernel

Ensuring crash consistency for the entire software stack—covering user program, runtime libraries, and the Linux kernel—is crucial for the successful implementation of cWSP. However, this is not adequately addressed in previous approaches [60, 245, 277, 307], due to the lack of C library in LLVM community and the incompatibility between the Clang/LLVM compiler and the GNU C library `glibc` [390]. To overcome this obstacle, cWSP introduces a comprehensive crash-consistent runtime for the first time. We patch essential libraries, including `glibc`, LLVM C++ library `libcxx` [391], LLVM `compiler-rt` [392], and LLVM stack unwinding library `libunwind` [393]. In addition, we patch the configuration of `glibc` to allow for its compilation with cWSP compiler. In particular, all the assembly files pertaining to x86_64 are manually patched to insert region boundaries and checkpoints. It is also feasible to lift assembly code up to LLVM bitcode using mature lifting tools, e.g., Remill [394], in which case cWSP compiler optimizations can be automatically applied along with the recoverable region formation.

6.4 cWSP Hardware

6.4.1 Asynchronous Store Persistence

Preventing Stale Read Issue on the Cheap: Recall that the stale read issue arises on LLC load misses. That is because there is no ordering guarantee between the persist path and the regular path where LLC silently drops dirty cachelines on their eviction, though the data of committed stores move to both paths in our case. That is, there is a potential for a race condition [395] between (1) the read on the regular data path and (2) the write on the persist path. Fortunately, we found out that the stale read issue almost never occurs due to the faster persist path, i.e., data being read by those loads missing LLC are sure to have already persisted in NVM. The *load-after-persist* order is made most of the time in that the data carried over the persist path can directly head to NVM whereas they go through multiple levels of caches in the regular path.

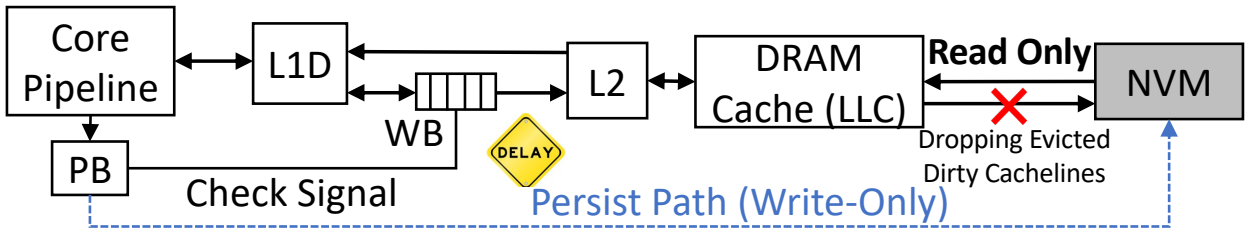


Figure 6.5. Solving stale read issue by delaying dirty cacheline writeback from the WB of the private L1D to the shared L2

Given the rare occurrence of the stale read issue, any possible solution must be lightweight enough to minimize the impact on the core pipeline execution. With that in mind, cWSP enforces the *load-after-persist* order occasionally, i.e., when the data is about to reach the shared L2 on the regular path. That is, cWSP only needs to ensure that no writeback is made to the L2 until the same data—once placed in the persist path—is eventually written to NVM. To achieve this, cWSP delays the writeback of dirty cachelines from the private L1D's write buffer (WB) to the shared L2, provided the persist path has not yet flushed the corresponding data to NVM. As shown in Figure 6.5, when a cacheline at the WB head is about to be drained, a check signal with the cacheline address is issued to search for a

matching entry in the PB. If found, cWSP holds the writeback of the head until the matched PB entry is persisted in NVM.

Coherence-Agnostic PB: The upshot of the above simple technique is that the PB is out of cache coherence domain, i.e., the entire caches and the coherence protocol both remain the same. That is because cWSP ensures a memory read always retrieves the up-to-date data either from the caches on their hits or from NVM when missing in the DRAM cache (LLC). Thus, accessing the PB for loads becomes unnecessary, while prior work [118, 287, 307] consults PB either directly from the core or through cache coherence requests for loads thereby complicating already complex cache coherence mechanisms.

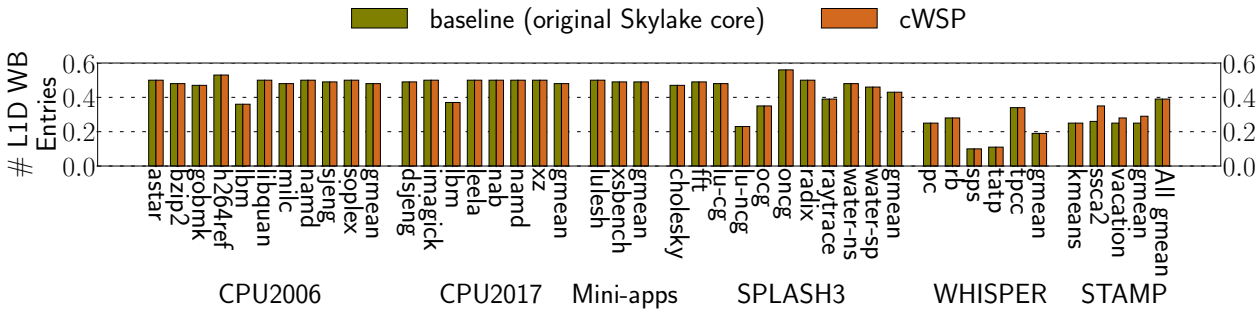


Figure 6.6. Average occupancy of the WB of L1 data cache for baseline and cWSP

At first glance, one might think that delaying the WB writeback could potentially slow down pipeline execution, especially when the WB is full and a new WB entry needs to be allocated for an incoming dirty eviction. However, our empirical evaluation shows that this delay has no adverse effect on performance at all (see Section 6.8.2). Figure 6.6 shows that both the baseline and cWSP maintain an average occupancy of only 0.39 WB entries, implying minimal pressure on the WB. This negligible impact can be attributed to two factors. First, the persist path is way faster than the regular path. When a dirty cacheline is about to be written back from the WB to L2, its corresponding PB entry is most likely persisted in NVM already, resulting in no matching found in the PB. Second, content-addressable memory (CAM) searching for the (50-entry) PB can complete in just 1 cycle ($0.5ns$), causing technically no delay on the WB writeback; this is supported by IBM's report on a $0.6ns$ CAM search time for a 64x72 CAM with $65nm$ CMOS technology [396].

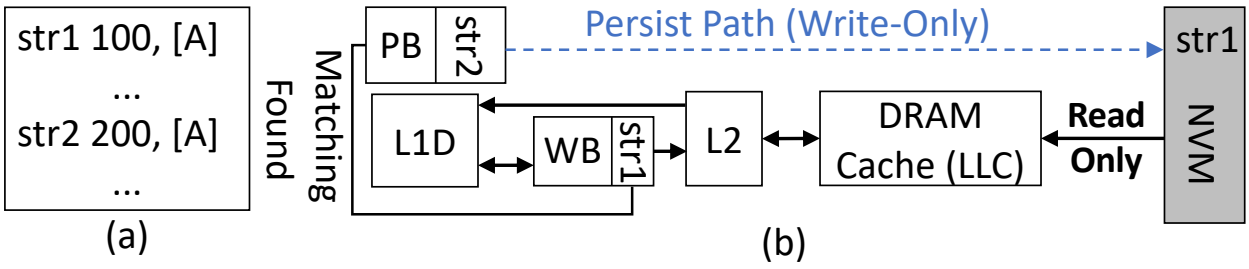


Figure 6.7. (a) assembly code; (b) a false positive

False Positives: Note that cWSP guarantees the absence of *false negatives* for the PB searching, since it creates a PB entry for each store before the data is merged into L1D. However, theoretically, cWSP could mis-identifies a PB entry though it does not collide with the WB head entry, causing false positives and unnecessary delay in the WB writeback. Nevertheless, we never observed false positives in our experimentation, owing to the huge speed gap between the regular path and the way faster persist path. Figure 6.7 illustrates a hypothetical false positive scenario. Consider two committed stores—*str*1 and *str*2—writing values 100 and 200, respectively, to the same memory address `A`. In this scenario, *str*1's data (100) has already been persisted in NVM via the persist path, and its corresponding dirty cacheline has been evicted from the L1 data cache to the WB. Meanwhile, the core pipeline allocates a PB entry for *str*2, which misses the L1 data cache. Here, if the WB is about to flush *str*1's dirty cacheline to the L2, *str*2's PB entry is mistakenly perceived as a match with *str*1 at the WB head, causing a false positive.

Lowering Persist Path Bandwidth at No Cost: Due to the 8-byte data granularity of the persist path, cWSP's write pending queue (WPQ) maintains 8-byte entries as well. Therefore, care must be taken to ensure correctness in that the memory system transfers data at a 64-byte granularity. If a load misses the LLC (DRAM cache) and encounters a WPQ hit, then it can only get the corresponding 8-byte WPQ entry failing to retrieve the remaining 56-byte data, in which case program correctness is broken.

To address this potential incorrectness without complex hardware support, cWSP simply postpones serving those loads hitting the WPQ until the matching WPQ entry persists

in NVM. In particular, this delay has no practical impact on performance owing to the remarkably low WPQ hit ratio. Figure 6.8 shows 0.98 hits per 1 million instructions. Such a low hit ratio has a twofold implication: (1) with an increasingly deeper memory hierarchy where fewer read requests reach the NVM, the WPQ hit ratio gets even lower; (2) cWSP effectively expands the WPQ's capacity by eightfold—compared to conventional WPQ whose entry size is 64-byte—without requiring additional storage. Consequently, cWSP is well-suited for the future deeper/wider memory hierarchy.

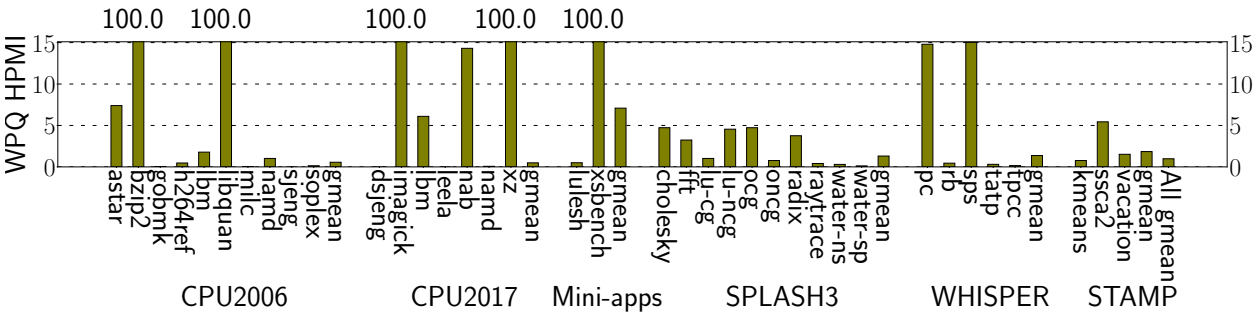


Figure 6.8. WPQ hits per 1 million instructions

6.4.2 Memory Controller Speculation

Store Persistence without Stalling at Region Boundaries: With multiple memory controllers (MCs), the stores of a younger region could persist before those of older regions due to the non-uniform memory access (NUMA) time of the MCs. As mentioned earlier in Section 6.1.2, this out-of-order region persistence breaks the FIFO nature of the persist path that serves as the basis for crash consistency. As such, prior schemes [60, 118, 248, 307, 382, 383] resort to stalling the core pipeline at each region boundary (transaction end) until every store of the region persists. That is, they do not allow inter-region persist reordering, leading to high performance loss server-class cores backed with many MCs [385].

To achieve correct yet performant crash consistency in the presence of multiple MCs, we make two observations. First, despite the NUMA effect, the resulting persist reordering within a region is not harmful, since it can be correctly recovered (re-executed) thanks to cWSP's idempotent region formation. So, we are only concerned about the inter-region

persist reordering that makes the idempotent recovery incorrect—since idempotence holds on a per-region basis. Second, nevertheless, the inter-region persist reordering can be alright, provided it is not caught by power outages. Even if they occur, cWSP's recovery runtime can leverage conventional logging to revert the out-of-region-order persists that might corrupt the input(s) to the oldest unpersisted region—being re-executed by the recovery protocol (Section 6.2.4); and the recovery cost should be insignificant given the rarity of power failure. Those observations inspire us to develop *memory controller speculation*, assuming that the oldest unpersisted region never encounters power failure. With that in mind, cWSP keeps speculatively persisting the data of the following regions, without waiting for the oldest to get persisted.

Note that the oldest unpersisted region is non-speculative and vice versa in that the prior one has already been persisted, whereas the following regions are under speculation. In case power failure interrupts the persistence of the oldest unpersisted region (i.e., misspeculation), cWSP undo-logs any data being speculatively persisted. This allows cWSP to restore the memory status to point where the oldest unpersisted region is about to start—for the correct re-execution of the idempotent region (Section 6.2.4).

To illustrate, consider 4 consecutive regions: Rg0, Rg1, Rg2, and Rg3. Suppose Rg0 has been persisted. Rg1 is currently the oldest unpersisted region, i.e., it is non-speculative, whereas Rg2 and Rg3 are under speculation. While Rg1 persists its data being stored in NVM, cWSP *speculatively* persists and undo-logs the data of Rg2 and Rg3, preparing them for potential reversal in the event of power failure. When the speculation turns out to be true, i.e., Rg1 has persisted all its stores without power interruption, Rg2 thus becomes non-speculative, which causes Rg2's logs to be deallocated. Here, Rg3 still remains speculative though.

To track speculation state and perform its corresponding actions, cWSP should recognize (1) if a region is speculative (or non-speculative) and (2) if it is persisted. For this reason, cWSP tracks two kinds of information, i.e., speculation and persistence metadata, for each region. As shown in Figure 6.9, cWSP prepares two FIFO queues[4]: the region boundary table (RBT) for the speculation metadata and the persist buffer (PB) for the persistence

[4]↑They have one read/write port and one search port to complete CAM searching in one cycle.

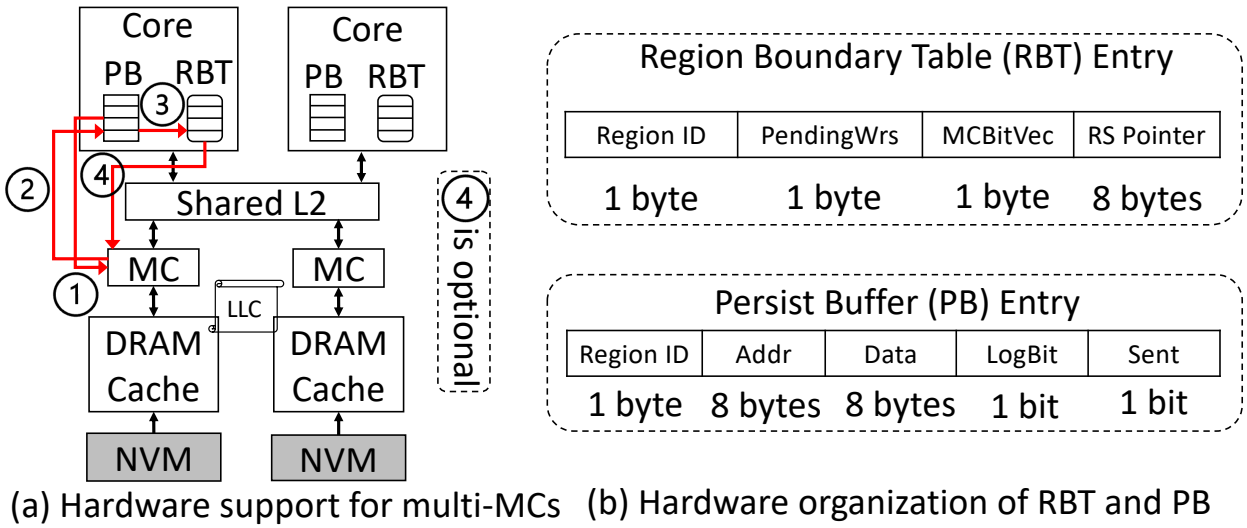


Figure 6.9. Hardware organization for MC speculation; RBT is newly proposed, while PB is built on Intel WCB

metadata, respectively. At a high level, each region is treated speculative upon entry into the RBT and remains so until it moves to the RBT head that always points to a non-speculative region. The RTB head entry is removed as soon as its corresponding region (i.e., the oldest unpersisted one) is persisted. The implication is that RBT size determines the number of speculative regions.

Specifically, when the core pipeline commits a region boundary instruction, cWSP allocates an RBT entry for the current region being started. The RBT entry contains 4 items: (1) `Region ID`, a hardware-managed counter that atomically increases to ensure unique ID allocation across all cores; (2) `PendingWrs` indicating the number of unpersisted stores in the region; (3) `MCBitVec` tracking the IDs of the MCs to which the region's stores are directed; and (4) `RS Pointer` referring to the starting address of the region's recovery slice (RS), which is encoded in the region boundary instruction; Section 6.6 details RS. Similarly, when a store instruction commits, cWSP performs two actions. First, a PB entry is allocated to track its persistence status. Here, each PB entry contains 5 items: (1) `Region ID`—never overflowing as at most 128 regions are allowed to be persisted concurrently, given 8 cores and 16 RBT entries by default—of the current region which is retrieved from the RBT tail

entry; (2) store address `Addr`; (3) `Data` being stored; (4) a boolean `LogBit` telling if the store is from a speculative region and thus should be undo-logged; and (5) a boolean `Sent` stating if the store has been delivered to NVM. Second, for the committed store, cWSP increases the RBT tail entry's `PendingWrs` by 1 and updates its `MCVectBit` with the store's MC ID.

As shown in Figure 6.9, the coordination between the RBT and the PB is crucial for keeping the speculation status of every region up-to-date. Each cycle the PB keeps sending its entry to the target MC (①) with the `Sent` set in a pipelined manner—unless the WPQ is full, which is not common (see Section 6.8.7). Technically, the first 4 items of the PB entry (i.e., `Region ID`, `Addr`, `Data`, and `LogBit`) are sent to the WPQ of the target MC[5]. Upon the arrival of the `Data` at the WPQ, it is considered persisted—as the WPQ is in the persistent domain [397]—and undo-logged if the `LogBit` is set. Simultaneously, the MC acknowledges the PB (②), which deallocates the entry if it is the head of the PB. Then, cWSP identifies the RBT entry corresponding to `Region ID` and decreases its `PendingWrs` by 1 (③). Finally, if the `PendingWrs` becomes zero with the entry pointed by the RBT head (i.e., the non-speculative region is now persisted), cWSP deallocates the entry, making the following region non-speculative; this results in (1) reclaiming its undo logs (see the section below) and writing the RBT head entry's `RS Pointer` to NVM for future power failure (④).

Hardware Undo Logging: Since the undo logging is on the critical path for every NVM write, its implementation should be performant. Figure 6.10 (a) shows how the critical path of each NVM write is extended by a naive implementation with fetching the old value from the address of the store (①); performing the log write (i.e., the address and the value) (②); performing the in-place data write (③); and responding to the core (④). Obviously, this causes a high run-time overhead.

To this end, cWSP proposes asynchronous undo logging, i.e., the MC immediately acknowledges a store arriving there (④), while its data is undo-logged and written to NVM in the background (①-③) as shown in Figure 6.10 (b). This allows the latter to be off the critical path. To achieve this correctly, cWSP requires that for each store, the undo logging and the data write should be failure-atomic as a whole. That is, the MC should secure enough

[5]↑ This requires one bus transaction for x86_64 since an `Addr` occupies only 48 bits thus being encoded with `Region ID` and `LogBit` into an 8-byte.

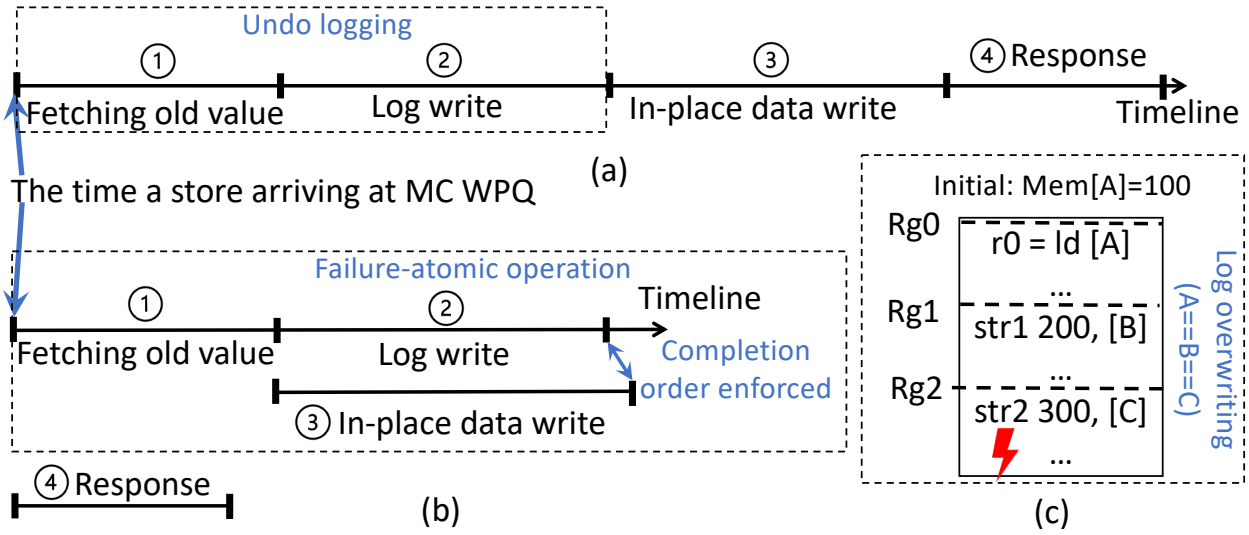


Figure 6.10. (a) Naive undo logging at MC; (b) cWSP hardware undo logging at MC; (c) Log overwriting issue; Rg0 is non-speculative, while Rg1 and Rg2 are speculative

energy for completing the entire operation (①-③) without power interruption in between—as Intel ADR secures the energy necessary for flushing all WPQ entries [397]. That way when power is about to be cut off, cWSP guarantees to flush every WPQ entry with its data undo logged. Note that the MC starts the undo logging of data being stored (①-②) as soon as it gets to WPQ. Nevertheless, when the WPQ entry is about to be flushed to NVM, its undo-logging might not have been finished, in which case the MC should hold the flushing until their completion order (②→③) is enforced for correct recovery. The rationale here is that the failure-atomic operation—including the undo logging—is only possible for the entries present in WPQ, not those already removed there.

In particular, care must be taken to prevent undo logs from being overwritten, which would otherwise corrupt NVM states causing incorrect power failure recovery. Figure 6.10 (c) shows how the undo log overwriting issue arises. Here, Rg0 is non-speculative, i.e., the oldest unpersisted region to be re-executed in case of power failure, while Rg1 and Rg2 are speculative. Suppose addresses `A`, `B`, and `C` happen to be the same, i.e., `str1`'s log is overwritten by `str2`'s log if they share the same log location. When power failure (ϟ) occurs

in Rg2, cWSP mistakenly uses the `str2`'s log to revert Rg1s speculative NVM updates, resulting in inconsistent NVM states. That is because *ld* in Rg0 incorrectly reads 200 (not 100) when it restarts in the wake of the power failure.

To this end, cWSP leverages *append-only* logging for eliminating the overwriting within a region and across regions. The implementation principle here is twofold: (1) lightweight log management without additional hardware support and (2) simple log deallocation with no search cost. In light of this, cWSP requires that each MC should (1) maintain the logs of stores arriving there in its local NVM space—rather than resorting to centralized logging with inter-MC communication—(2) manage the logs on a per-region basis such that each region's logs can be deallocated using the `Region ID`; upon receiving the first store of a speculative region, the target MC allocates a log array for the `Region ID` in its own log area; once a region gets non-speculative, its idempotent recovery no longer requires its own logs—though it needs those of the following speculative region(s). This implies that cWSP can safely deallocate the logs of the non-speculative region without compromising the recovery guarantee. To achieve that, cWSP consults the `MCVecBit` of the RBT head—referring to the non-speculative region—and signals its target MCs to reclaim the log arrays corresponding to the `Region ID`. Notably, the size of the log area is limited since each region has only a handful of stores (4 on average) and the number of regions being concurrently persisted is capped by the RBT size.

6.5 Interaction with OS

In the pursuit of whole-system persistence, cWSP faces a challenge in ensuring consistent NVM states during system calls that require context switch from the user space to the kernel space. This challenge arises because the `entry function`—invoked by every system call—is implemented using assembly code, and it cannot be partitioned by cWSP compiler into idempotent regions. As a result, the *entry function* is not recoverable if power failure occurs therein.

To address the above challenge, we manually delineate region boundaries and insert register checkpoints in the *entry function*, i.e., `entry_SYSCALL_64` in the assembly file

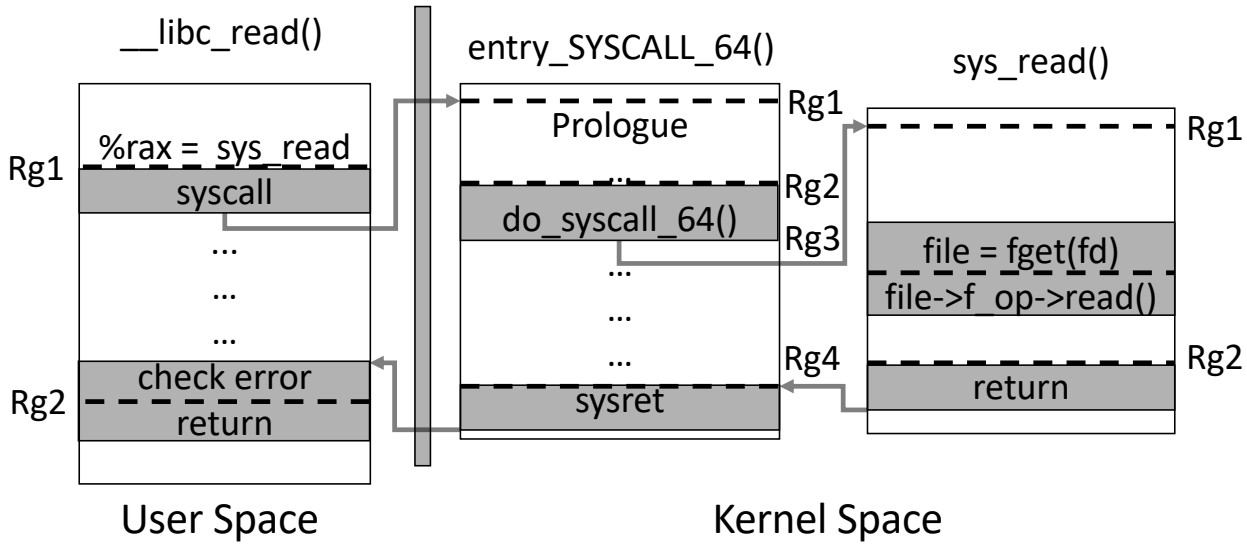


Figure 6.11. Region formation for Linux system calls

`entry_64.S`. The overhead caused by these checkpoints is minimal, since a typical system call involves more than 4000 instructions [398], though other auxiliary functions called by `entry_SYSCALL_64` should also be instrumented. Figure 6.11 shows that 2 region boundaries are inserted at the entry and exit points of `entry_SYSCALL_64`, and another region boundary is inserted right before the callsite `do_syscall_64`; it transfers the program control to the beginning of `sys_read` function pointed to by the input register `%rax`; region boundaries for other callsites are omitted in the figure. With the help of the manual annotation on `entry_SYSCALL_64` and its auxiliary functions, cWSP can ensure crash consistency for all in that cWSP compiler already recompiles the `glibc` and the Linux kernel.

6.6 Recovery Protocol

In the wake of power failure, cWSP follows its recovery protocol to resume the power-interrupted program from the recovery point—the beginning of the oldest unpersisted region. That is, for the preparation of the region re-execution, cWSP's recovery runtime (1) leverages undo logs to make NVM states consistent and (2) jumps to the region's recovery slice (RS) where its live-in registers are restored. Figure 6.12 shows how the recovery protocol works.

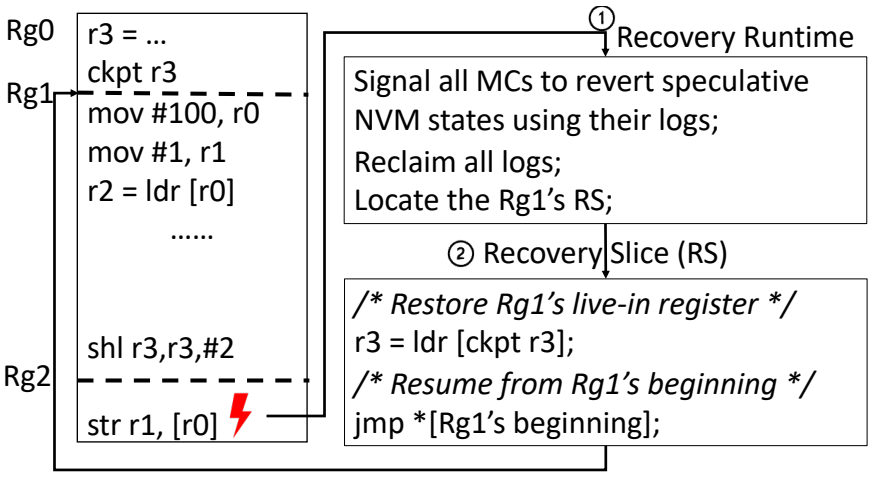


Figure 6.12. Recovery process for the interrupted (⚡) Rg1 and Rg2

Suppose Rg0 has already been persisted, and Rg1 is the oldest unpersisted region while Rg2 is speculative. When power failure interrupts Rg1 and Rg2, cWSP's runtime first signals all MCs to revert speculative NVM states; each MC processes its own per-region logs in a reverse chronological order of `Region ID` and then deallocates all its logs (①). The runtime then jumps to Rg1's RS (②) that restores its live-in register $r3$. Finally, at the end of the RS, it transfers the program control back to the beginning of Rg1, and the program resumes the execution as is thereafter.

6.7 Discussion

Recovery for Multi-Cores: To ensure correct power failure recovery for multi-threaded applications on multi-core processors, cWSP maintains inter-thread dependency [299] by treating synchronization primitives, such as *atomics* and *fences*, as region boundaries as with prior techniques [18, 57, 232, 248, 285, 286]. That way, cWSP ensures that stores prior to synchronization primitives are not only merged into the L1 data cache but also persisted before the primitives are committed. As a result, for data-race-free (DRF) program assumed by C/C++ 11 onward, a dependent thread can only enter a critical section after a source thread has already persisted the stores of the section and exited the section. The implication

is twofold: (1) upon power failure, there is at most one thread in the same critical section; (2) in the wake of power failure, each thread resumes its execution from the end of the latest persisted section (region) independently without the need to track the happen-before relationship among threads.

Why Not Use Store Queue as Persist Buffer: Utilizing the store queue (SQ) as a persist buffer would result in stores being held in the SQ for an extended period, thereby putting more pressure on the SQ. While enlarging the SQ could alleviate the pressure to some extent, it would also increase the latency of the critical store-to-load forwarding [138, 399], thus affecting the core pipeline performance.

I/O and Device States: To the best of our knowledge, supporting irrevocable operations like I/O has been an open problem. Despite, cWSP can be extended to have battery-backed redo buffers—organized as a FIFO queue—to ensure consistent I/O device states across power failure. We suggest that the number of the redo buffers should match the RBT size with each buffer indexed by a `Region ID`. This allows multiple regions to be persisted concurrently as with the RBT. During the execution of a region, its I/O operations are held in the corresponding redo buffer. Once the oldest unpersisted region becomes persisted, i.e., all its I/O operations already arrive at the corresponding redo buffer, cWSP flushes their data to the corresponding devices.

In the event of power failure, cWSP performs two actions for recovery. First, it exploits the system's ACPI (Advanced Configuration and Power Interface) [110, 400] to save device states—including internal buffers and registers—to NVM. Second, cWSP examines the FIFO queue from front to rear to flush I/O data of each persisted region to their target devices. To ensure in-order region persistence, cWSP stops such an examination when an unpersisted region is encountered even if there might exist following persisted regions. As such, the device states get consistent back with those when the oldest unpersisted region started in the first place. When power comes back, cWSP's runtime resumes the execution of the device driver code from the beginning of the oldest unpersisted region—which is the recovery point as always.

Software Bugs: Software bugs can corrupt memory data and in turn lead to system crash. However, this is different from what cWSP pursues since they are two different problems.

For example, any existing systems that maintain crash consistency, including databases and long-running machine learning (ML)/high performance computing (HPC) applications, could still experience crash caused by software bugs.

No Power Failure Recovery Test: At this moment, cWSP does not conduct experiments for system-level recovery from power failure, which we admit is a limitation of the current evaluation. We leave addressing the limitation for our future work. Nevertheless, the recovery overhead of cWSP would be negligible since it re-executes only tens of instructions in power-interrupted regions as described in Section 6.8.5.

6.8 Evaluation

We implement our compilation optimizations atop LLVM [139] that compile all the evaluated applications with -O3 flag; they are statically linked against cWSP runtime. Our compiler passes consist of about 4000 LOC with comments excluded.

We implement our hardware design atop gem5 [204] simulator to model an 8-core Skylake processor [401] with 2 memory controllers (MCs). Each of them manages DRAM as an off-chip direct-mapped LLC as with Intel PMEM's memory mode. Each core is equipped with a 64KB 8-way private L1 data cache with 4-cycle hit latency and a 32KB 8-way private L1 instruction cache with 3-cycle hit latency. All the 8 cores share a 16MB 16-way L2 cache with 44-cycle hit latency. The DRAM cache is configured to 4GB DDR4 2400 8x8. We set NVM main memory to 32GB with read/write latency of $175ns/90ns$ [120, 338]. Each MC has a 24-entry battery-backed WPQ, while RBT/PB sizes are set to 16/50. The round-trip latency of the persist path is set to $20ns$ (40 cycles) as with prior schemes [60, 248], which is considered conservative as a prior work Hermes [402] assumes a separate data path of 36-cycle round-trip latency. In addition, cWSP's persist path requires a bandwidth of only 4GB/s, which is realistic considering a 25GB/s DRAM bus [403]. We treat the original program running on the original hardware platform without crash consistency support as our baseline.

To highlight WSP's benefits, we evaluate a variety of benchmarks, e.g., CPU2006/2017[140, 141], SPLASH3 [217], WHISPER [307], STAMP [384], and Mini-apps [218, 219].

They represent various application domains, e.g., traditional CPU benchmarks, multi-threaded applications, key-value stores, transactional applications, and HPC applications. We simulate CPU2006/2017 program with reference input and modify the source code of WHISPER to stress the DRAM cache. all SPLASH3/WHISPER/STAMP applications are simulated in gem5 FS mode. The FS simulation boots an Ubuntu 18.04 with Linux kernel 5.4.49. As with prior schemes [40, 114, 116, 119, 214, 244, 343, 344, 404], we synchronize the simulation window by measuring the number of function calls—a constant across different binary versions generated by varying compiler optimizations—in the baseline to fast-forward 5 billion instructions and then simulate 1 billion instructions in gem5's O3CPU model.

6.8.1 Run-time Overhead Analysis

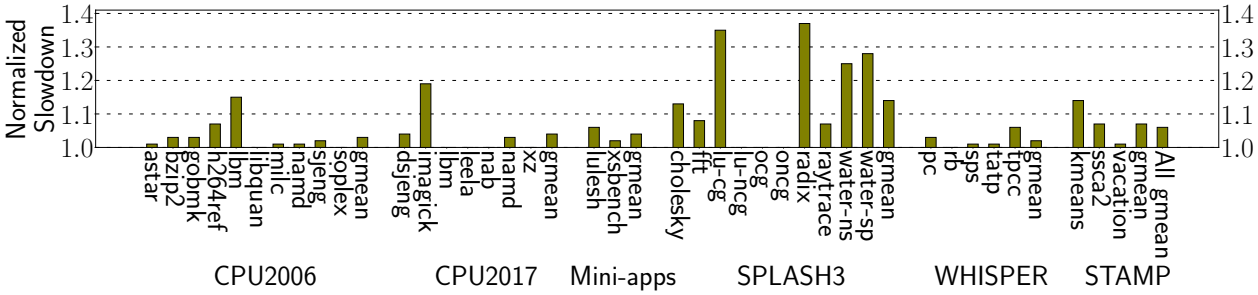


Figure 6.13. Normalized slowdown of cWSP to the baseline; the persist path bandwidth is 4GB/s; lower is better

Figure 6.13 shows that cWSP incurs an average of only 6% run-time overhead across 37 applications. Notably, cWSP incurs higher overheads for SPLASH3 applications, e.g., `lu-contig` and `radix`. This is because: (1) their baselines have a short execution time due to their low L1 data cache miss rates (∼2%); they have good data locality due to many sequential/repeated writes; (2) these sequential/repeated writes exert a high pressure on the persist path, overflowing the PB/WPQ frequently and thus causing the higher overheads. In contrast, other applications exhibit less normalized run-time overheads due to less frequent NVM writes.

In addition, we compare cWSP with two prior WSP schemes, ReplayCache [245] and Capri [60], to underscore the exceptional performance of cWSP. ReplayCache is adapted to

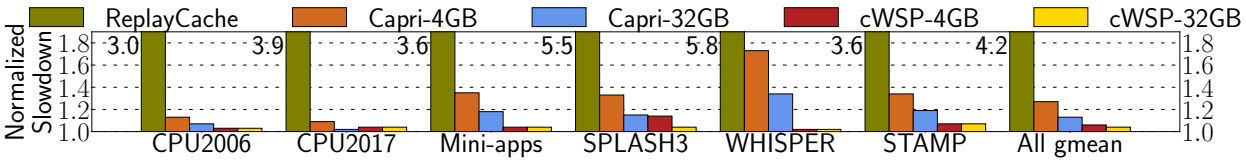


Figure 6.14. Normalized slowdown of cWSP and other WSP schemes; lower is better; the numbers after dash in the legend indicate the persist path bandwidth

to the evaluated server-class processor since it was originally designed for energy harvesting systems [87, 95, 134, 189, 246, 249–251, 327, 405] where WSP is the norm. In the evaluation of cWSP and Capri, we consider two persist path bandwidth configurations: a practical 4GB/s and an ideal 32GB/s. As shown in Figure 6.14, cWSP outperforms both prior schemes across all benchmarks. ReplayCache results in a significant slowdown (4.3x) due to its software-oriented design, while Capri backed with 4GB/s persist path bandwidth incurs an average of 27% run-time overhead due to the contention on the persist path. Only with the ideal persist path bandwidth, can Capri be on par with cWSP.

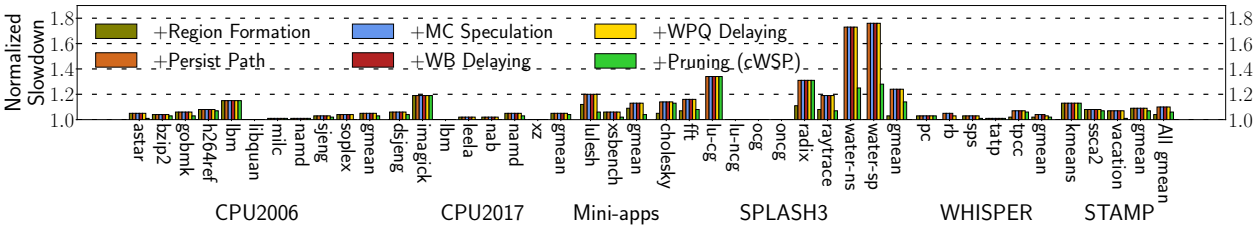


Figure 6.15. The performance impact of each cWSP optimization; lower is better

6.8.2 Performance Impact of Each Optimization

To show how each cWSP optimization affects the run-time overhead, we break down the overhead as shown in Figure 6.15.

Region Formation: reveals that cWSP's region formation incurs an average of only 4% run-time overhead.

Persist Path: is the combination of above optimization with persisting stores to NVM through the persist path. This increases the average run-time overhead to 10% primarily because of the contention for the persist path.

MC Speculation: is the combination of all above optimizations with MC speculation. The resulting overhead remains the same since a 16-entry RBT is sufficiently large to cover the persist path latency; details deferred to Section 6.8.7.

WB Delay: is the combination of all above optimizations with delaying the writeback from the L1D's WB; there is no extra overhead incurred (see Section 6.4.1 for the reason).

WPQ Delay: is the combination of all above optimizations with delaying the response from the WPQ in MC for loads hitting in the WPQ. There is no observable increase in the run-time overhead (see Section 6.8.1 for the reason).

Pruning (cWSP): uses all above optimizations along with checkpoint pruning, lowering the average run-time overhead to only 6%. This technique dramatically reduces the overheads of certain applications, e.g., `water-ns` and `LULESH`.

6.8.3 Run-Time Overhead Analysis for CXL-Based NVM

Table 6.1. CXL memory devices

Device	CXL IP	Memory Technology	Max. Bandwidth	Latency (read/write)
CXL-A (NVDIMM)	Hard IP	DDR5-4800	38.4GB/s	158ns/120ns
CXL-B (NVDIMM)	Hard IP	DDR4-2400	19.2GB/s	223ns/139ns
CXL-C (NVDIMM)	Soft IP	DDR4-3200	25.6GB/s	348ns/241ns
CXL-D (PMEM)	Simulation	Intel Optane	6.6GB/s for read 2.3GB/s for write	245ns/160ns

To showcase the scalability of cWSP for the future far CXL-based NVM, we model three CXL NVDIMMs (CXL-A to CXL-C) in our simulator with the parameters from a recent empirical analysis of CXL DRAM memory [406]. Additionally, we model another CXL PMEM (CXL-D) by adding $70ns$ CXL interconnect latency [407] to the existing PMEM technology [338]. We keep all other parameters the same as those listed in Section 6.8, except for the latency and bandwidth parameters of the CXL-based NVM.

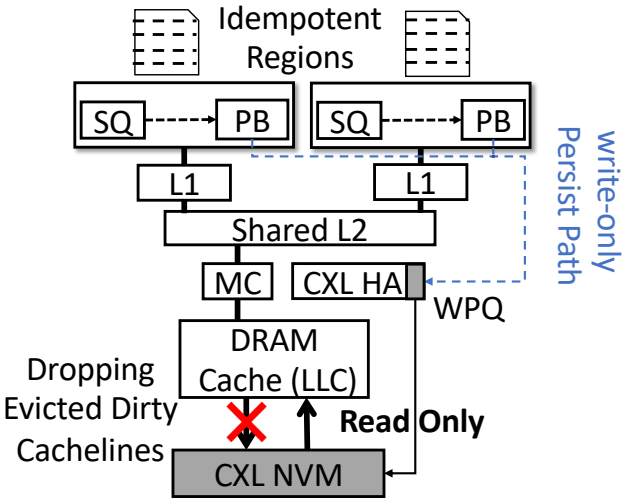


Figure 6.16. cWSP architecture for CXL-based NVM; local DRAM served as an LLC atop the NVM

Figure 6.16 depicts the high-level architecture of cWSP where local DRAM works as an LLC atop of CXL-based NVM. Note that the persistence domain just moves from the battery-backed WPQ of conventional MC—as shown in Figure 6.3 (b)—to the one of CXL Home Agent (HA)[6], keeping the persist path length technically the same. The implication of the battery-backed WPQ of the HA is that the data being stored become persistent as soon as they arrive in the WPQ. In other words, on power failure occurs, data buffered in the WPQ are ensured to be flushed to the CXL-based NVM through the internal buffers along the way, i.e., cWSP does not have to pay for the long latency of traveling from the HA to the CXL-based NVM. With the help of the same persist path length, cWSP maintains high performance for such a deep cache hierarchy.

Figure 6.17 shows the normalized slowdown of cWSP across selected memory intensive applications with varying CXL devices. The memory footprints of those program range from 2.5GB to 6GB. Notably, cWSP maintains an average of only 4% run-time overhead, regardless of the speed of the underlying CXL memory. Intriguingly, cWSP exhibits a slightly higher overhead with faster CXL memory. This is because cWSP benefits less from the speed

[6]↑It controls the communication between the processor core and the CXL-based NVM, e.g., translating load/store requests into PCIe transactions [374].

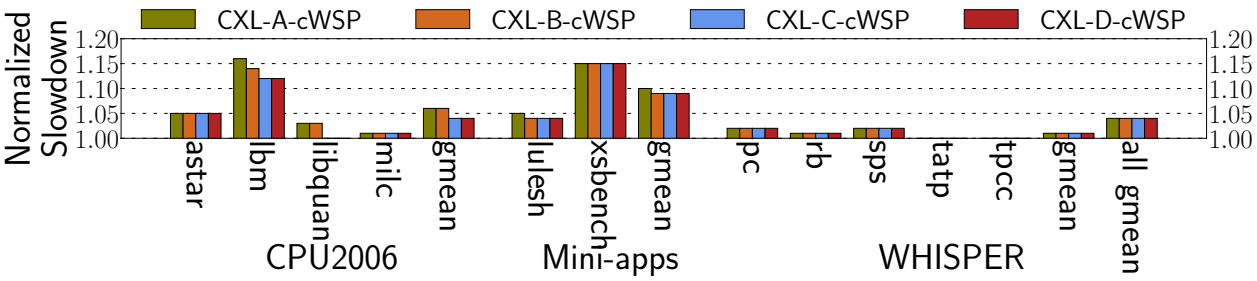


Figure 6.17. Normalized slowdown of cWSP to the baseline (original program on CXL devices without crash consistency support) with varying CXL configuration; lower is better

enhancement in comparison to the baseline—primarily due to store persistence, leading to a higher normalized overhead. This trend aligns with the observation made with fast NVM technology, as detailed in Section 6.8.7. Note that cWSP incurs higher overheads for some applications, e.g., `lbm` and `XSBench`, due to more RBT overflow caused by their shorter regions; see Section 6.8.5.

6.8.4 Comparison to Partial-System Persistence

To highlight the benefits of enabling DRAM as a cache, we implement an optimized version of BBB [287] that behaves as an ideal PSP scheme like Intel eADR with DRAM disabled. We believe that this ideal PSP attains the performance akin to LightPC [349], a system that replaces DRAM with slower PCM RAM and relies on the modified OS to flush the entire volatile data to NVM right before power failure. We then compare cWSP to this ideal PSP scheme across those selected memory intensive applications; their L2 miss rates range from 20% to 100%. Figure 6.18 shows that cWSP incurs an average of only 3% run-time overhead, thanks to enabling the DRAM cache. In contrast, the ideal PSP scheme causes a substantial 52% performance slowdown on average in that every single memory operation must access slower NVM.

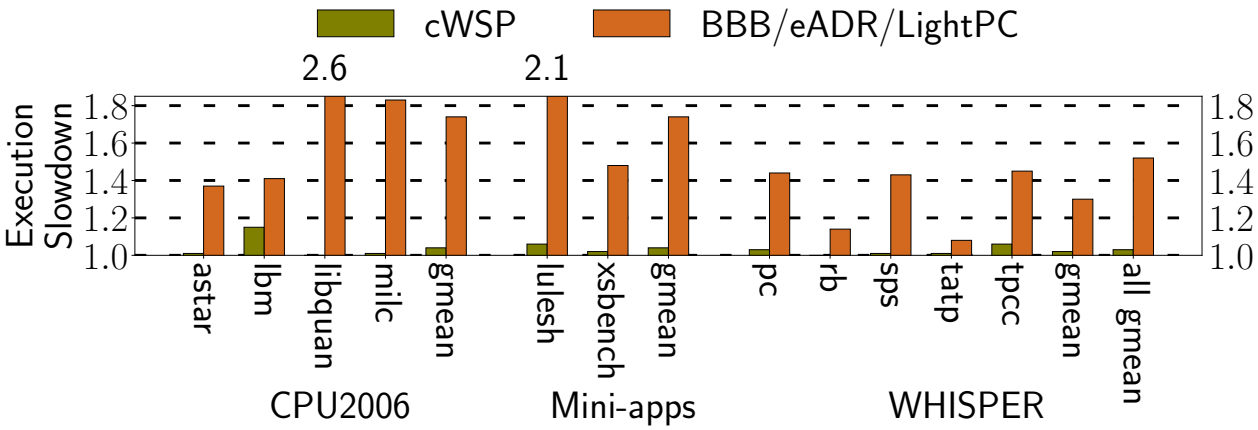


Figure 6.18. Normalized slowdown of cWSP and the ideal PSP (BBB/eADR/LightPC) to the baseline; lower is better

6.8.5 Region Characteristics

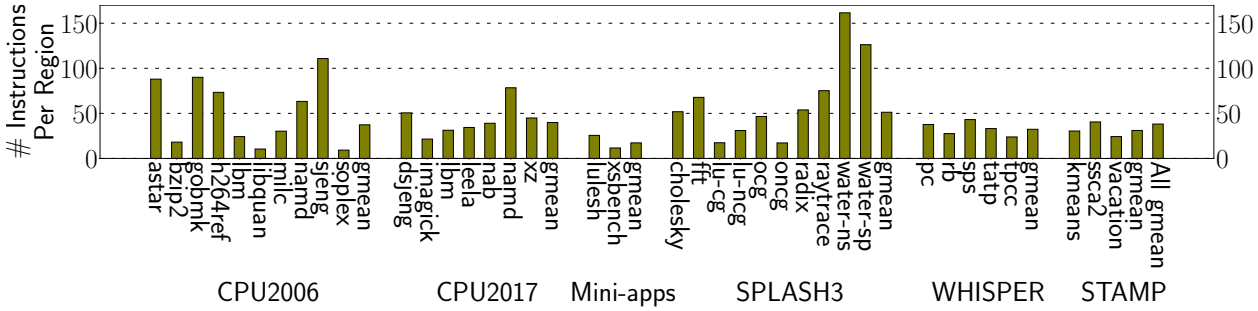


Figure 6.19. Average number of instructions in regions

Given the critical role of idempotent region size in influencing power failure recovery time and the efficiency of the asynchronous store persistence, we collect the number of dynamic instructions in each region and report the average numbers in Figure 6.19. It shows that there are 38.15 instructions in each region on average, which signifies cWSP's fast failure recovery. Furthermore, with a 16-entry RBT, cWSP overlaps the long persistence latency of the oldest unpersisted region with the execution latency of 572 (16x38.15) instructions.

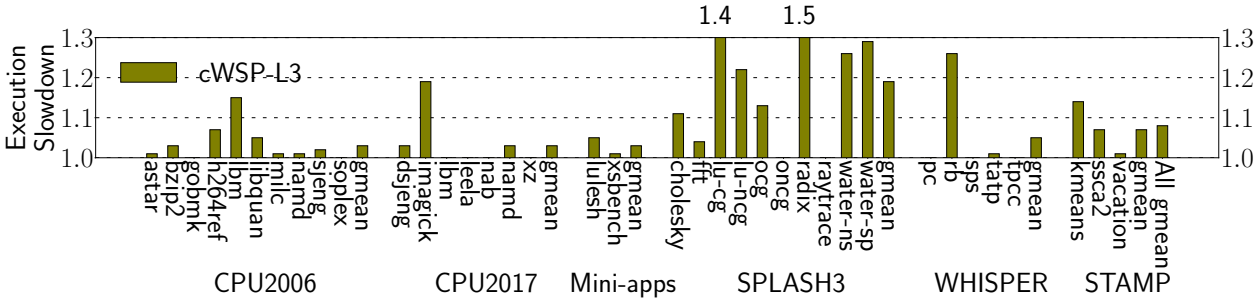


Figure 6.20. cWSP's slowdown with L3 cache

6.8.6 Impact of Deeper Cache Hierarchy

To show how cWSP performs for a deeper cache hierarchy, i.e., a 3-level SRAM cache atop DRAM cache, we add a shared 16-way set-associative writeback L3 cache of 44-cycle hit latency to both cWSP and the baseline. We also change the existing L2 cache in Figure 6.3 (b) to a private 8-way set-associative L2 with 1MB and 14-cycle hit latency. Figure 6.20 depicts that cWSP still incurs a low run-time overhead, i.e., only 8% on average, thanks to the efficient asynchronous store persistence; see Section 6.8.7 for details.

6.8.7 Sensitivity Analysis

Sensitivity to Persist Path Bandwidth: Since cWSP persists stores to NVM through the persist path, its bandwidth plays a key role in determining the overall performance. To explore the impact of persist path bandwidth on cWSP's performance, we conduct experiments with varying persist path bandwidth, from 1GB/s up to 32GB/s, as shown in Figure 6.21. The key trend is that the run-time overhead of cWSP decreases as the bandwidth rises. Thanks to cWSP's 8-byte persist granularity, cWSP's run-time overhead remains the same once the bandwidth rises beyond 10GB/s, confirming cWSP's low demand for persist path bandwidth.

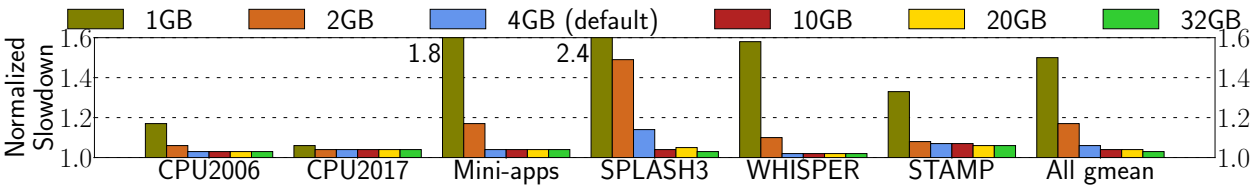


Figure 6.21. cWSP's slowdown with varying persist path bandwidth from 1GB/s up to 32GB/s; lower is better

Sensitivity to Region Boundary Table (RBT) Size: The region boundary table (RBT) is so critical that its size highly affects how frequently the core pipeline stalls; the core pipeline stalls at a region boundary if RBT is full. We systematically vary RBT size from 8 to 32 to assess cWSP's performance. As shown in Figure 6.22, cWSP's run-time overhead rises to 11% on average and up to 20% for SPLASH3, when setting RBT size to 8. This is because the regions of SPLASH3 program are relatively short, leading to more pipeline stalls for a 8-entry RBT. Here, cWSP's run-time overhead decreases to only 4% with a 32-entry RBT.

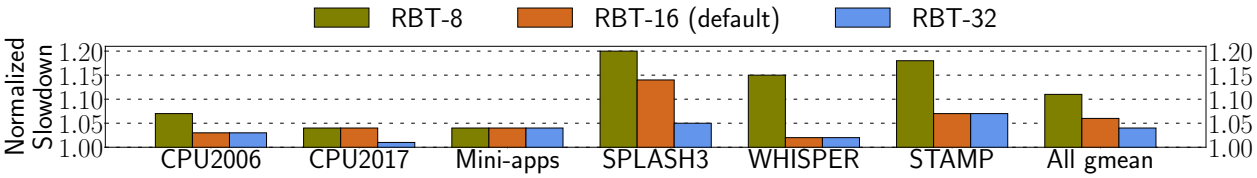


Figure 6.22. cWSP's normalized slowdown with varying RBT size

Sensitivity to Persist Path Latency: To show the impact of persist path latency on cWSP's performance, we vary the persist path latency from 10ns to 40ns. Figure 6.23 shows that the persist path latency can be almost entirely overlapped by region execution, even if the latency increases up to 40ns, thanks to the efficient asynchronous store persistence enabled by RBT. Notably, SPLASH3 exhibits a higher run-time overhead caused by more frequent NVM writes; please refer to Section 6.8.1 for details.

Sensitivity to Write Buffer (WB) Size: To show the impact of delaying dirty cacheline writeback from the L1D's WB, we conduct a series of simulations with varying the WB size. Figure 6.24 depicts that cWSP's overhead remains the same no matter how small the WB is owing to the faster enough persist path; refer to Section 6.4.1 for details.

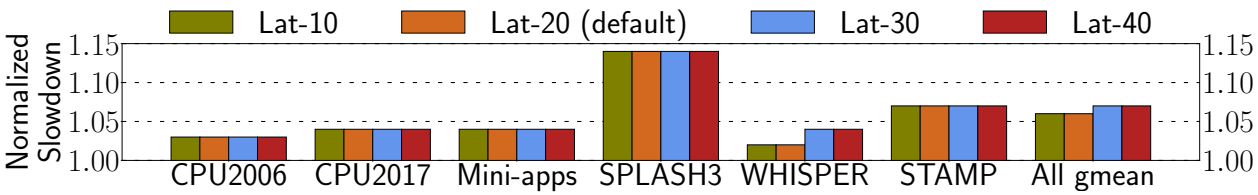


Figure 6.23. cWSP's slowdown with varying persist path latency from $10ns$ to $40ns$

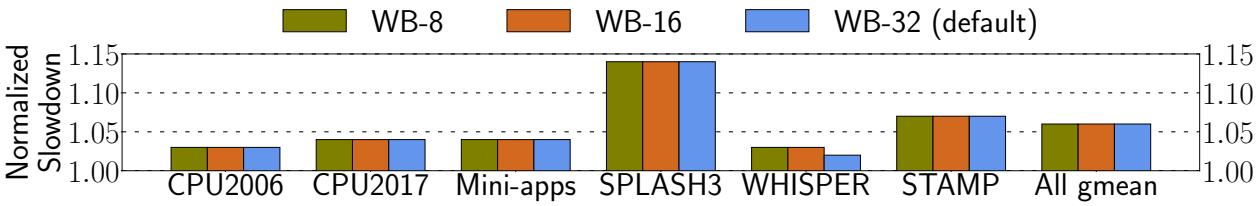


Figure 6.24. cWSP's slowdown with varying L1D's WB size

Sensitivity to Persist Buffer (PB) Size: As a critical component, the PB should be large enough so as not to congest the persist path frequently. Figure 6.25 shows that cWSP's performance is insensitive to PB size. Here, cWSP's overhead rises to only 7% even if the PB size is 20, thanks to the asynchronous store persistence; cWSP sets default PB size to 50 for maximal performance.

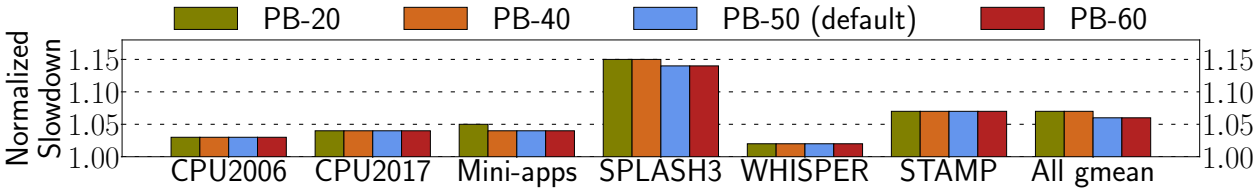


Figure 6.25. cWSP's slowdown with varying PB size

Sensitivity to NVM WPQ Size: As a shared component among multiple cores, NVM WPQ should be appropriately sized to keep the pressure on it low. Figure 6.26 shows that a 24-entry WPQ is large enough maintain cWSP's low run-time overhead; it is cheap to scale the WPQ size to 24 owing to its 8-byte granularity. As the WPQ size decreases to 8, cWSP still incurs a moderate run-time overhead, i.e., 11% on average. Notably, cWSP incurs an up

to 31% overhead for SPLASH3 due to a high pressure on the WPQ caused by its frequent NVM writes.

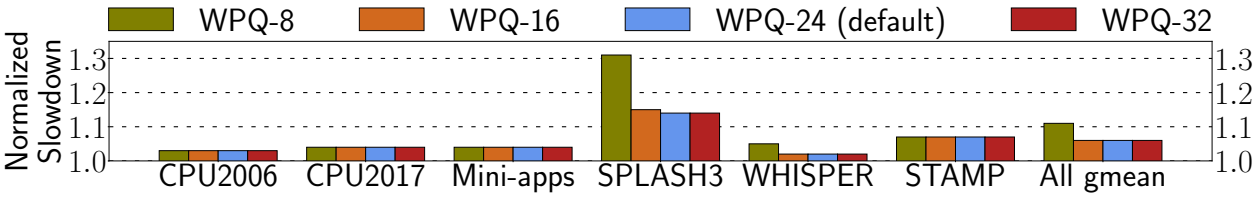


Figure 6.26. cWSP's slowdown with varying WPQ size

Sensitivity to NVM Technology: To analyze how NVM technologies affect the performance of cWSP, we evaluate cWSP for 3 NVM technologies: PMEM [120], STT-MRAM [408], and ReRAM [409]. Figure 6.27 shows that cWSP maintains its low overhead (8%), regardless of the NVM technique. Note that cWSP incurs a marginally elevated overhead with fast NVM techniques, e.g., ReRAM. This phenomenon arises from the fact that cWSP benefits less from faster NVM techniques than the baseline—due to the store persistence, resulting in a higher normalized overhead; the same phenomenon appears for faster CXL devices (see Section 6.8.3).

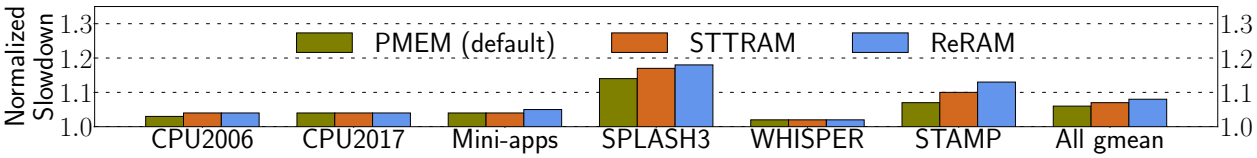


Figure 6.27. cWSP's slowdown with varying NVM technologies

6.8.8 Hardware Overhead

cWSP incurs only a storage overhead of 176 bytes for the 16-entry RBT whose entry size is 11 bytes (see Figure 6.9). Note that cWSP does not incur hardware overhead for the PB since it can be covered by the Intel 1KB write-combing buffer (WCB) [112]. We also use CACTI [345] with $22nm$ technology to calculate the hardware overhead of the RBT. The calculation results turn out that the RBT costs only $0.001mm^2$. Table 6.2 shows that RBT and PB have a minimal impact on the critical path of program execution.

Table 6.2. cWSP's hardware overheads

	Area (mm^2)	Access Latency (ns)	Dynamic Access (pJ)
50-entry PB	0.005	0.092	0.001
16-entry RBT	0.001	0.077	0.0004

6.9 Other Related Work

In general, there are two types of application-level crash consistency schemes that are implemented by software: (1) failure-atomic sections (FASEs) protected by the outermost pair of lock and unlock as in iDO [57] and (2) persistent transactions such as Clobber-NVM [24] and LAD [354]. On one hand, iDO achieves correct power failure recovery using idempotent processing and live-out register checkpointing. However, it incurs a high run-time overhead due to introducing persist barriers at each region boundary. On the other hand, Clobber-NVM undo logs program stores as with other transaction-based schemes yet in a more intelligent way. That is, rather than undo logging every store in a transaction, Clobber-NVM does only antidependent therein—since others are to be reinitialized during the re-execution of power-interrupted transaction. In a sense, Clobber-NVM resembles the MC speculation of cWSP in that it also undo-logs cross-idempotent-region stores that might be antidependent on some prior region's loads. However, Clobber-NVM still suffers from persist barrier cost between the transactional store and its log stores. LAD [354] uses a hardware redo buffer in MC to log memory updates of transactions. Unfortunately, LAD needs to fall back to undo logging upon full redo buffer. Moreover, LAD suffers from frequent core pipeline stalls, i.e., 163 cycles on average, at end of short each transaction; its size is limited to the redo buffer size.

LightPC [349] and Zhuque [362] offer crash consistency and persistence at the process level. They are inferior to cWSP for 3 reasons: (1) requiring extensive modifications on the OS source code or C library, (2) having poor performance due to the inability to enable DRAM; LightPC uses PCM RAM, while Zhuque maps the memory objects of user processes

to PMEM space, and (3) consuming a lot of energy to dump entire volatile states to NVM upon power loss as with the pioneering work on WSP [110].

6.10 Summary

This dissertation presents cWSP, a compiler-directed whole-system persistence (WSP) scheme. cWSP compiler partitions input program into a series of idempotent regions so that the program can correctly recover from power failure by reexecuting the oldest unpersisted region. During the execution of the idempotent regions, cWSP architecture persists their data being stored in a performant way without breaking the recovery guarantee. Experimental results with 37 applications from SPEC CPU2006/2017/DOE Mini-apps/SPLASH3/ WHISPER/STAMP highlight the low run-time overhead of cWSP, i.e., 6% on average, achieving a 4.5x reduction compared to that of the state-of-the-art work.

7. CONCLUSION

Reliability against errors such as soft errors and crash consistency is as crucial as performance and power efficiency for modern processors due to their potential significant impact, such as massive financial losses and severe injuries to human lives. The dynamic nature of modern computing workloads further complicates the implementation of reliable systems, often leading to compromises in reliability to maintain high performance and low hardware complexity.

In this dissertation, we first demonstrate how sophisticated compiler/architecture co-design achieves near-zero-overhead soft error resilience for embedded cores. We then introduce a software-only approach that ensures crash consistency for energy harvesting systems, which are backed with embedded cores, and outperforms the state-of-the-art by a factor of 9. Additionally, we highlight that a simple microarchitectural technique can provide near-zero-overhead soft error resilience for server-class cores, incurring a storage overhead of only three registers and a countdown timer. Finally, we explore two techniques that achieve high-performance crash consistency for server-class cores by leveraging existing architectural features.

Through these innovations, this dissertation paves the way for more reliable and efficient computing systems. These advancements are particularly desirable in various applications, such as datacenters, HPC clusters, space exploration control systems, aviation control systems, and intelligent agriculture drones.

REFERENCES

[1] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE micro*, vol. 23, no. 4, pp. 14–19, 2003.

[2] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, IEEE, 2005, pp. 243–247.

[3] Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, 2014, pp. 467–478.

[4] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proc. USENIX Annual Technical Conference (ATC10)*, 2010, pp. 75–88.

[5] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, IEEE, 2010, pp. 691–696.

[6] A. Rezaei, H. Khetawat, O. Patil, F. Mueller, P. Hargrove, and E. Roman, "End-to-end resilience for hpc applications," in *International Conference on High Performance Computing*, Springer, 2019, pp. 271–290.

[7] S. Hukerikar and C. Engelmann, "Plexus: A pattern-oriented runtime system architecture for resilient extreme-scale high-performance computing systems," in *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, IEEE, 2020, pp. 31–39.

[8] R. A. Ashraf, S. Hukerikar, and C. Engelmann, "Pattern-based modeling of multiresilience solutions for high-performance computing," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 80–87.

[9] C.-H. Ho, M. de Kruijf, K. Sankaralingam, B. Rountree, M. Schulz, and B. R. de Supinski, "Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing," in *2012 41st International Conference on Parallel Processing*, IEEE, 2012, pp. 510–519.

[10] M. E. Gomez *et al.*, "A routing methodology for achieving fault tolerance in direct networks," *IEEE transactions on Computers*, vol. 55, no. 4, pp. 400–415, 2006.

[11] M. E. Gómez *et al.*, "A new adaptive fault-tolerant routing methodology for direct networks," in *International Conference on High-Performance Computing*, Springer, 2004, pp. 462–473.

[12] J. M. Montañana, J. Flich, A. Robles, and J. Duato, "A scalable methodology for computing fault-free paths in infiniband torus networks," in *High-Performance Computing*, Springer, 2005, pp. 79–92.

[13] M. Andjelkovic, J. Chen, A. Simevski, Z. Stamenkovic, M. Krstic, and R. Kraemer, "A review of particle detectors for space-borne self-adaptive fault-tolerant systems," in *2020 IEEE East-West Design & Test Symposium (EWDTS)*, IEEE, 2020, pp. 1–8.

[14] H. D. Dixit *et al.*, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[15] D. F. Bacon, "Detection and prevention of silent data corruption in an exabyte-scale database system," 2022.

[16] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, pp. 1–13, 2023. DOI: 10.1109/TC.2023.3285094.

[17] D. Agiakatsikas *et al.*, "Impact of voltage scaling on soft errors susceptibility of multicore server cpus," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 957–971.

[18] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[19] J. Coburn *et al.*, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[20] J. Condit *et al.*, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[21] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 271–282.

[22] J. Gu *et al.*, "Pisces: A scalable and efficient persistent transactional memory," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 913–928.

[23] R. M. Krishnan *et al.*, "Durable transactional memory can scale with timestone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[24] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: Log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 346–359.

[25] S. M. Shahri, S. A. V. Ghahani, and A. Kolli, "(almost) fence-less persist ordering," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 539–554.

[26] I. Amazon Web Services, *Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region*, https://aws.amazon.com/message/41926/, 2017.

[27] A. T. S. B. (ATSB), *Qantas flight 72 airbus a330 incident report*, https://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070.aspx, 2008.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2005, pp. 243–254.

[29] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2016, pp. 227–238.

[30] M. Didehban and A. Shrivastava, "Nzdc: A compiler technique for near zero silent data corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2016, pp. 1–6.

[31] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee, "Expert: Effective and flexible error protection by redundant multithreading," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 533–538.

[32] M. Didehban and A. Shrivastava, "A compiler technique for processor-wide protection from soft errors in multithreaded environments," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 249–263, 2018.

[33] M. Didehban, A. Shrivastava, and S. R. D. Lokam, "Nemesis: A software approach for computing in presence of soft errors," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2017, pp. 297–304.

[34] H. So, M. Didehban, A. Shrivastava, and K. Lee, "A software-level redundant multi-threading for soft/hard error detection and recovery," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1559–1562.

[35] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: Communication-optimised multi-threaded error-detection technique," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, IEEE, 2016, pp. 1–10.

[36] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, 2001, pp. 214–224.

[37] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings 29th annual international symposium on computer architecture*, IEEE, 2002, pp. 99–110.

[38] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 188–201, 2006.

[39] J. Yu, M. J. Garzaran, and M. Snir, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *2009 International Symposium on Code Generation and Optimization*, IEEE, 2009, pp. 35–46.

[40] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, p. 25.

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

[42] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, p. 32, 2017.

[43] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower, "Ibm z990 soft error detection and recovery," *IEEE Transactions on device and materials reliability*, vol. 5, no. 3, pp. 419–427, 2005.

[44] E. Rojas, D. Pérez, J. C. Calhoun, L. B. Gomez, T. Jones, and E. Meneses, "Understanding soft error sensitivity of deep learning models and frameworks through checkpoint alteration," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2021, pp. 492–503.

[45] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2063–2069, 2008.

[46] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, 2013, pp. 1–12.

[47] M. Didehban, S. R. D. Lokam, and A. Shrivastava, "Incheck: An in-application recovery scheme for soft errors," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[48] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Fingerprinting: Bounding soft-error detection latency and bandwidth," *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5, pp. 224–234, 2004.

[49] J. P. A. Neto, D. M. Pianto, and C. G. Ralha, "A prediction approach to define checkpoint intervals in spot instances," in *Cloud Computing–CLOUD 2018: 11th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25–30, 2018, Proceedings 11*, Springer, 2018, pp. 84–93.

[50] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *International Conference on Machine Learning*, PMLR, 2019, pp. 5220–5230.

[51] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai, "Algorithm-directed crash consistence in non-volatile memory for hpc," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2017, pp. 475–486.

[52] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin, "Efficient checkpointing with recompute scheme for non-volatile main memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–27, 2019.

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

[54] M. K. Ramanathan *et al.*, "Durable transactional memory can scale with timestone," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, 2020, pp. 335–349.

[55] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 399–411.

[56] W. Wang and S. Diestelhorst, "Quantify the performance overheads of pmdk," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 50–52.

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

[58] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

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

[60] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 71–83.

[61] S. Hahn and J. Reineke, "Design and analysis of sic: A provably timing-predictable pipelined processor core," *Real-Time Systems*, pp. 1–39, 2019.

[62] V. Venkataramani, B. Bodin, A. Kulkarni, T. Mitra, and L.-S. Peh, "Time-predictable software-defined architecture with sdf-based compiler flow for 5g baseband processing," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2020, pp. 1553–1557.

[63] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer, "How to enhance a superscalar processor to provide hard real-time capable in-order smt," in *International Conference on Architecture of Computing Systems*, Springer, 2010, pp. 2–14.

[64] M. Schoeberl *et al.*, "Towards a time-predictable dual-issue microprocessor: The patmos approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, vol. 18, 2011, pp. 11–21.

[65] M. Schoeberl *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[66] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software (ERTS'14)*, 2014.

[67] ARM, *The arm automotive guide: Arm-based automotive partner demonstrations highlights for ces 2020*, 2018.

[68] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (tcls) processor," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 3, pp. 1–30, 2019.

[69] K. Berntorp, P. Inani, R. Quirynen, and S. Di Cairano, "Motion planning of autonomous road vehicles by particle filtering: Implementation and validation," in *2019 American Control Conference (ACC)*, IEEE, 2019, pp. 1382–1387.

[70] ARM, *How to make autonomous vehicles a reality with arm*, 2018.

[71] C. Takahashi *et al.*, "4.5 a 16nm finfet heterogeneous nona-core soc complying with iso26262 asil-b: Achieving 10- 7 random hardware failures per hour reliability," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2016, pp. 80–81.

[72] C. Hernandez and J. Abella, "Timely error detection for effective recovery in light-lockstep automotive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1718–1729, 2015.

[73] A. Mehra, W.-L. Ma, F. Berg, P. Tabuada, J. W. Grizzle, and A. D. Ames, "Adaptive cruise control: Experimental validation of advanced controllers on scale-model cars," in *2015 American Control Conference (ACC)*, IEEE, 2015, pp. 1411–1418.

[74] T. Instruments, *Advanced driver assistance (adas) solutions guide*, 2018.

[75] B. Nassi, R. Bitton, R. Masuoka, A. Shabtai, and Y. Elovici, "Sok: Security and privacy in the age of commercial drones," in *Proc. IEEE Symp. Security Privacy (SP)*, 2021, pp. 73–90.

[76] B. Nassi, R. Ben-Netanel, A. Shamir, and Y. Elovici, "Drones' cryptanalysis-smashing cryptography with a flicker," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1397–1414.

[77] R. A. Clothier, D. A. Greer, D. G. Greer, and A. M. Mehta, "Risk perception and the public acceptance of drones," *Risk analysis*, vol. 35, no. 6, pp. 1167–1183, 2015.

[78] X. Corporation, *Bionic bird (biomimetic drone) instruction manual*, https://bionicbird.com/wp-content/uploads/pdf/Bionicbird_manual/Bionic_Bird_Deluxe_Package_Manual_EN.pdf, 2021.

[79] L. Zhu, H. Guan, and C. Wu, "A study of a three-dimensional self-propelled flying bird with flapping wings," *SCIENCE CHINA Physics, Mechanics & Astronomy*, vol. 58, no. 9, pp. 1–16, 2015.

[80] G. Upasani, X. Vera, and A. Gonzalez, "Setting an error detection infrastructure with low cost acoustic wave detectors.," in *ISCA*, 2012, pp. 333–343.

[81] G. Upasani, X. Vera, and A. Gonzalez, "Reducing due-fit of caches by exploiting acoustic wave detectors for error recovery.," in *IOLTS*, 2013, pp. 85–91.

[82] G. Upasani, X. Vera, and A. Gonzalez, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery.," in *ISCA*, 2014, pp. 37–48.

[83] G. Upasani, X. Vera, and A. Gonzalez, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. XX, no. 99, 2015.

[84] G. R. Upasani, "Soft error mitigation techniques for future chip multiprocessors," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2016.

[85] A. Limited, *Arm cortex a77*, "https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a77", 2019.

[86] J. Doweck *et al.*, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

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

[88] E. Kamenar *et al.*, "Harvesting of river flow energy for wireless sensor network technology," *Microsystem Technologies*, vol. 22, no. 7, pp. 1557–1574, 2016.

[89] W. Sun, T. Tan, Z. Yan, D. Zhao, X. Luo, and W. Huang, "Energy harvesting from water flow in open channel with macro fiber composite," *AIP Advances*, vol. 8, no. 9, p. 095 107, 2018.

[90] P. Cahill, R. O'Keeffe, N. Jackson, A. Mathewson, and V. Pakrashi, "Structural health monitoring of reinforced concrete beam using piezoelectric energy harvesting system," in *EWSHM-7th European workshop on structural health monitoring*, 2014.

[91] G. Park, T. Rosing, M. D. Todd, C. R. Farrar, and W. Hodgkiss, "Energy harvesting for structural health monitoring sensor networks," *Journal of Infrastructure Systems*, vol. 14, no. 1, pp. 64–79, 2008.

[92] S. Cao and J. Li, "A survey on ambient energy sources and harvesting methods for structural health monitoring applications," *Advances in Mechanical Engineering*, vol. 9, no. 4, p. 1 687 814 017 696 210, 2017.

[93] T. Galchev, J. McCullagh, R. Peterson, and K. Najafi, "A vibration harvesting system for bridge health monitoring applications," in *Proc. PowerMEMS*, 2010, pp. 179–182.

[94] H. Jayakumar, A. Raha, and V. Raghunathan, "Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, IEEE, 2014, pp. 330–335.

[95] K. Ma *et al.*, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015, pp. 526–537.

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

[97] M. Hicks, "Clank: Architectural support for intermittent computation," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 228–240, 2017.

[98] F. Su, Y. Liu, Y. Wang, and H. Yang, "A ferroelectric nonvolatile processor with 46*mu*s system-level wake-up time and 14*mu*s sleep time for energy harvesting applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 3, pp. 596–607, 2016.

[99] M. Azure, *Next generation sap hana large instances with intel optane driver lower tco*, https://azure.microsoft.com/en-us/blog/next-generation-sap-hana-large-instances-with-intel-optane-drive-lower-tco/, 2020.

[100] G. Cloud, *Partnering with intel and sap on intel optane dc persistent memory for sap hana*, https://cloudplatform.googleblog.com/2018/07/intel-and-sap-partnership-brings-persistent-memory-to-gcp-vms.html, 2018.

[101] H. Cloud, *Huawei fusionserver pro servers set to supercharge virtualization using intel optane memory*, https://www.networkworld.com/article/3562838/huawei-fusionserver-pro-servers-set-to-supercharge-virtualization-using-intel-optane-pmem.html, 2020.

[102] M. Corporation, *Intel optane dc persistent memory, azure netapp files, and azure ultra disk for sap hana*, https://azure.microsoft.com/en-us/blog/intel-optane-dc-persistent-memory-azure-netapp-files-and-more-for-sap-hana/.

[103] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 304–315.

[104] R. S. Venkatesh, T. Mason, P. Fernando, G. Eisenhauer, and A. Gavrilovska, "Scheduling hpc workflows with intel optane persistent memory," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2021, pp. 56–65.

[105] I. Corporation, *Vmware solutions with intel optane technology*, https://www.intel.com/content/www/us/en/architecture-and-technology/vmware-solutions-with-optane-technology.html, 2021.

[106] A. Cloud, *Intel optane persistent memory accelerates mars performance for more efficient scientific computing*, https://www.intel.com/content/dam/www/central-libraries/us/en/documents/alibaba-cloud-optane-pmem-mars-white-paper.pdf, 2020.

[107] I. Corporation, *Memory optimized for data-centric workloads*, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[108] Intel, *Intel optane dc persistent memory quick start guide*, https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf, Jun. 2020.

[109] I. Coporation, *Chapter 12. intel optane dc persistent memory, intel 64 and ia-32 architectures optimization reference manual*, https://software.intel.com/content/www/us/en/develop/download/intel64-and-ia-32-architectures_optimization-reference-manual.html.

[110] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 401–410, 2012.

[111] Y. Zhou, J. Zeng, and C. Jung, "Lightwsp: Whole-system persistence on the cheap," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*, 2024.

[112] I. Corporation, *Intelő 64 and ia-32 architectures optimization reference*, https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf, 2023.

[113] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, pp. 265–276.

[114] M. De Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2011, pp. 140–151.

[115] M. A. De Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 475–486.

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

[117] A. Kolli *et al.*, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017, pp. 481–493.

[118] A. Kolli *et al.*, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.

[119] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2015, pp. 672–685.

[120] S. Yadalam, N. Shah, X. Yu, and M. Swift, "Asap: A speculative approach to persistence," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2022, pp. 892–907.

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

[122] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 392–407.

[123] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th international parallel and distributed processing symposium*, IEEE, 2014, pp. 1181–1190.

[124] W. Jang, "Soft-error tolerant quasi delay-insensitive circuits," Ph.D. dissertation, California Institute of Technology, Pasadena, CA, USA, 2011.

[125] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05, 2005, pp. 243–247, ISBN: 0-7695-2275-0.

[126] N. DeBardeleben *et al.*, "Extra Bits on SRAM and DRAM Errors - More Data From the Field," *Silicon Errors in Logic - System Effects (SELSE-10), Stanford University*, Apr. 2014.

[127] G. Upasani, X. Vera, and A. Gonzalez, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 5–18, 2016.

[128] M. K. Naveed and H. Wu, "Aster: Multi-bit soft error recovery using idempotent processing," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 4, 2020.

[129] C. Chen, "Compiler-assisted resilience framework for recovery from transient faults," Ph.D. dissertation, Georgia Institute of Technology, 2020.

[130] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *ACM Sigplan Notices*, ACM, vol. 50, 2015, p. 2.

[131] Q. Liu, "Compiler-directed error resilience for reliable computing," Ph.D. dissertation, Virginia Tech, 2018.

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

[133] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2016, pp. 228–239.

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

[135] S. Muchnick *et al.*, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[136] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[137] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[138] T. Sha, M. M. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, IEEE, 2005, 12–pp.

[139] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[140] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[141] J. Bucek, K.-D. Lange, *et al.*, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ACM, 2018, pp. 41–42.

[142] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111. DOI: 10.1109/ISPASS.2016.7482078.

[143] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718.

[144] ARM., *Arm cortex-a53 processor technique reference manual*. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf.

[145] I. Jeong, S. Park, C. Lee, and W. W. Ro, "Casino core microarchitecture: Generating out-of-order schedules using cascaded in-order scheduling windows," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 383–396.

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

[147] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[148] K. Mitropoulou, V. Porpodas, and M. Cintra, "Drift: Decoupled compiler-based instruction-level fault-tolerance," in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2013, pp. 217–233.

[149] J. Ma and Y. Wang, "Identification of critical variables for soft error detection," in *International Conference on Human Centered Computing*, Springer, 2016, pp. 310–321.

[150] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *International Symposium on Code Generation and Optimization (CGO'07)*, IEEE, 2007, pp. 244–258.

[151] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "Daft: Decoupled acyclic fault tolerance," *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140, 2012.

[152] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, IEEE, 1999, pp. 84–91.

[153] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*. ACM, 2000, vol. 28.

[154] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, IEEE, 2007, pp. 317–326.

[155] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, IEEE, 2006, pp. 223–234.

[156] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, IEEE, 2007, pp. 297–306.

[157] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 145–154.

[158] B. Döbel and H. Härtig, "Can we put concurrency back into redundant multithreading?" In *Proceedings of the 14th International Conference on Embedded Software*, 2014, pp. 1–10.

[159] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Swat: An error resilient system," *Proceedings of SELSE*, 2008.

[160] S. K. S. Hari, "Low-cost hardware fault detection and diagnosis for multicore systems running multithreaded workloads," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2009.

[161] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010.

[162] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings 29th Annual International Symposium on Computer Architecture*, IEEE, 2002, pp. 123–134.

[163] P. Kamalinejad, C. Mahapatra, Z. Sheng, S. Mirabbasi, V. C. Leung, and Y. L. Guan, "Wireless energy harvesting for the internet of things," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 102–108, 2015.

[164] C.-W. Yau, T. T.-O. Kwok, C.-U. Lei, and Y.-K. Kwok, "Energy harvesting in internet of things," in *Internet of Everything*, Springer, 2018, pp. 35–79.

[165] J. Bito, R. Bahr, J. G. Hester, S. A. Nauroze, A. Georgiadis, and M. M. Tentzeris, "A novel solar and electromagnetic energy harvesting system with a 3-d printed package for energy efficient internet-of-things wireless sensors," *IEEE Transactions on Microwave Theory and Techniques*, vol. 65, no. 5, pp. 1831–1842, 2017.

[166] M. Gorlatova, J. Sarik, G. Grebla, M. Cong, I. Kymissis, and G. Zussman, "Movers and shakers: Kinetic energy harvesting for the internet of things," in *The 2014 ACM international conference on Measurement and modeling of computer systems*, 2014, pp. 407–419.

[167] Y.-W. Chong, W. Ismail, K. Ko, and C.-Y. Lee, "Energy harvesting for wearable devices: A review," *IEEE Sensors Journal*, vol. 19, no. 20, pp. 9047–9062, 2019.

[168] M. Magno, D. Kneubühler, P. Mayer, and L. Benini, "Micro kinetic energy harvesting for autonomous wearable devices," in *2018 International symposium on power electronics, electrical drives, automation and motion (SPEEDAM)*, IEEE, 2018, pp. 105–110.

[169] M. Magno and D. Boyle, "Wearable energy harvesting: From body to battery," in *2017 12th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, IEEE, 2017, pp. 1–6.

[170] Q. Cheng, Z. Peng, J. Lin, S. Li, and F. Wang, "Energy harvesting from human motion for wearable devices," in *10th IEEE International Conference on Nano/Micro Engineered and Molecular Systems*, IEEE, 2015, pp. 409–412.

[171] V. Leonov, "Energy harvesting for self-powered wearable devices," in *Wearable monitoring systems*, Springer, 2011, pp. 27–49.

[172] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of last-level on-chip cache using spin-torque transfer ram (stt ram)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 3, pp. 483–493, 2009.

[173] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy, "Future cache design using stt mrams for improved energy efficiency: Devices, circuits and architecture," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 492–497.

[174] M. R. Jokar, M. Arjomand, and H. Sarbazi-Azad, "Sequoia: A high-endurance nvm-based cache architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 954–967, 2015.

[175] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "I 2 wap: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 234–245.

[176] S. Mittal, J. S. Vetter, and D. Li, "Writesmoothing: Improving lifetime of non-volatile caches using intra-set wear-leveling," in *Proceedings of the 24th edition of the great lakes symposium on VLSI*, 2014, pp. 139–144.

[177] S. Mittal, J. S. Vetter, and D. Li, "Lastingnvcache: A technique for improving the lifetime of non-volatile caches," in *2014 IEEE Computer Society Annual Symposium on VLSI*, IEEE, 2014, pp. 534–540.

[178] S. Agarwal and H. K. Kapoor, "Improving the lifetime of non-volatile cache by write restriction," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1297–1312, 2019.

[179] P.-F. Chiu *et al.*, "A low store energy, low vddmin, nonvolatile 8t2r sram with 3d stacked rram devices for low power mobile applications," in *2010 Symposium on VLSI Circuits*, IEEE, 2010, pp. 229–230.

[180] X. Li *et al.*, "Design of nonvolatile sram with ferroelectric fets for energy-efficient backup and restore," *IEEE Transactions on Electron Devices*, vol. 64, no. 7, pp. 3037–3040, 2017.

[181] J. Singh and B. Raj, "Design and investigation of 7t2m-nvsram with enhanced stability and temperature impact on store/restore energy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1322–1328, 2019.

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

[183] C. Liu *et al.*, "A low power 4t2c nvsram with dynamic current compensation operation scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2469–2473, 2020.

[184] S.-S. Sheu *et al.*, "A reram integrated 7t2r non-volatile sram for normally-off computing application," in *2013 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, IEEE, 2013, pp. 245–248.

[185] S. Majumdar, S. K. Kingra, M. Suri, and M. Tikyani, "Hybrid cmos-oxram based 4t-2r nvsram with efficient programming scheme," in *2016 16th Non-Volatile Memory Technology Symposium (NVMTS)*, IEEE, 2016, pp. 1–4.

[186] 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)*, IEEE, 2001, pp. 3–14.

[187] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, IEEE, 1997, pp. 330–335.

[188] A. Kolli, "Architecting persistent memory systems," Ph.D. dissertation, 2017.

[189] Y. Liu *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[190] A. Lee *et al.*, "A reram-based nonvolatile flip-flop with self-write-termination scheme for frequent-off fast-wake-up nonvolatile processors," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 8, pp. 2194–2207, 2017.

[191] J.-M. Portal *et al.*, "An overview of non-volatile flip-flops based on emerging memory technologies," *J. Electron. Sci. Technol*, vol. 12, no. 2, pp. 173–181, 2014.

[192] S. Onkaraiah *et al.*, "Bipolar reram based non-volatile flip-flops for low-power architectures," in *10th IEEE International NEWCAS Conference*, IEEE, 2012, pp. 417–420.

[193] T. Na, K. Ryu, J. Kim, S. H. Kang, and S.-O. Jung, "A comparative study of stt-mtj based non-volatile flip-flops," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2013, pp. 109–112.

[194] X. Li *et al.*, "Lowering area overheads for fefet-based energy-efficient nonvolatile flip-flops," *IEEE Transactions on Electron Devices*, vol. 65, no. 6, pp. 2670–2674, 2018.

[195] D. Balsamo *et al.*, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[196] P.-F. Chiu *et al.*, "Low store energy, low vddmin, 8t2r nonvolatile latch and sram with vertical-stacked resistive memory (memristor) devices for low power mobile applications," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 6, pp. 1483–1496, 2012.

[197] T. Miwa *et al.*, "Nv-sram: A nonvolatile sram with backup ferroelectric capacitors," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 3, pp. 522–527, 2001.

[198] W. Wei, K. Namba, J. Han, and F. Lombardi, "Design of a nonvolatile 7t1r sram cell for instant-on operation," *IEEE transactions on nanotechnology*, vol. 13, no. 5, pp. 905–916, 2014.

[199] A. Lee *et al.*, "Rram-based 7t1r nonvolatile sram with 2x reduction in store energy and 94x reduction in restore energy for frequent-off instant-on applications," in *2015 Symposium on VLSI Circuits (VLSI Circuits)*, IEEE, 2015, pp. C76–C77.

[200] S. Masui *et al.*, "Design and applications of ferroelectric nonvolatile sram and flip-flop with unlimited read/program cycles and stable recall," in *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.*, IEEE, 2003, pp. 403–406.

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

[202] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 17–32.

[203] J. San Miguel *et al.*, "The eh model: Early design space exploration of intermittent processor architectures," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 600–612.

[204] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

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

[206] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, IEEE, 2012, pp. 392–397.

[207] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ACM, 2019, pp. 143–156.

[208] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 310–323.

[209] S. Yamamoto, Y. Shuto, and S. Sugahara, "Nonvolatile sram (nv-sram) using functional mosfet merged with resistive switching devices," in *2009 IEEE Custom Integrated Circuits Conference*, IEEE, 2009, pp. 531–534.

[210] J. Elliott, M. Hoemmen, and F. Mueller, "Tolerating silent data corruption in opaque preconditioners," *arXiv preprint arXiv:1404.5552*, 2014.

[211] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: Opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1153–1158.

[212] J. Henkel *et al.*, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2013, pp. 1–10.

[213] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

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

[215] Y. Zhang and C. Jung, "Featherweight soft error resilience for gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 245–262.

[216] J. Löff *et al.*, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021.

[217] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2016, pp. 101–111.

[218] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA, Tech. Rep., 2013.

[219] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[220] J. Chang *et al.*, "12.1 a 7nm 256mb sram in high-k metal-gate finfet technology with write-assist circuitry for low-v min applications," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2017, pp. 206–207.

[221] D. Matzke, "Will physical scalability sabotage performance gains?" *Computer*, vol. 30, no. 9, pp. 37–39, 1997.

[222] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 248–259.

[223] K. Krewell, "Cortex-a53 is arms next little thing," *Microprocessor Report*, vol. 11, no. 5, pp. 12–2, 2012.

[224] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, IEEE, 2007, pp. 286–296.

[225] J. L. Lo and S. J. Eggers, "Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 151–162, 1995.

[226] W. Chen *et al.*, "A 22nm 2.5 mb slice on-die l3 cache for the next generation xeonő processor," in *2013 Symposium on VLSI Circuits*, IEEE, 2013, pp. C132–C133.

[227] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 264, 2004.

[228] N. J. Wang, J. Quek, T. M. Rafacz, *et al.*, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks, 2004*, IEEE Computer Society, 2004, pp. 61–61.

[229] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, IEEE, 2012, pp. 1–12.

[230] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[231] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 187–196, 2013.

[232] J. Zeng, J. Jeong, and C. Jung, "Persistent processor architecture," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1075–1091.

[233] A. Ros and A. Jimborean, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.

[234] R. Sugumar, M. Shah, and R. Ramirez, "Marvell thunderx3: Next-generation arm-based server processor," *IEEE Micro*, vol. 41, no. 2, pp. 15–21, 2021.

[235] A. limited Corporation, *Cortex-a76 technique reference manual*, https://developer.arm.com/Processors/Cortex-A76, 2019.

[236] C. Kenyon and C. Capano, "Apple silicon performance in scientific computing," in *IEEE High Performance Extreme Computing Conference*, 2022, pp. 1–10.

[237] J. Zeng, T. Zhang, and C. Jung, "Compiler-directed whole-system persistence," in *Proceedings of the 51th Annual International Symposium on Computer Architecture*, 2024.

[238] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[239] S.-Y. Wu *et al.*, "A 16nm finfet cmos technology for mobile soc and computing applications," in *2013 IEEE International Electron Devices Meeting*, IEEE, 2013, pp. 9–1.

[240] S.-Y. Wu *et al.*, "A 7nm cmos platform technology featuring 4 th generation finfet transistors with a 0.027 um 2 high density 6-t sram cell for mobile soc applications," in *2016 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2016, pp. 2–6.

[241] S. Ainsworth and T. M. Jones, "Parallel error detection using heterogeneous cores," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2018, pp. 338–349.

[242] S. Ainsworth and T. M. Jones, "Paramedic: Heterogeneous parallel error correction," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2019, pp. 201–213.

[243] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–17.

[244] S.-Y. Huang *et al.*, "Rtailor: Parameterizing soft error resilience for mixed-criticality real-time systems," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2023.

[245] J. Zeng *et al.*, "Replaycache: Enabling volatile cachesfor energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 170–182.

[246] Y. Zhou, J. Zeng, J. Jeong, J. Choi, and C. Jung, "Sweepcache: Intermittence-aware cache on the cheap," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1059–1074.

[247] J. Choi, J. Zeng, D. Lee, C. Min, and C. Jung, "Write-light cache for energy harvesting systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

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

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

[250] J. Choi, H. Joe, and C. Jung, "Capos: Capacitor error resilience for energy harvesting systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4539–4550, 2022.

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

[252] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[253] Y. Chen, "Reram: History, status, and future," *IEEE Transactions on Electron Devices*, vol. 67, no. 4, pp. 1420–1433, 2020.

[254] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[255] V. V. Tyagi and D. Buddhi, "Pcm thermal storage in buildings: A state of art," *Renewable and sustainable energy reviews*, vol. 11, no. 6, pp. 1146–1166, 2007.

[256] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, IEEE, 2009, pp. 14–23.

[257] D. Sengupta *et al.*, "A framework for emulating non-volatile memory systemswith different performance characteristics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 317–320.

[258] N. S. Kim, C. Song, W. Y. Cho, J. Huang, and M. Jung, "Ll-pcm: Low-latency phase change memory architecture," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[259] Y. Huai *et al.*, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[260] A. Khvalkovskiy *et al.*, "Basic principles of stt-mram cell operation in memory arrays," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074 001, 2013.

[261] P. Chi, S. Li, Y. Cheng, Y. Lu, S. H. Kang, and Y. Xie, "Architecture design with stt-ram: Opportunities and challenges," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 109–114.

[262] K. Korgaonkar *et al.*, "Density tradeoffs of non-volatile memory as a replacement for sram based last level cache," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 315–327.

[263] B. Oh, N. Abeyratne, N. S. Kim, J. Ahn, R. G. Dreslinski, and T. Mudge, "Rethinking dram's page mode with stt-mram," *IEEE Transactions on Computers*, 2022.

[264] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 105–119.

[265] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, 2021, pp. 424–439.

[266] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 496–508.

[267] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories.," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[268] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 46–61, 2018.

[269] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2017.

[270] S. Haria, M. D. Hill, and M. M. Swift, "Mod: Minimally ordered durable data structures for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 775–788.

[271] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[272] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[273] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-Optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[274] W.-H. Kim, J. Seo, J. Kim, and B. Nam, "Clfb-tree: Cacheline friendly persistent b-tree for nvram," *ACM Trans. Storage*, 2018.

[275] I. Corporation, *Persistent memory programming*, https://pmem.io.

[276] B. Di, J. Liu, H. Chen, and D. Li, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 503–516.

[277] S. Liu, S. Mahar, B. Ray, and S. Khan, "Pmfuzz: Test case generation for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 487–502.

[278] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1187–1202.

[279] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 411–425.

[280] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory," in *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[281] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 413–422.

[282] I. Neal, A. Quinn, and B. Kasikci, "Hippocrates: Healing persistent memory bugs without doing any harm," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 401–414.

[283] M. Liu *et al.*, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.

[284] C. Cascaval *et al.*, "Software transactional memory: Why is it only a research toy?" *Communications of the ACM*, vol. 51, no. 11, pp. 40–46, 2008.

[285] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 468–482.

[286] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[287] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 111–124.

[288] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information sciences*, vol. 275, pp. 314–347, 2014.

[289] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[290] G. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero, "A content aware integer register file organization," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 314, 2004.

[291] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.

[292] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2003, pp. 26–34.

[293] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "Ilp versus tlp on smt," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999, 37–es.

[294] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, IEEE, 1999, pp. 54–58.

[295] A. Morari *et al.*, "Smt malleability in ibm power5 and power6 processors," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 813–826, 2012.

[296] Y. Zhang and W.-M. Lin, "Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors," *Microprocessors and Microsystems*, vol. 45, pp. 270–282, 2016.

[297] K. Patel, W. Lee, and M. Pedram, "Active bank switching for temperature control of the register file in a microprocessor," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 2007, pp. 231–234.

[298] M. H. Lipasti, B. R. Mestan, and E. Gunadi, "Physical register inlining," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 325, 2004.

[299] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 660–671.

[300] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 520–532.

[301] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 361–372.

[302] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 584–596.

[303] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[304] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 507–519.

[305] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[306] C. C. Minh *et al.*, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 69–80.

[307] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[308] JEDEC, *Cxl consortium and jedec sign mou agreement to advance dram and persistent memory technology*, https://www.jedec.org/news/pressreleases/cxl-consortium-and-jedec-sign-mou-agreement-advance-dram-and-persistent-memory, 2022.

[309] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 45–51.

[310] C. Mellor, *Cxl-led big memory taking over from age of san*, https://blocksandfiles.com/2022/06/20/cxl-led-big-memory/, 2022.

[311] C. Mellor, *Redis is ready for cxl memory pooling*, https://blocksandfiles.com/2022/07/20/redis-cxl-memory-pooling/, 2022.

[312] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access,{high-performance} memory disaggregation with {directcxl}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[313] G. Hinton, D. Sager, M. Upton, D. Boggs, *et al.*, "The microarchitecture of the pentiumő 4 processor," in *Intel technology journal*, Citeseer, 2001.

[314] V. R. K. Naresh, D. J. Palframan, and M. H. Lipasti, "Cram: Coded registers for amplified multiporting," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 196–205.

[315] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 316–325.

[316] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE micro*, vol. 19, no. 2, pp. 24–36, 1999.

[317] L. Gwennap, "Mips r10000 uses decoupled architecture," *Microprocessor Report*, vol. 8, no. 14, pp. 18–22, 1994.

[318] W. Wang and T. Dey, "A survey on arm cortex a processors," *Retrieved March*, 2011.

[319] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[320] D. Sima, "The design space of register renaming techniques," *IEEE micro*, vol. 20, no. 5, pp. 70–83, 2000.

[321] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[322] J. Zha, L. Huang, L. Wu, S.-a. Zheng, and H. Liu, "A consistency mechanism for nvm-based in-memory file systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 197–204.

[323] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "Nvm malloc: Memory allocation for nvram.," *Adms@ Vldb*, vol. 15, pp. 61–72, 2015.

[324] W. M. Jones, J. T. Daly, and N. DeBardeleben, "Application monitoring and checkpointing in hpc: Looking towards exascale systems," in *Proceedings of the 50th Annual Southeast Regional Conference*, 2012, pp. 262–267.

[325] B. Nicolae, A. Moody, G. Kosinovsky, K. Mohror, and F. Cappello, "Veloc: Very low overhead checkpointing in the age of exascale," *arXiv preprint arXiv:2103.02131*, 2021.

[326] G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, IEEE, 2012, pp. 1–6.

[327] J. Choi, J. Zeng, D. Lee, C. Min, and C. Jung, "Write-light cache for energy harvesting systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[328] S. Senni, L. Torres, G. Sassatelli, and A. Gamatie, "Non-volatile processor based on mram for ultra-low-power iot devices," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 2, pp. 1–23, 2016.

[329] K. Qiu *et al.*, "Design insights of non-volatile processors and accelerators in energy harvesting systems," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020, pp. 369–374.

[330] Y. Wang *et al.*, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *2012 Proceedings of the ESSCIRC (ESSCIRC)*, IEEE, 2012, pp. 149–152.

[331] A. Bhattacharyya, A. Somashekhar, and J. S. Miguel, "Nvmr: Non-volatile memory renaming for intermittent computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 1–13.

[332] I. Corporation, "Intelő 64 and ia-32 architectures software developers manual," 2023.

[333] C. SPARC International Inc, *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., 1994.

[334] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "When storage response time catches up with overall context switch overhead, what is next?" *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4266–4277, 2020.

[335] D. Waddington and J. Harris, "Software challenges for the changing storage landscape," *Communications of the ACM*, vol. 61, no. 11, pp. 136–145, 2018.

[336] K. Suo, Y. Shi, C.-C. Hung, and P. Bobbie, "Quantifying context switch overhead of artificial intelligence workloads on the cloud and edges," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1182–1189.

[337] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[338] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 169–182.

[339] A. Limaye and T. Adegbija, "A workload characterization of the spec cpu2017 benchmark suite," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2018, pp. 149–158.

[340] M. Hassan, C. H. Park, and D. Black-Schaffer, "A reusable characterization of the memory system behavior of spec2017 and spec2006," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 2, pp. 1–20, 2021.

[341] memcached organization, *Memcachd - a distributed memory object caching system*, http://memcached.org/, 2017.

[342] B. Aker, *Libmemcached - a open source c/c++ library for the memcached server*, https://libmemcached.org/libMemcached.html, 2011.

[343] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Soft error and energy consumption interactions: A data cache perspective," in *Proceedings of the 2004 international symposium on Low power electronics and design*, 2004, pp. 132–137.

[344] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 141–152.

[345] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 1, pp. 1–24, 2009.

[346] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[347] D. Nayak, D. P. Acharya, and K. Mahapatra, "An improved energy efficient sram cell for access over a wide frequency range," *Solid-State Electronics*, vol. 126, pp. 14–22, 2016.

[348] D. Pandiyan and C.-J. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2014, pp. 171–180.

[349] S. Lee, M. Kwon, G. Park, and M. Jung, "Lightpc: Hardware and software co-design for energy-efficient full system persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 289–305.

[350] Y. Zhu *et al.*, "Carbon-based supercapacitors produced by activation of graphene," *science*, vol. 332, no. 6037, pp. 1537–1541, 2011.

[351] D. Pech *et al.*, "Ultrahigh-power micrometre-sized supercapacitors based on onion-like carbon," *Nature nanotechnology*, vol. 5, no. 9, pp. 651–654, 2010.

[352] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[353] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[354] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 466–478.

[355] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 836–848.

[356] S. Pandey, A. K. Kamath, and A. Basu, "Gpm: Leveraging persistent memory from a gpu," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 142–156.

[357] A. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "Asap: Architecture support for asynchronous persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 306–319.

[358] A. Freij, H. Zhou, and Y. Solihin, "Secpb: Architectures for secure non-volatile memory with battery-backed persist buffers," in *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA-29)*, 2023.

[359] S. Pandey, A. K. Kamath, and A. Basu, "Scoped buffered persistency model for gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 688–701.

[360] C. Ye *et al.*, "Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory," 2023.

[361] C. Ye, Y. Xu, X. Shen, H. Jin, X. Liao, and Y. Solihin, "Preserving addressability upon gc-triggered data movements on non-volatile memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 2, pp. 1–26, 2022.

[362] G. Hodgkins, Y. Xu, S. Swanson, and J. Izraelevitz, "Zhuque: Failure is not an option,{its} an exception," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 833–849.

[363] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 613–626, 2017.

[364] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[365] G. Liu, K. Li, Z. Xiao, and R. Wang, "Ps-oram: Efficient crash consistency support for oblivious ram on nvm," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 188–203.

[366] S. Aggarwal *et al.*, "Demonstration of a reliable 1 gb standalone spin-transfer torque mram for industrial applications," in *2019 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2019, pp. 2–1.

[367] F. S. M. S. Limited, https://www.fujitsu.com/jp/group/fsm/en/products/reram/ReRAM_whitepaper_2023e.pdf, 2023.

[368] R. Stevens, J. Ramprakash, P. Messina, M. Papka, and K. Riley, "Aurora: Argonnes next-generation exascale supercomputer," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2019.

[369] M. Ferdman *et al.*, "A case for specialized processors for scale-out workloads," *IEEE Micro*, vol. 34, no. 3, pp. 31–42, 2014.

[370] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[371] P. Lotfi-Kamran *et al.*, "Scale-out processors," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 500–511, 2012.

[372] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX annual technical conference (USENIX ATC 20)*, 2020, pp. 205–218.

[373] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2014, pp. 1–12.

[374] C. Consortium, *Compute express link: The breakthrough cpu-to-device interconnect*, https://www.computeexpresslink.org/, 2023.

[375] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 643–656.

[376] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 96–109.

[377] I. Corporation, *Eadr: New opportunities for persistent memory applications*, https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[378] A. M. D. Inc, *Amd epyc 9654p*, https://www.amd.com/en/products/cpu/amd-epyc-9654p, 2023.

[379] Samsung, *Samsung electronics unveils industrys highest-capacity 12nm-class 32gb ddr5 dram, ideal for the ai era*, https://news.samsung.com/global/samsung-electronics-unveils-industrys-first-and-highest-capacity-12nm-class-32gb-ddr5-dram-ideal-for-the-ai-era, 2023.

[380] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: Efficient, htm-compatible persistent transactions," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 59–74.

[381] H. Gorjiara, G. H. Xu, and B. Demsky, "Yashme: Detecting persistency races," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 830–845.

[382] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 652–665.

[383] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan, "Lazy release persistency," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1173–1186.

[384] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization*, IEEE, 2008, pp. 35–46.

[385] A. M. D. Inc, *Amd epyc 9754*, https://www.amd.com/en/products/cpu/amd-epyc-9754, 2023.

[386] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 452–465.

[387] J. F. Peters, M. Baumann, B. Zimmermann, J. Braun, and M. Weil, "The environmental impact of li-ion batteries and the role of key parameters–a review," *Renewable and Sustainable Energy Reviews*, vol. 67, pp. 491–506, 2017.

[388] S. K. Mohapatra, P. Nayak, S. Mishra, and S. K. Bisoy, "Green computing: A step towards eco-friendly computing," in *Emerging trends and applications in cognitive computing*, IGI global, 2019, pp. 124–149.

[389] A. Köhler and L. Erdmann, "Expected environmental impacts of pervasive computing," *Human and Ecological Risk Assessment*, vol. 10, no. 5, pp. 831–852, 2004.

[390] F. S. Foundation, *Gnu c standard library*, https://www.gnu.org/software/libc/.

[391] L. Foundation, *A new c++ standard library for llvm*, https://libcxx.llvm.org/.

[392] L. Foundation, *Compiler-rt runtime libraries*, https://compiler-rt.llvm.org/.

[393] L. Foundation, *A llvm-compatible unwinder*, https://bcain-llvm.readthedocs.io/projects/libunwind/en/latest/.

[394] T. of Bits, *Remill: A static binary translator from machine code instructions to llvm bitcode*, https://github.com/lifting-bits/remill, 2024.

[395] R. H. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[396] I. Arsovski and R. Wistort, "Self-referenced sense amplifier for across-chip-variation immune sensing in high-performance content-addressable memories," in *IEEE Custom Integrated Circuits Conference 2006*, IEEE, 2006, pp. 453–456.

[397] I. Corporation, *Deprecating the pcommit instruction*, https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html, 2016.

[398] L. Soares and M. Stumm, "{Flexsc}: Flexible system call scheduling with {exception-less} system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[399] S. S. Stone, K. M. Woley, and M. I. Frank, "Address-indexed memory disambiguation and store-to-load forwarding," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, IEEE, 2005, 12–pp.

[400] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," vol. 48, no. 4, pp. 473–484, 2013.

[401] I. limited Corporation, *Intel 64 and ia-32 architectures optimization reference manual*, https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, 2020.

[402] R. Bera *et al.*, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 1–18.

[403] I. Corporation, *Memory performance in a nutshell*, https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html, 2016.

[404] J. Zeng, S.-Y. Huang, J. Liu, and C. Jung, "Soft error resilience at near-zero cost," in *2024 38th ACM International Conference on Supercomputing (ICS)*, ACM, 2024, pp. 176–187.

[405] A. Hoseinghorban, A. M. H. H. Monazzah, M. Bazzaz, B. Safaei, and A. Ejlali, "Coach: Consistency aware check-pointing for nonvolatile processor in energy harvesting systems," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[406] Y. Sun *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.

[407] H. Li *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[408] P. Chi, C. Xu, T. Zhang, X. Dong, and Y. Xie, "Using multi-level cell stt-ram for fast and energy-efficient local checkpointing," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2014, pp. 301–308.

[409] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

PUBLICATIONS

Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung, "Compiler-Directed Soft Error Resilience for Lightweight GPU Register File Protection," International Conference on Programming Language Design and Implementation (PLDI), June, 2020.

Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung, "Turnpike: Lightweight Soft Error Resilience for In-Order Cores," International Symposium on Microarchitecture (MICRO), October, 2021.

Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay P. Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung, "ReplayCache: Enabling Volatile Caches for Energy Harvesting Systems," International Symposium on Microarchitecture (MICRO), October, 2021.

Jungi Jeong, Jianping Zeng, Changhee Jung, "Capri: Compiler and Architecture Support for Whole-System Persistence," International Symposium on High-Performance Parallel and Distributed Computing (HPDC), June, 2022.

Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, Changhee Jung, "Write-Light Cache for Energy Harvesting Systems," International Symposium on Computer Architecture (ISCA), June, 2023.

Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, Changhee Jung, "SweepCache: Intermittence-Aware Cache on the Cheap," International Symposium on Microarchitecture (MICRO), October, 2023.

Jianping Zeng, Jungi Jeong, Changhee Jung, "Persistent Processor Architecture," International Symposium on Microarchitecture (MICRO), October, 2023.

Shao-Yu Huang, Jianping Zeng, Xuanliang Deng, Sen Wang, Ashrarul Haq Sifat, Burhanuddin Bharmal, Jia-Bin Huang, Ryan Williams, Haibo Zeng and Changhee Jung, "RTailor: Parameterizing Soft Error Resilience for Mixed-Criticality Real-Time Systems," International Real-Time Systems Symposium (RTSS), December, 2023.

Jianping Zeng, Shao-Yu Huang, Jiuyang Liu, Changhee Jung, "Soft Error Resilience at Near-Zero Cost," International Conference on Supercomputing (ICS), June 2024.

Jianping Zeng, Tong Zhang, Changhee Jung, "Compiler-Directed Whole-System Persis-

tence," International Symposium on Computer Architecture (ISCA), June, 2024.
Yuchen Zhou, Jianping Zeng, Changhee Jung, "LightWSP: Whole-System Persistence on the Cheap," International Symposium on Microarchitecture (MICRO), November, 2024.